[Building a Command Line Program 2](#_Toc486341770)

[Accepting Command Line Arguments 3](#_Toc486341771)

[Reading the Argument Values 3](#_Toc486341772)

[Saving the Argument Values in Variables 4](#_Toc486341773)

[Reading a File 6](#_Toc486341774)

[Refactoring to Improve Modularity and Error Handling 8](#_Toc486341775)

[Separation of Concerns for Binary Projects 9](#_Toc486341776)

[Extracting the Argument Parser 10](#_Toc486341777)

[Grouping Configuration Values 10](#_Toc486341778)

[The Tradeoffs of Using clone 12](#_Toc486341779)

[Creating a Constructor for Config 13](#_Toc486341780)

[Fixing the Error Handling 14](#_Toc486341781)

[Improving the Error Message 14](#_Toc486341782)

[Returning a Result from new Instead of Calling panic! 15](#_Toc486341783)

[Calling Config::new and Handling Errors 17](#_Toc486341784)

[Extracting Logic Functionality from main 18](#_Toc486341785)

[Returning Errors from the run Function 19](#_Toc486341786)

[Handling Errors Returned from run in main 21](#_Toc486341787)

[Splitting Code into a Library Crate 21](#_Toc486341788)

[Testing the Library’s Functionality 24](#_Toc486341789)

[Writing a Failing Test 24](#_Toc486341790)

[Writing Code to Pass the Test 27](#_Toc486341791)

[Iterating Through Lines with the lines method 27](#_Toc486341792)

[Searching Each Line for the Query 28](#_Toc486341793)

[Storing Matching Lines 28](#_Toc486341794)

[Using the search Function in the run Function 30](#_Toc486341795)

[Working with Environment Variables 31](#_Toc486341796)

[Writing a Failing Test for the Case-Insensitive search Function 31](#_Toc486341797)

[Implementing the search\_case\_insensitive Function 33](#_Toc486341798)

[Writing Error Messages to stderr Instead of stdout 38](#_Toc486341799)

[Checking Where Errors are Written to 38](#_Toc486341800)

[Printing Errors to Standard Error 38](#_Toc486341801)

[Summary 40](#_Toc486341802)

Chapter 12

An I/O Project Using Environment Variables and Standard Error

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

Rust’s speed, safety, single binary output, and cross-platform support make it a good language for creating command line tools, so for our project we’ll make our own version of the classic command line tool grep. Grep is an acronym for “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 a filename and a string as its arguments, then reads the file and finds lines in that file that contain the string argument. It’ll then print out 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 in order to allow the user to configure the behavior of our tool. We'll print to the standard error console stream (stderr) instead of standard output (stdout) so that, for example, the user can choose to redirect successful output to a file while still seeing error messages on the screen.

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 to help you understand a real-world project like ripgrep.

This project will bring together a number of concepts you’ve learned so far:

Organizing code (using what we 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.

Building a Command Line Program

Let’s create a new project with, as always, cargo new. We’re calling our project minigrep to distinguish from the grep tool that you may already have on your system:

$ cargo new --bin minigrep

Created binary (application) `minigrep` project

$ cd

minigrep

Accepting Command Line Arguments

Our first task is to make minigrep able to 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. There are some existing libraries on crates.io that can help us accept command line arguments, but since you’re learning, let’s implement this ourselves.

Reading the Argument Values

We first need to make sure our program is able to get the values of command line arguments we pass to it, for which we’ll need a function provided in Rust’s standard library: std::env::args. This function returns an iterator of the command line arguments that were given to our program. We haven’t discussed iterators yet, and we’ll cover them fully in Chapter 13, but for our purposes now we only need to know two things 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 of the elements the iterator produces.

Let’s give it a try: use the code in Listing 12-1 to allow your minigrep program to read any command line arguments passed it and then collect the values into a vector.

Filename: src/main.rs

use std::env;

fn main() {

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

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

}

Listing 12-1: Collect the command line arguments into a vector and print them out

First, we bring the std::env module into scope with a use statement so that we can use its args function. Notice the std::env::args function is nested in two levels of modules. As we talked about 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 itself. This lets us easily use other functions from std::env. It’s also less ambiguous than adding use std::env::args; then calling the function with just args; that might easily be mistook for a function that’s defined in the current module.

start box

The args Function and Invalid Unicode

Note that std::env::args will panic if any argument contains invalid Unicode. If you need 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 of the values produced by the iterator. The collect function can be used to create many kinds of collections, so we explicitly annotate the type of args to specify that we want a vector of strings. Though 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 what kind of collection you want.

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

$ cargo run

["target/debug/minigrep"]

$ cargo run needle haystack

...snip...

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

You may notice that the first value in the vector is “target/debug/minigrep”, which is the name of our binary. The reasons for this are out of the scope of this chapter, but we’ll need to remember this as we save the two arguments we need.

Saving the Argument Values in Variables

Printing out the value of the vector of arguments has illustrated that the program is able to access the values specified as command line arguments. That’s not what we actually our end goal, though. We want to save the values of the two arguments in variables so that we can use the values in our program. Let’s do that as shown in Listing 12-2:

Filename: 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: Create variables to hold the query argument and filename argument

As we saw when we printed out the vector, the program’s name takes up the first value in the vector at args[0], so that 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’re temporarily printing out the values of these variables, again to prove to ourselves that our code is working as we intend. Let’s try running this program again with the arguments test and sample.txt:

$ cargo run test sample.txt

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

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

Searching for test

In file sample.txt

Great, it’s 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, but for now we’ll ignore that and work on adding file reading capabilities instead.

Reading a File

Next, we’re going to add functionality to read the file that 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 that 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?”:

Filename: 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: The poem “I’m nobody! Who are you?” by Emily Dickinson that will make a good test case

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

Filename: 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 for dealing with files, and std::io::prelude::\* contains various traits that are useful when 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 things: 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 out the value of contents after the file is read, so that we can check that our program is working so far.

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

$ cargo run the poem.txt

Finished debug [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! Our code read in and printed out the content of the file. We’ve got a few flaws though. 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. While our program is still small, these flaws aren’t a big problem, but as our program grows, it will be harder to fix them cleanly. It’s good practice to begin refactoring early on when developing a program, as it’s much easier to refactor smaller amounts of code, so we’ll do that now.

Refactoring to Improve Modularity and Error Handling

There are four problems that we’d like to fix to improve our program, and they have to do with the way the program is structured and how it’s handling potential errors.

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

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

The third problem is that we’ve used expect to print out an error message when opening the file fails, but the error message only says file not found. There are a number of ways that opening a file can fail 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 advice!

Fourth, we use expect repeatedly to deal with different errors, and if the user runs our programs 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 better if all our error handling code was in one place so that future maintainers only have 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 help us to ensure that we’re printing messages that will be meaningful to our end users.

Let’s address these problems by refactoring our project.

Separation of Concerns for Binary Projects

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

Split your program into both a main.rs and a lib.rs and move your program’s logic into 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 into lib.rs as well.

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

If run returns an error, handling that error

This pattern is all about separating concerns: main.rs handles running the program, and lib.rs handles all of 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 re-work our program by following this process.

Extracting the Argument Parser

First, we’ll extract the functionality for parsing arguments. Listing 12-5 shows the new start of main that calls a new function parse\_config, which we’re still going to define in src/main.rs for the moment:

Filename: 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: Extract 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 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, as 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 up into individual parts again. This is a sign that perhaps we don’t have the right abstraction yet.

Another indicator that 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. This 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:

Filename: 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: query,

filename: 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’ve now chosen to 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, though, which means we’d violate Rust’s borrowing rules if Config tried to take ownership of the values in args.

There are a number of different ways we could manage the String data, and 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 does take more time and memory than storing a reference to the string data. However, cloning the data also makes our code very straightforward since we don’t have to manage the lifetimes of the references, so in this circumstance giving up a little performance to gain simplicity is a worthwhile trade-off.

The Tradeoffs of Using clone

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

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

Our code now 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 into a function named new that is associated with the Config struct. Making this change will make our code more idiomatic: we can create instances of types in the standard library like 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’ll need to make:

Filename: 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: query,

filename: 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 makes the new function associated with Config. Try compiling this again to make sure it works.

Fixing the Error Handling

Now we’ll work on fixing our error handling. Recall that we mentioned 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 3 items. Try running the program without any arguments; it will look like this:

$ cargo run

Finished debug [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 that states index out of bounds: the len is 1 but the index is 1 is an error message intended for programmers, and won’t really 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’re adding a check in the new function that will check that the slice is long enough before accessing index 1 and 2. If the slice isn’t long enough, the program panics, with a better error message than the index out of bounds message:

Filename: 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 is similar to the Guess::new function we wrote in Listing 9-8, where panic! was called 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 3 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 try running our program without any arguments again and see what the error looks like now:

$ cargo run

Finished debug [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 a bunch of extra information we don’t want to give to our users. So 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 we discussed in Chapter 9. Instead, we can use the other technique you also learned about in Chapter 9: returning a Result that can indicate either success or an error.

Returning a Result from new Instead of Calling panic!

We can choose to 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 that 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 you need to make to the return value of Config::new and the body of the function needed to return a Result:

Filename: 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: query,

filename: filename,

})

}

}

Listing 12-9: Return 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.

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.

By having Config::new return an Err value, it 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

In order 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’re also going to 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 our program ended with an error state.

Filename: 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’re using 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 be covering closures in more detail in Chapter 13. What you need to know for now is 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.

We’ve added a new use line to import process from the standard library. The code in the closure that will get run in the error case is only two lines: we print out the err value, 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, with the exception that we no longer get all the extra output. Let’s try it:

$ cargo run

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

Finished debug [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 Functionality from main

Now we’re done refactoring our configuration parsing; let’s turn to our program’s logic. As we laid out in the “Separation of Concerns for Binary Projects” section, we’re going to extract a function named run that will hold all of the logic currently in the main function not involved with setting up configuration or handling errors. Once we’re done, main will be concise and easy to verify by inspection, and we’ll be able to write tests for all of the other logic.

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

Filename: 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 like 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 the logic around handling errors in a user-friendly way into main. Listing 12- 12 shows the changes you need to make to the signature and body of run:

Filename: 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 big changes here. First, we’re changing 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 our error type, we’re using the trait object Box<Error> (and we’ve brought std::error::Error into scope with a use statement at the top). We’ll be covering 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.

The second change we’re making is removing the calls to expect in favor of ?, like we talked about in Chapter 9. Rather than panic! on an error, this will return the error value from the current function for the caller to handle.

Thirdly, 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 may 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, it will compile, but with 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 is telling us that our code ignores the Result value, which might be indicating that there was an error. We’re not checking to see if there was an error or not, though, and the compiler is reminding us that we probably meant to have some error handling code here! Let’s rectify that now.

Handling Errors Returned from run in main

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

Filename: 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 to check whether run returns an Err value, rather than unwrap\_or\_else, and call process::exit(1) if it does. run doesn’t return a value that we want to unwrap like 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 as it would only be ().

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

Splitting Code into a Library Crate

This is looking pretty good so far! Now we’re going to split the src/main.rs file up and put some code into src/lib.rs so that we can test it and have a smaller main file.

Let’s move everything that isn't the main function from src/main.rs to a new file, 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):

Filename: 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 as shown in Listing 12-14:

Filename: 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. With that, all the functionality should be connected and should work. Give it a cargo run and make sure everything is wired up 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 our 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 hard with our old code, but is easy with our new code: write some tests!

Testing the Library’s Functionality

Now that we’ve extracted the logic into src/lib.rs and left all the argument parsing and error handling in src/main.rs, it’s much easier for us to write tests for the core functionality of our code. We can call our functions directly with various arguments and check return values without having to call our binary from the command line.

In this section, we’re going to follow the Test Driven Development (TDD) process. This is a software development technique that follows this set of 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!

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

We’re going to test drive the implementation 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’re going to add this functionality in a function called search.

Writing a Failing Test

First, since we don’t really need them any more, let’s remove the println! statements from both src/lib.rs and src/main.rs. Then we’ll add a test module with a test function like we did in Chapter 11. The test function specifies the behavior we’d like 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:

Filename: 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

The string we are searching for is “duct” in this test. 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 though, since this test doesn’t even compile–the search function doesn't exist yet! So now we’ll add just enough code to get the tests to compile and run: a definition of the search function that always returns an empty vector, as shown in Listing 12-16. Once we have this, the test should compile and fail because an empty vector doesn’t match a vector containing the line "safe, fast, productive.".

Filename: 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 that 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. Remember from Chapter 10 that the lifetime parameters specify which argument lifetime is connected to the lifetime of the return value. In this case, we’re indicating that the returned vector should contain string slices that reference slices of the argument contents (rather than the argument query).

In other words, we’re telling 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 in order for the reference to be valid; if the compiler assumed we were making string slices of query rather than contents, it would do its safety checking incorrectly.

If we tried to compile this function without lifetimes, we would 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 this may still feel strange, but will get easier over time. You may want to compare this example with the Lifetime Syntax section in Chapter 10.

Now let’s try running our test:

$ cargo test

...warnings...

Finished debug [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, our 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 if 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 take each step at a time, 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:

Filename: 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 be talking about iterators in depth in Chapter 13, but we’ve already seen this way of using an iterator in Listing 3-6, where we used a for loop with an iterator to run some code on each item in a collection.

Searching Each Line for the Query

Next, we’ll add functionality to check if the current line contains the query string. Luckily, strings have another helpful method named contains that does this for us! Add the contains method to the search function as shown in Listing 12-18:

Filename: 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 if the line contains the string in query

Storing Matching Lines

Finally, we 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:

Filename: 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 that we can return them

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

$ cargo test

running 1 test

test test::one\_result ... ok

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

Our test passed, great, it works!

Now that our test is passing, we could consider opportunities for refactoring the implementation of the search function while keeping the code that passes the tests, in order to maintain the same functionality. The code in the search function isn’t too bad, but it isn’t taking advantage of some useful features of iterators. We’ll be coming back to this example in Chapter 13 where we’ll explore iterators in detail and see how to improve it.

Using the search Function in the run Function

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

Filename: 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 again using a for loop to get each line returned from search, and the code that we run for each line prints it out.

Now our whole program should be working! 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 debug [unoptimized + debuginfo] target(s) in 0.38 secs

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

How public, like a frog

Cool! Next, how about a word that will match multiple lines, like “the”:

$ cargo run the poem.txt

Finished debug [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, like “monomorphization”:

$ cargo run monomorphization poem.txt

Finished debug [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’re going to briefly demonstrate how to work with environment variables and how to print to standard error, both of which are useful when writing command line programs. Feel free to move on to Chapter 13 if you’d like at this point.

Working with Environment Variables

We’re going to improve our tool with an extra feature: an option for case insensitive searching that the user can turn on via an environment variable. We could make this a command line option and require that users enter it each time they want it to apply, but instead we’re going to use an environment variable. This 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 will call when the environment variable is on.

We’re going to continue following 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 be clearer about the differences between the two tests, as shown in Listing 12-20:

Filename: 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 added a new line to the contents, to be searched, that contains the text “Duct tape”, which shouldn’t match “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 going to add, the query “rUsT” should match both the line containing “Rust:” with a capital R and also the line “Trust me.” even though both of those 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 array in the same way that we did for the search function in Listing 12-16 in order 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 that whatever the case of the input arguments, they will be the same case when we check whether the line contains the query.

Filename: 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 both 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 that no matter if 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 does not 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 if it contains query to lowercase all characters. Now that we’ve converted both line and query to lowercase, we’ll find matches no matter what the case of the query.

Let’s see if this implementation passes the tests:

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

Running target/debug/deps/minigrep-e58e9b12d35dc861

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! Now, let’s actually call the new search\_case\_insensitive function from the run function. First, we’re going to add a configuration option for switching between case sensitive and case insensitive search to the Config struct:

Filename: src/lib.rs

pub struct Config {

pub query: String,

pub filename: String,

pub case\_sensitive: bool,

}

We add the case\_sensitive field that holds a boolean. Then we need our 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:

Filename: 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 actually 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’re going to use the var method from the env module to check for an environment variable named CASE\_INSENSITIVE, as shown in Listing 12-23:

Filename: 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: query,

filename: filename,

case\_sensitive: case\_sensitive,

})

}

}

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

Here, we create a new variable case\_sensitive. In order 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 to see if 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 it 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 that 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 debug [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”, and we should get lines that contain “to” that might have uppercase letters:

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

Finished debug [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 both 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 a command line argument as well as through the environment variable, and decide which should take precedence the program is run with contradictory values.

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

Writing Error Messages to stderr Instead of stdout

At the moment we’re writing all of our output to the terminal with the println! function. Most terminals provide two kinds of output: standard out for general information, and standard error for error messages. This distinction enables users to choose whether to direct a 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 out, though, so we have to use something else in order to print to standard error.

Checking Where Errors are Written to

First, let’s observe how all content printed by minigrep is currently being written to standard out, including error messages that we want to write to standard error instead. We’ll do that by redirecting the standard output stream to a file while we also intentionally cause 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 that we can still see error messages on the screen even if we choose to 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 the 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're not going to pass any arguments, which should cause an error:

$ cargo run > output.txt

The > syntax tells the shell to write the contents of standard out 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. Let’s see what output.txt contains:

Problem parsing arguments: not enough arguments

Yup, our error message is being printed to standard out. 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 out in this way. We’ll change that.

Printing Errors to Standard Error

Let’s change how error messages are printed using the code in Listing 12-24. 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 so that these spots use eprintln! instead:

Filename: 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) = greprs::run(config) {

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

process::exit(1);

}

}

Listing 12-24: Writing error messages to stderr instead of stdout using eprintln!

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

$ cargo run > output.txt

Problem parsing arguments: not enough arguments

Now we see our error on the screen and output.txt contains nothing, which is the behavior expected of command line programs.

If we run the program again with arguments that don’t cause an error, but still redirect standard out to a file:

$ cargo run to poem.txt > output.txt

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

Filename: output.txt

Are you nobody, too?

How dreary to be somebody!

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

Summary

In this chapter, we’ve recapped on some of the major concepts 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 with stderr, you’re now prepared to write command line applications. By using the concepts from previous chapters, your code will be well-organized, be able to store data effectively in the appropriate data structures, handle errors nicely, and be well tested.

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