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 search 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 file path and a string. Then it reads the file, finds lines in that file that contain the string argument, and prints those lines.

Along the way, we’ll show how to make our command line tool use the terminal features that many other 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 error messages to the standard error console stream (stderr) instead of standard output (stdout) so that, 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 will be fairly simple, but this chapter will give you some of the background knowledge you need to understand a real-world project such as ripgrep.

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

* Organizing code (Chapter 7)
* Using vectors and strings (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 Chapter 13 and Chapter 18 will cover in detail.

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 minigrep

Created binary (application) `minigrep` project

$ cd minigrep

The first task is to make minigrep accept its two command line arguments: the file path and a string to search for. That is, we want to be able to run our program with cargo run, two hyphens to indicate the following arguments are for our program rather than for cargo, 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. Some existing libraries on https://crates.io can help with writing a program that accepts command line arguments, but because you’re just learning this concept, let’s implement this capability ourselves.

Reading the Argument Values

To enable minigrep to read the values of command line arguments we pass to it, we’ll need the std::env::args function provided in Rust’s standard library. This function returns an iterator of the command line arguments passed to minigrep. We’ll cover iterators fully in Chapter 13. For now, you only need to know two details about iterators: iterators produce a series of values, and we can call the collect method on an iterator to turn it into a collection, such as a vector, that contains all the elements the iterator produces.

The code in Listing 12-1 allows 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();

dbg!(args);

}

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, we’ve chosen to bring the parent module into scope rather than the function. By doing so, 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.

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 an iterator that produces 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.

On the first line of main, we call env::args, and we 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 you 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 macro. Let’s try running the code first with no arguments and then with two arguments:

$ cargo run

--snip--

[src/main.rs:5] args = [

"target/debug/minigrep",

]

$ cargo run -- needle haystack

--snip--

[src/main.rs:5] args = [

"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 you want to print it in messages or change the 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

The program is currently 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 file\_path = &args[2];

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

println!("In file {}", file\_path);

}

Creating variables to hold the query argument and file path 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 arguments 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 file path, so we put a reference to the second argument in the variable file\_path.

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

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

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

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 specified in the file\_path argument. First we need a sample file to test it with: we’ll use a file 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!

A poem by Emily Dickinson makes a good test case.

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

src/main.rs

use std::env;

1 use std::fs;

fn main() {

--snip--

println!("In file {}", file\_path);

2 let contents = fs::read\_to\_string(file\_path)

.expect("Should have been able to read the file");

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

}

Reading the contents of the file specified by the second argument

First we bring in a relevant part of the standard library with a use statement: we need std::fs to handle files 1.

In main, the new statement fs::read\_to\_string takes the file\_path, opens that file, and returns an std::io::Result<String> of the file’s contents 2.

After that, we again add 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 3.

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

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

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

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 contents of the file. But the code has a few flaws. At the moment, 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. The program is still small, so these flaws aren’t a big problem, but as the program grows, it will be harder to fix them cleanly. It’s a 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 reads files. As our program grows, 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 file\_path are configuration variables to our program, variables like 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 reading the file fails, but the error message just prints Should have been able to read the file. Reading a file can fail in a number of ways: for example, the file could be missing, or we might not have permission to open it. Right now, regardless of the situation, we’d print the same error message for everything, which wouldn’t give the user any 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 were in one place so future maintainers had only one place to consult the code if the error-handling logic needed 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 guidelines for splitting the separate concerns of a binary program when main starts getting large. This process has the following steps:

* Split your program into a main.rs file and a lib.rs file and move your program’s logic to lib.rs.
* As long as 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 the following:

* 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 you can’t test the main function directly, this structure lets you test all of your program’s logic by moving it into functions in lib.rs. The 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, file\_path) = parse\_config(&args);

--snip--

}

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

let query = &args[1];

let file\_path = &args[2];

(query, file\_path)

}

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 file\_path 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 file\_path 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, to help identify the cause of problems when they occur.

Grouping Configuration Values

We can take another small step to improve the parse\_config 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 by grouping the two values into a tuple; we’ll instead 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.

Listing 12-6 shows the improvements to the parse\_config function.

src/main.rs

fn main() {

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

1 let config = parse\_config(&args);

println!("Searching for {}",2 config.query);

println!("In file {}",3 config.file\_path);

let contents = fs::read\_to\_string(4config.file\_path)

.expect("Should have been able to read the file");

--snip--

}

5 struct Config {

query: String,

file\_path: String,

}

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

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

8 let file\_path = args[2].clone();

Config { query, file\_path }

}

Refactoring parse\_config to return an instance of a Config struct

We’ve added a struct named Config defined to have fields named query and file\_path 5. The signature of parse\_config now indicates that it returns a Config value 6. 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.

There are a number of ways we could manage the String data; the easiest, though somewhat inefficient, route is to call the clone method on the values 7 8. 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.

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 you’ll make these copies only once and your file path and query string are very small. It’s better to have a working program that’s a bit inefficient than to try to hyperoptimize code on your first pass. As you become more experienced with Rust, it’ll be easier to start with the most efficient solution, but for now, it’s perfectly acceptable to call clone.

We’ve updated main so it places the instance of Config returned by parse\_config into a variable named config 1, and we updated the code that previously used the separate query and file\_path variables so it now uses the fields on the Config struct instead 2 3 4.

Now our code more clearly conveys that query and file\_path are related and that 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 and placed it in the parse\_config function. Doing so helped us see that the query and file\_path 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 file\_path 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 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. Similarly, 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();

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

--snip--

}

--snip--

2 impl Config {

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

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

let file\_path = args[2].clone();

Config { query, file\_path }

}

}

Changing parse\_config into Config::new

We’ve updated main where we were calling parse\_config to instead call Config::new 1. We’ve changed the name of parse\_config to new 3 and moved it within an impl block 2, 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

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

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

Running `target/debug/minigrep`

thread 'main' panicked at src/main.rs:27:21:

index out of bounds: the len is 1 but the index is 1

note: run with `RUST\_BACKTRACE=1` environment variable to display

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 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 index 2. If the slice isn’t long enough, the program panics and displays a better error message.

src/main.rs

--snip--

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

if args.len() < 3 {

panic!("not enough arguments");

}

--snip--

Adding a check for the number of arguments

This code is similar to the Guess::new function we wrote in Listing 9-13, 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

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

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

Running `target/debug/minigrep`

thread 'main' panicked at src/main.rs:26:13:

not enough arguments

note: run with `RUST\_BACKTRACE=1` environment variable to display

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 the technique we used in Listing 9-13 isn’t the best one to use here: a call to panic! is more appropriate for a programming problem than a usage problem, as discussed in Chapter 9. Instead, we’ll use the other technique you learned about in Chapter 9—returning a Result that indicates either success or an error.

Returning a Result 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. We’re also going to change the function name from new to build because many programmers expect new functions to never fail. When Config::build 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 the function we’re now calling Config::build and the body of the function needed to return a Result. Note that this won’t compile until we update main as well, which we’ll do in the next listing.

src/main.rs

impl Config {

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

if args.len() < 3 {

return Err("not enough arguments");

}

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

let file\_path = args[2].clone();

Ok(Config { query, file\_path })

}

}

Returning a Result from Config::build

Our build function returns a Result with a Config instance in the success case and an &'static str in the error case. Our error values will always be string literals that have the 'static lifetime.

We’ve made two changes in the body of the 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::build allows the main function to handle the Result value returned from the build function and exit the process more cleanly in the error case.

Calling Config::build 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::build, as shown in Listing 12-10. We’ll also take the responsibility of exiting the command line tool with a nonzero error code away from panic! and instead 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

1 use std::process;

fn main() {

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

2 let config = Config::build(&args).3unwrap\_or\_else(|4err| {

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

6 process::exit(1);

});

--snip--

Exiting with an error code if building a Config fails

In this listing, we’ve used a method we haven’t covered in detail yet: unwrap\_or\_else, which is defined on Result<T, E> by the standard library 2. 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 that 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 3. 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 4. The code in the closure can then use the err value when it runs.

We’ve added a new use line to bring process from the standard library into scope 1. The code in the closure that will be run in the error case is only two lines: we print the err value 5 and then call process::exit 6. 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 0.48s

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 isn’t 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.

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.

src/main.rs

fn main() {

--snip--

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

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

run(config);

}

fn run(config: Config) {

let contents = fs::read\_to\_string(config.file\_path)

.expect("Should have been able to read the file");

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

}

--snip--

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::build 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 into main 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

1 use std::error::Error;

--snip--

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

let contents = fs::read\_to\_string(config.file\_path)3?;

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

4 Ok(())

}

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<dyn Error>> 2. 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<dyn Error> (and we’ve brought std::error::Error into scope with a use statement at the top 1). We’ll cover trait objects in Chapter 18. For now, just know that Box<dyn 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 dyn keyword is short for dynamic.

Second, we’ve removed the call to expect in favor of the ? operator 3, as we talked about in Chapter 9. Rather than panic! on an error, ? will return the error value from the current function for the caller to handle.

Third, the run function now returns an Ok value in the success case 4. 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` that must be used

--> src/main.rs:19:5

|

19 | run(config);

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

|

= note: `#[warn(unused\_must\_use)]` on by default

= note: this `Result` may be an `Err` variant, which should be

handled

Rust tells us that our code ignored the Result value and the Result value 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 one we used with Config::build in Listing 12-10, but with a slight difference:

src/main.rs

fn main() {

--snip--

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

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

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 to 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::build returns the Config instance. Because run returns () in the success case, we only care about detecting an error, so we don’t need unwrap\_or\_else to return the unwrapped value, which 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. That way, we can test the code and have a src/main.rs file with fewer responsibilities.

Let’s move all the code that isn’t in 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::build 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). Note that this won’t compile until we modify src/main.rs in Listing 12-14.

src/lib.rs

use std::error::Error;

use std::fs;

pub struct Config {

pub query: String,

pub file\_path: String,

}

impl Config {

pub fn build(

args: &[String],

) -> Result<Config, &'static str> {

--snip--

}

}

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

--snip--

}

Moving Config and run into src/lib.rs

We’ve made liberal use of the pub keyword: on Config, on its fields and its build method, and on the run function. We now have a library crate that has a public API 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, as shown in Listing 12-14.

src/main.rs

use std::env;

use std::process;

use minigrep::Config;

fn main() {

--snip--

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

--snip--

}

}

Using the minigrep library crate in src/main.rs

We add a use minigrep::Config line to bring the Config type from the library crate into the binary crate’s scope, and we 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 the 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.

In this section, we’ll add the searching logic to the minigrep program using the test-driven development (TDD) process with the following steps:

1. Write a test that fails and run it to make sure it fails for the reason you expect.
2. Write or modify just enough code to make the new test pass.
3. Refactor the code you just added or changed and make sure the tests continue to pass.
4. Repeat from step 1!

Though it’s just one of many ways to write software, TDD can help drive code design. 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 anymore, let’s remove the println! statements from src/lib.rs and src/main.rs that we used to check the program’s behavior. Then, in src/lib.rs, we’ll add a tests 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, and it will return only the lines from the text that contain the query. Listing 12-15 shows this test, which won’t compile yet.

src/lib.rs

#[cfg(test)]

mod tests {

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)

);

}

}

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" (note that the backslash after the opening double quote tells Rust not to put a newline character at the beginning of the contents of this string literal). We assert that the value returned from the search function contains only the line we expect.

We aren’t yet able to run this test and watch it fail because the test doesn’t even compile: the search function doesn’t exist yet! In accordance with TDD principles, 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![]

}

Defining just enough of the search function so our test will compile

Notice that we need to define an explicit lifetime 'a in the signature of search and use that lifetime 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).

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.

If we forget the lifetime annotations and try to compile this function, we’ll get this error:

error[E0106]: missing lifetime specifier

--> src/lib.rs:31:10

|

29 | query: &str,

| ----

30 | contents: &str,

| ----

31 | ) -> Vec<&str> {

| ^ expected named 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`

help: consider introducing a named lifetime parameter

|

28 ~ pub fn search<'a>(

29 ~ query: &'a str,

30 ~ contents: &'a str,

31 ~ ) -> Vec<&'a str> {

|

Rust can’t possibly know which of the two arguments we need, so we need to tell it explicitly. 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, but this practice will get easier over time. You might want to compare this example with the examples in “Validating References with Lifetimes” on page XX.

Now let’s run the test:

$ cargo test

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

Finished test [unoptimized + debuginfo] target(s) in 0.97s

Running unittests src/lib.rs (target/debug/deps/minigrep-9cd200e5fac0fc94)

running 1 test

test tests::one\_result ... FAILED

failures:

---- tests::one\_result stdout ----

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

left: `["safe, fast, productive."]`,

right: `[]`', src/lib.rs:47:9

note: run with `RUST\_BACKTRACE=1` environment variable to display a backtrace

failures:

tests::one\_result

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

finished in 0.00s

error: test failed, to rerun pass '--lib'

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:

1. Iterate through each line of the contents.
2. Check whether the line contains our query string.
3. If it does, add it to the list of values we’re returning.
4. If it doesn’t, do nothing.
5. 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. Note that this won’t compile yet.

src/lib.rs

pub fn search<'a>(

query: &str,

contents: &'a str,

) -> Vec<&'a str> {

for line in contents.lines() {

// do something with line

}

}

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 3-5, 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 check whether the current line contains our query string. 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. Note that this still won’t compile yet.

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

}

}

}

Adding functionality to see whether the line contains the string in query

At the moment, we’re building up functionality. To get the code to compile, we need to return a value from the body as we indicated we would in the function signature.

Storing Matching Lines

To finish this function, we need a way to store the matching lines that we want to return. 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

}

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

--snip--

running 1 test

test tests::one\_result ... ok

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

filtered out; finished in 0.00s

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 tests passing 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.

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<dyn Error>> {

let contents = fs::read\_to\_string(config.file\_path)?;

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 0.38s

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 body:

$ cargo run -- body poem.txt

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

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

I'm nobody! Who are you?

Are you nobody, too?

How dreary to be somebody!

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 0.0s

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, 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.

Working with Environment Variables

We’ll improve minigrep 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 by instead making it an environment variable, we allow 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 first add a new search\_case\_insensitive function that will be called when the environment variable has a value. 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 search\_case\_insensitive 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 tests {

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)

);

}

}

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 match the line "Trust me." even though both have different casing from 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 that 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

pub fn search\_case\_insensitive<'a>(

query: &str,

contents: &'a str,

) -> Vec<&'a str> {

1 let query = query.to\_lowercase();

let mut results = Vec::new();

for line in contents.lines() {

if 2 line.to\_lowercase().contains(3&query) {

results.push(line);

}

}

results

}

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 new variable with the same name, shadowing the original query 1. Calling to\_lowercase on the query is necessary so that no matter whether the user’s query is "rust", "RUST", "Rust", or "rUsT", we’ll treat the query as if it were "rust" and be insensitive to the case. While to\_lowercase will handle basic Unicode, it won’t be 100 percent accurate. If we were writing a real application, we’d want to do a bit more work here, but this section is about environment variables, not Unicode, so we’ll leave it at that here.

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 3 because the signature of contains is defined to take a string slice.

Next, we add a call to to\_lowercase on each line to lowercase all characters 2. 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 tests::case\_insensitive ... ok

test tests::case\_sensitive ... ok

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

filtered out; finished in 0.00s

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 the Config struct to switch between case-sensitive and case-insensitive search. Adding this field will cause compiler errors because we aren’t initializing this field anywhere yet:

src/lib.rs

pub struct Config {

pub query: String,

pub file\_path: String,

pub ignore\_case: bool,

}

We added the ignore\_case field that holds a Boolean. Next, we need the run function to check the ignore\_case 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. This still won’t compile yet.

src/lib.rs

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

let contents = fs::read\_to\_string(config.file\_path)?;

let results = if config.ignore\_case {

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

} else {

search(&config.query, &contents)

};

for line in results {

println!("{line}");

}

Ok(())

}

Calling either search or search\_case\_insensitive based on the value in config.ignore\_case

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 bring that module into scope at the top of src/lib.rs. Then we’ll use the var function from the env module to check to see if any value has been set for an environment variable named IGNORE\_CASE, as shown in Listing 12-23.

src/lib.rs

use std::env;

--snip--

impl Config {

pub fn build(

args: &[String]

) -> Result<Config, &'static str> {

if args.len() < 3 {

return Err("not enough arguments");

}

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

let file\_path = args[2].clone();

let ignore\_case = env::var("IGNORE\_CASE").is\_ok();

Ok(Config {

query,

file\_path,

ignore\_case,

})

}

}

Checking for any value in an environment variable named IGNORE\_CASE

Here, we create a new variable, ignore\_case. To set its value, we call the env::var function and pass it the name of the IGNORE\_CASE environment variable. The env::var function returns a Result that will be the successful Ok variant that contains the value of the environment variable if the environment variable is set to any value. It will return the Err variant if the environment variable is not set.

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

We pass the value in the ignore\_case variable to the Config instance so the run function can read that value and decide whether to call search\_case\_insensitive or search, 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

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

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

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 IGNORE\_CASE set to 1 but with the same query to:

$ IGNORE\_CASE=1 cargo run -- to poem.txt

If you’re using PowerShell, you will need to set the environment variable and run the program as separate commands:

PS> $Env:IGNORE\_CASE=1; cargo run -- to poem.txt

This will make IGNORE\_CASE persist for the remainder of your shell session. It can be unset with the Remove-Item cmdlet:

PS> Remove-Item Env:IGNORE\_CASE

We should get lines that contain to that might have uppercase letters:

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 sensitivity 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 ignore case.

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! macro. In most terminals, there are two kinds of output: standard output (stdout) for general information and standard error (stderr) for error messages. This distinction enables users to choose to direct the successful output of a program to a file but still print error messages to the screen.

The println! macro 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

First let’s observe how the content printed by minigrep is currently being written to standard output, including any error messages we want to write to standard error instead. We’ll do that by redirecting the standard output stream to a file while 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!

To demonstrate this behavior, we’ll run the program with > and the file path, 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 to 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 so only data from a successful run ends up in the file. 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, 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::build(&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);

}

}

Writing error messages to standard error instead of standard output using eprintln!

Let’s now 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

This chapter recapped some of the major concepts you’ve learned so far and covered how to perform common I/O operations in Rust. By using command line arguments, files, environment variables, and the eprintln! macro for printing errors, you’re now prepared to write command line applications. Combined with 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 Rust features that were influenced by functional languages: closures and iterators.