Table of Contents

Command line apps in Rust

Rust is a statically compiled, fast language with great tooling and a rapidly growing ecosystem. That makes it a great fit for writing command line applications: They should be small, portable, and quick to run. Command line applications are also a great way to get started with learning Rust; or to introduce Rust to your team!

Writing a program with a simple command line interface (CLI) is a great exercise for a beginner who is new to the language and wants to get a feel for it. There are many aspects to this topic, though, that often only reveal themselves later on.

This book is structured like this: We start with a quick tutorial, after which you'll end up with a working CLI tool. You'll be exposed to a few of the core concepts of Rust as well as the main aspects of CLI applications. What follows are chapters that go into more detail on some of these aspects.

One last thing before we dive right into CLI applications: If you find an error in this book or want to help us write more content for it, you can find its source in the CLI book repository. We'd love to hear your feedback! Thank you!

Learning Rust by Writing a Command Line App in 15 Minutes

This tutorial will guide you through writing a CLI (command line interface) application in Rust. It will take you roughly fifteen minutes to get to a point where you have a running program (around chapter 1.3). After that, we'll continue to tweak our program until we reach a point where we can ship our little tool.

You'll learn all the essentials about how to get going, and where to find more information. Feel free to skip parts you don't need to know right now or jump in at any point.

> **Prerequisites:** This tutorial does not replace a general introduction to programming, and expects you to be familiar with a few common concepts. You should be comfortable with using a command line/terminal. If you already know a few other languages, this can be a good first contact with Rust.
>
> **Getting help:** If you at any point feel overwhelmed or confused with the features used, have a look at the extensive official documentation that comes with Rust, first and foremost the book, The Rust Programming Language. It comes with most Rust installations (`rustup doc`), and is available online on doc.rust-lang.org.
>
> You are also very welcome to ask questions – the Rust community is known to be friendly and helpful. Have a look at the community page to see a list of places where people discuss Rust.

What kind of project do you want to write? How about we start with something simple: Let's write a small `grep` clone. That is a tool that we can give a string and a path and it'll print only the lines that contain the given string. Let's call it `grrs` (pronounced "grass").

In the end, we want to be able to run our tool like this:

```
$ cat test.txt
foo: 10
bar: 20
baz: 30
$ grrs foo test.txt
foo: 10
$ grrs --help
[some help text explaining the available options]
```

> **Note:** This book is written for Rust 2018. The code examples can also be used on Rust 2015, but you might need to tweak them a bit; add `extern crate foo;` invocations, for example.
>
> Make sure you run Rust 1.31.0 (or later) and that you have `edition = "2018"` set in the `[package]` section of your `Cargo.toml` file.

Project setup

If you haven't already, install Rust on your computer (it should only take a few minutes). After that, open a terminal and navigate to the directory you want to put your application code into.

Start by running `cargo new grrs` in the directory you store your programming projects in. If you look at the newly created `grrs` directory, you'll find a typical setup for a Rust project:

- A `Cargo.toml` file that contains metadata for our project, incl. a list of dependencies/external libraries we use.
- A `src/main.rs` file that is the entry point for our (main) binary.

If you can execute `cargo run` in the `grrs` directory and get a "Hello World", you're all set up.

What it might look like

```
$ cargo new grrs
     Created binary (application) `grrs` package
$ cd grrs/
$ cargo run
   Compiling grrs v0.1.0 (/Users/pascal/code/grrs)
    Finished dev [unoptimized + debuginfo] target(s) in 0.70s
     Running `target/debug/grrs`
Hello, world!
```

Parsing command-line arguments

A typical invocation of our CLI tool will look like this:

```
$ grrs foobar test.txt
```

We expect our program to look at `test.txt` and print out the lines that contain `foobar`. But how do we get these two values?

The text after the name of the program is often called the "command-line arguments", or "command-line flags" (especially when they look like `--this`). Internally, the operating system usually represents them as a list of strings. Generally, they get separated by spaces.

There are many ways to think about these arguments and how to parse them into something easier to work with. You will also need to tell the users of your program which arguments they need to give and in which format they are expected.

Getting the arguments

The standard library contains the function `std::env::args()` that gives you an iterator of the given arguments. The first entry (at index `0`) will be the name used to invoke your program (e.g. `grrs`). The ones that follow are what the user wrote afterwards.

Getting the raw arguments this way is straightforward (in file `src/main.rs`):

```
fn main() {
    let pattern = std::env::args().nth(1).expect("no pattern given");
    let path = std::env::args().nth(2).expect("no path given");

    println!("pattern: {:?}, path: {:?}", pattern, path)
}
```

We can run it using `cargo run`, passing arguments by writing them after `--`:

```
$ cargo run -- some-pattern some-file
    Finished dev [unoptimized + debuginfo] target(s) in 0.11s
     Running `target/debug/grrs some-pattern some-file`
pattern: "some-pattern", path: "some-file"
```

CLI arguments as data types

Instead of thinking about them as a bunch of text, it often pays off to think of CLI arguments as a custom data type that represents the inputs to your program.

Looking at `grrs foobar test.txt`, there are two arguments: first, the `pattern` (the string to look for), and then, the `path` (the file to look in).

What more can we say about them? Well, for a start, both are required. We haven't talked about any default values, so we expect our users to always provide two values. Furthermore, we can say a bit about their types: The pattern is expected to be a string while the second argument is expected to be a path to a file.

In Rust, it is common to structure programs around the data they handle, so this way of looking at CLI arguments fits very well. Let's start with this (in file `src/main.rs`, before `fn main() {`):

```
struct Cli {
    pattern: String,
    path: std::path::PathBuf,
}
```

This defines a new structure (a `struct`) that has two fields to store data in: `pattern` and `path`.

> **Note:** `PathBuf` is like a `String` but for file system paths that work cross-platform.

Now, we still need to convert the actual arguments into this form. One option would be to manually parse the list of strings we get from the operating system and build the structure ourselves. It would look something like this:

```
fn main() {
    let pattern = std::env::args().nth(1).expect("no pattern given");
    let path = std::env::args().nth(2).expect("no path given");

    let args = Cli {
        pattern,
        path: std::path::PathBuf::from(path),
    };

    println!("pattern: {:?}, path: {:?}", args.pattern, args.path);
}
```

This works, but it's not very convenient. How would you deal with the requirement to support `--pattern="foo"` or `--pattern "foo"`? How would you implement `--help`?

Parsing CLI arguments with Clap

A more convenient way is to use one of the many available libraries. The most popular library for parsing command-line arguments is called `clap`. It has all the functionality you'd expect, including support for sub-commands, shell completions, and great help messages.

Let's first import `clap` by adding `clap = { version = "4.0", features = ["derive"] }` to the `[dependencies]` section of our `Cargo.toml` file.

Now, we can write `use clap::Parser;` in our code and add `#[derive(Parser)]` right above our `struct Cli`. Let's also write some documentation comments along the way.

It'll look like this (in file `src/main.rs`, before `fn main() {`):

```
use clap::Parser;

/// Search for a pattern in a file and display the lines that contain it.
#[derive(Parser)]
struct Cli {
    /// The pattern to look for
    pattern: String,
    /// The path to the file to read
    path: std::path::PathBuf,
}
```

> **Note:** There are a lot of custom attributes you can add to fields. For example, to say you want to use this field for the argument after `-o` or `--output`, you'd add `#[arg(short = 'o', long = "output")]`. For more information, see the clap documentation.

Right below the `Cli` struct our template contains its `main` function. When the program starts, it will call this function:

```
fn main() {
    let args = Cli::parse();

    println!("pattern: {:?}, path: {:?}", args.pattern, args.path)
}
```

This will try to parse the arguments into our `Cli` struct.

But what if that fails? That's the beauty of this approach: Clap knows which fields to expect and their expected format. It can automatically generate a nice `--help` message as well as give some great errors to suggest you pass `--output` when you wrote `--putput`.

> **Note:** The `parse` method is meant to be used in your `main` function. When it fails, it will print out an error or help message and immediately exit the program. Don't use it in other places!

Wrapping up

Your code should now look like:

```
use clap::Parser;

/// Search for a pattern in a file and display the lines that contain it.
#[derive(Parser)]
struct Cli {
    /// The pattern to look for
    pattern: String,
    /// The path to the file to read
    path: std::path::PathBuf,
}

fn main() {
    let args = Cli::parse();

    println!("pattern: {:?}, path: {:?}", args.pattern, args.path)
}
```

Running it without any arguments:

```
$ cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 10.16s
     Running `target/debug/grrs`
error: The following required arguments were not provided:
    <pattern>
    <path>

USAGE:
    grrs <pattern> <path>

For more information try --help
```

Running it passing arguments:

```
$ cargo run -- some-pattern some-file
    Finished dev [unoptimized + debuginfo] target(s) in 0.11s
     Running `target/debug/grrs some-pattern some-file`
pattern: "some-pattern", path: "some-file"
```

The output demonstrates that our program successfully parsed the arguments into the `Cli` struct.

First implementation of *grrs*

After the last chapter on command line arguments, we have our input data, and we can start to write our actual tool. Our `main` function only contains this line right now:

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

We can drop the `println` statement that we put there temporarily to demonstrate that our program works as expected.

Let's start by opening the file we got.

```
    let content = std::fs::read_to_string(&args.path).expect("could not read file");
```

> **Note:** See that `.expect` method here? This is a shortcut function that will make the program exit immediately when the value (in this case, the input file) could not be read. It's not very pretty, and in the next chapter on Nicer error reporting, we will look at how to improve this.

Now, let's iterate over the lines and print each one that contains our pattern:

```
    for line in content.lines() {
        if line.contains(&args.pattern) {
            println!("{}", line);
        }
    }
```

Wrapping up

Your code should now look like:

```rust
use clap::Parser;

/// Search for a pattern in a file and display the lines that contain it.
#[derive(Parser)]
struct Cli {
    /// The pattern to look for
    pattern: String,
    /// The path to the file to read
    path: std::path::PathBuf,
}

fn main() {
    let args = Cli::parse();
    let content = std::fs::read_to_string(&args.path).expect("could not read file");

    for line in content.lines() {
        if line.contains(&args.pattern) {
            println!("{}", line);
        }
    }
}
```

Give it a try: `cargo run -- main src/main.rs` should work now!

> **Exercise for the reader:** This is not the best implementation as it will read the whole file into memory, no matter how large the file may be. Find a way to optimize it! (One idea might be to use a `BufReader` instead of `read_to_string()`.)

Nicer error reporting

We all can do nothing but accept the fact that errors will occur. In contrast to many other languages, it's very hard not to notice and deal with this reality when using Rust because it doesn't have exceptions. All possible error states are often encoded in the return types of functions.

Results

A function like `read_to_string` doesn't return a string. Instead, it returns a `Result` that contains either a `String` or an error of some type. In this case, `std::io::Error`.

How do you know which it is? Since `Result` is an `enum`, you can use `match` to check which variant it is:

```
let result = std::fs::read_to_string("test.txt");
match result {
    Ok(content) => { println!("File content: {}", content); }
    Err(error) => { println!("Oh noes: {}", error); }
}
```

> **Note:** Not sure what enums are or how they work in Rust? Check out this chapter of the Rust book to get up to speed.

Unwrapping

Now, we were able to access the content of the file, but we can't really do anything with it after the `match` block. For this, we'll need to deal with the error case. While it's a challenge that all arms of a `match` block need to return something of the same type, there's a neat trick to get around that:

```
let result = std::fs::read_to_string("test.txt");
let content = match result {
    Ok(content) => { content },
    Err(error) => { panic!("Can't deal with {}, just exit here", error); }
};
println!("file content: {}", content);
```

We can use the String in `content` after the match block, but if `result` were an error, the String wouldn't exist. That's fine because the program would exit before it ever reached a point where we use `content`.

This may seem drastic, but it's very convenient. If your program needs to read that file and can't do anything if the file doesn't exist, exiting is a valid strategy. There's even a shortcut method on `Result` called `unwrap`:

```
let content = std::fs::read_to_string("test.txt").unwrap();
```

No need to panic

Of course, aborting the program is not the only way to deal with errors. Instead of using `panic!`, we can just use `return`:

```
let result = std::fs::read_to_string("test.txt");
let content = match result {
    Ok(content) => { content },
    Err(error) => { return Err(error.into()); }
};
```

However, this changes the return type in our function. There was something hidden in our examples all this time: The function signature this code lives in. And in this last example with `return`, it becomes important. Here's the *full* example:

```
fn main() -> Result<(), Box<dyn std::error::Error>> {
    let result = std::fs::read_to_string("test.txt");
    let content = match result {
        Ok(content) => { content },
        Err(error) => { return Err(error.into()); }
    };
    println!("file content: {}", content);
    Ok(())
}
```

Our return type is a `Result`! This is why we can write `return Err(error);` in the second match arm. See how there is an `Ok(())` at the bottom? It's the default return value of the function and means: "Result is okay, and has no content".

> **Note:** Why is this not written as `return Ok(());`? It easily could be – this is totally valid as well. The last expression of any block in Rust is its return value, and it is customary to omit a needless `return`.

Question Mark

Just like calling `.unwrap()` is a shortcut for the `match` with `panic!` in the error arm, we have another shortcut for the `match` that `return`s in the error arm: `?`.

That's right, a question mark. You can append this operator to a value of type `Result`, and Rust will internally expand this to something very similar to the `match` we just wrote.

Give it a try:

```
fn main() -> Result<(), Box<dyn std::error::Error>> {
    let content = std::fs::read_to_string("test.txt")?;
    println!("file content: {}", content);
    Ok(())
}
```

Very concise!

> **Note:** There are a few more things happening here that are not required to understand to work with this. For example, the error type in our `main` function is `Box<dyn std::error::Error>`, but we've seen above that `read_to_string` returns a `std::io::Error`. This works because `?` expands to code that *converts* error types.
>
> `Box<dyn std::error::Error>` is also an interesting type. It's a `Box` that can contain *any* type that implements the standard `Error` trait. This means that all errors can be put into this box, and we can use `?` on all of the usual functions that return a `Result`.

Providing Context

The errors you get when using `?` in your `main` function are okay, but they are not great. For example, when you run `std::fs::read_to_string("test.txt")?` and the file `test.txt` doesn't exist, you get this output:

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

In cases where your code doesn't actually contain the file name, it would be hard to tell which file was `NotFound`. There are multiple ways to deal with this.

For one, we can create our own error type and use that to build a custom error message:

```
#[derive(Debug)]
struct CustomError(String);

fn main() -> Result<(), CustomError> {
    let path = "test.txt";
    let content = std::fs::read_to_string(path)
        .map_err(|err| CustomError(format!("Error reading `{}`: {}", path, err)))?;
    println!("file content: {}", content);
    Ok(())
}
```

Running this, we'll get our custom error message:

```
Error: CustomError("Error reading `test.txt`: No such file or directory (os error 2)")
```

Not very pretty, but we can adapt the debug output for our type later on.

This pattern is very common. It has one problem though: We don't store the original error, only its string representation. The popular `anyhow` library has a neat solution for that: Its `Context` trait can be used to add a description similar to our `CustomError` type. Additionally, it keeps the original error, so we get a "chain" of error messages pointing to the root cause.

Let's first import the `anyhow` crate by adding `anyhow = "1.0"` to the `[dependencies]` section of our `Cargo.toml` file.

The full example will look like this:

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

fn main() -> Result<()> {
    let path = "test.txt";
    let content = std::fs::read_to_string(path)
        .with_context(|| format!("could not read file `{}`", path))?;
    println!("file content: {}", content);
    Ok(())
}
```

This will print an error:

```
Error: could not read file `test.txt`

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

Wrapping up

Your code should now look like:

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

/// Search for a pattern in a file and display the lines that contain it.
#[derive(Parser)]
struct Cli {
    /// The pattern to look for
    pattern: String,
    /// The path to the file to read
    path: std::path::PathBuf,
}

fn main() -> Result<()> {
    let args = Cli::parse();

    let content = std::fs::read_to_string(&args.path)
        .with_context(|| format!("could not read file `{}`", args.path.display()))?;

    for line in content.lines() {
        if line.contains(&args.pattern) {
            println!("{}", line);
        }
    }

    Ok(())
}
```

Output

Printing “Hello World”

```
println!("Hello World");
```

Well, that was easy. Great! Onto the next topic.

Using `println!`

You can pretty much print all the things you like with the `println!` macro. This macro has some pretty amazing capabilities, but also a special syntax. It expects a string literal that contains placeholders as the first parameter. The string will be filled in by the values of the parameters that follow as further arguments.

For example:

```
let x = 42;
println!("My lucky number is {}.", x);
```

will print:

```
My lucky number is 42.
```

The curly braces (`{}`) in the string above is one of these placeholders. This is the default placeholder type that tries to print the given value in a human readable way. For numbers and strings, this works very well, but not all types can do that. This is why there is also a “debug representation” that you can get by filling the braces of the placeholder like this: `{:?}` .

For example:

```
let xs = vec![1, 2, 3];
println!("The list is: {:?}", xs);
```

will print:

```
The list is: [1, 2, 3]
```

If you want your own data types to be printable for debugging and logging, you can typically add a `#[derive(Debug)]` above their definition.

> **Note:** “User-friendly” printing is done using the `Display` trait and debug output (human-readable but targeted at developers) uses the `Debug` trait. You can find more information about the syntax you can

use in `println!` in the documentation for the `std::fmt` module.

Printing errors

Printing errors should be done via `stderr` to make it easier for users and other tools to pipe their outputs to files or more tools.

> **Note:** On most operating systems, a program can write to two output streams: `stdout` and `stderr`. `stdout` is for the program's actual output while `stderr` allows errors and other messages to be kept separate from `stdout`. That way, output can be stored to a file or piped to another program while errors are shown to the user.

In Rust, this is achieved with `println!` and `eprintln!`, the former printing to `stdout` and the latter to `stderr`.

```
println!("This is information");
eprintln!("This is an error! :(");
```

> **Beware**: Printing escape codes can be dangerous and put the user's terminal into a weird state. Always be careful when manually printing them!
>
> Ideally, you should be using a crate like `ansi_term` when dealing with raw escape codes to make your (and your user's) life easier.

A note on printing performance

Printing to the terminal is surprisingly slow! If you call things like `println!` in a loop, it can easily become a bottleneck in an otherwise fast program. To speed this up, there are two things you can do.

First, you might want to reduce the number of writes that actually "flush" to the terminal. `println!` tells the system to flush to the terminal *every* time because it is common to print each new line. If you don't need that, you can wrap your `stdout` handle in a `BufWriter`, which buffers up to 8 kB by default. You can still call `.flush()` on this `BufWriter` when you want to print immediately.

```
use std::io::{self, Write};

let stdout = io::stdout(); // get the global stdout entity
let mut handle = io::BufWriter::new(stdout); // optional: wrap that handle in a buffer
writeln!(handle, "foo: {}", 42); // add `?` if you care about errors here
```

Second, it helps to acquire a lock on `stdout` (or `stderr`) and use `writeln!` to print to it directly. This prevents the system from locking and unlocking `stdout` over and over again.

```
use std::io::{self, Write};

let stdout = io::stdout(); // get the global stdout entity
let mut handle = stdout.lock(); // acquire a lock on it
writeln!(handle, "foo: {}", 42); // add `?` if you care about errors here
```

You can also combine the two approaches.

Showing a progress bar

Some CLI applications run less than a second while others take minutes or hours. If you are writing one of the latter types of programs, you might want to show the user that something is happening. For this, you should try to print useful status updates, ideally in a form that can be easily consumed.

Using the indicatif crate, you can add progress bars and little spinners to your program. Here's a quick example:

```
fn main() {
    let pb = indicatif::ProgressBar::new(100);
    for i in 0..100 {
        do_hard_work();
        pb.println(format!("[+] finished #{}", i));
        pb.inc(1);
    }
    pb.finish_with_message("done");
}
```

See the documentation and examples for more information.

Logging

To make it easier to understand what is happening in our program, we might want to add some log statements. This is usually easy while writing your application, and it will become super helpful when running this program again in half a year. In some ways, logging is the same as using `println!` except that you can specify the importance of a message. The levels you can usually use are *error*, *warn*, *info*, *debug*, and *trace* (*error* has the highest priority, *trace* the lowest).

To add simple logging to your application, you'll need two things: The log crate (this contains macros named after the log level) and an *adapter* that actually writes the log output somewhere useful. Having the ability to use log adapters is very flexible: You can, for example, use them to write logs not only to the terminal but also to syslog or to a central log server.

Since we are only concerned with writing a CLI application, an easy adapter to use is env_logger. It's called "env" logger because you can use an environment variable to specify which parts of your application you want to log and at which level you want to log them. It will prefix your log messages with a timestamp and the module where the log messages come from. Since libraries can also use `log`, you easily configure their log output, too.

Here's a quick example:

```
use log::{info, warn};

fn main() {
    env_logger::init();
    info!("starting up");
    warn!("oops, nothing implemented!");
}
```

Assuming you have this file as `src/bin/output-log.rs`, on Linux and macOS, you can run it like this:

```
$ env RUST_LOG=info cargo run --bin output-log
    Finished dev [unoptimized + debuginfo] target(s) in 0.17s
     Running `target/debug/output-log`
[2018-11-30T20:25:52Z INFO  output_log] starting up
[2018-11-30T20:25:52Z WARN  output_log] oops, nothing implemented!
```

In Windows PowerShell, you can run it like this:

```
$ $env:RUST_LOG="info"
$ cargo run --bin output-log
    Finished dev [unoptimized + debuginfo] target(s) in 0.17s
     Running `target/debug/output-log.exe`
[2018-11-30T20:25:52Z INFO  output_log] starting up
[2018-11-30T20:25:52Z WARN  output_log] oops, nothing implemented!
```

In Windows CMD, you can run it like this:

```
$ set RUST_LOG=info
$ cargo run --bin output-log
    Finished dev [unoptimized + debuginfo] target(s) in 0.17s
     Running `target/debug/output-log.exe`
[2018-11-30T20:25:52Z INFO  output_log] starting up
[2018-11-30T20:25:52Z WARN  output_log] oops, nothing implemented!
```

`RUST_LOG` is the name of the environment variable you can use to set your log settings. `env_logger` also contains a builder so you can programmatically adjust these settings like showing *info* level messages by default.

There are a lot of alternative logging adapters out there as well as alternatives and extensions to `log`. If you know your application will have a lot to log, make sure to review them and make your users' lives easier.

> **Tip:** Experience has shown that even mildly useful CLI programs can end up being used for years to come, especially if they were meant as a temporary solution. If your application doesn't work and someone (e.g., you, in the future) needs to figure out why, being able to pass `--verbose` to get additional log output can make the difference between minutes and hours of debugging. The clap-verbosity-flag crate contains a quick way to add a `--verbose` to a project using `clap`.

Testing

Over decades of software development, people have discovered one truth: Untested software rarely works. Many people would go as far as saying that most tested software doesn't work either. But we are all optimists here, right? To ensure that your program does what you expect it to do, it is wise to test it.

A good starting point is to write a `README` file that describes what your program should do, and when you feel ready to make a new release, go through the `README` and ensure that the behavior is still as expected. You can make this a more rigorous exercise by also writing down how your program should react to erroneous inputs.

Here's another fancy idea: Write that `README` before you write the code.

> **Note:** Have a look at test-driven development (TDD) if you haven't heard of it.

Automated testing

Now, this is all fine and dandy, but doing all of this manually? That can take a lot of time. At the same time, many people have come to enjoy telling computers to do things for them. Let's talk about how to automate these tests.

Rust has a built-in test framework, so let's start by writing our first test:

```
#[test]
fn check_answer_validity() {
    assert_eq!(answer(), 42);
}
```

You can put this snippet of code in pretty much any source file in your package and `cargo test` will find and run it. The key here is the `#[test]` attribute. It allows the build system to discover such functions and run them as tests, verifying that they don't panic.

> **Exercise for the reader:** Make this test work.
>
> You should end up with output like the following:
>
> ```
> running 1 test
> test check_answer_validity ... ok
>
> test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
> ```

Now that we've seen *how* we can write tests, we still need to figure out *what* to test. As you've seen, it takes little code to write assertions for functions, but a CLI application is often more than one function! Worse, it often deals with user input, reads files, and writes output.

Making your code testable

There are two complementary approaches to testing functionality. One is to test the small units that you use to build your complete application. These are called "unit tests". Another is to test the final application from the outside, called black box tests or integration tests. Let's begin with the first one.

To figure out what we should test, let's see what our program features are. `grrs` is supposed to print out the lines that match a given pattern, so let's write unit tests for *exactly this*. We want to ensure that our most important piece of logic works, and we want to do it in a way that is not dependent on any of the setup code we have around it like the CLI arguments.

Going back to our first implementation of `grrs`, we added this block of code to the `main` function:

```
// ...
for line in content.lines() {
    if line.contains(&args.pattern) {
        println!("{}", line);
    }
}
```

Sadly, this is not very easy to test. First of all, it's in the main function, so we can't easily call it. This is fixed by moving this piece of code into a function:

```
fn find_matches(content: &str, pattern: &str) {
    for line in content.lines() {
        if line.contains(pattern) {
            println!("{}", line);
        }
    }
}
```

Now, we can call this function in our test and see what its output is:

```
#[test]
fn find_a_match() {
    find_matches("lorem ipsum\ndolor sit amet", "lorem");
    assert_eq!( // uhhhh
```

Or... can we? Right now, `find_matches` prints directly to `stdout`, i.e., the terminal. We can't easily capture this in a test! This is a problem that often comes up when writing tests after the implementation: We have written a function that is firmly integrated in the context it is used in.

> **Note:** This is totally fine when writing small CLI applications. There's no need to make everything testable! It is important to think about which parts of your code you might want to write unit tests for. While we'll see that it's straightforward to change this function to be testable, this is not always the case.

Alright, how can we make this testable? We'll need to capture the output somehow. Rust's standard library has some neat abstractions for dealing with I/O (input/output), and we'll make use of one called `std::io::Write`. This is a trait that abstracts over things we can write to, which includes strings and `stdout`.

If this is the first time you've heard "trait" in the context of Rust, you are in for a treat. Traits are one of the most powerful features of Rust. You can think of them like interfaces in Java or type classes in Haskell, whatever you are more familiar with. They allow you to abstract over behavior that can be shared by different types. Code that uses traits can express ideas in very generic and flexible ways. This means it can also get difficult to read. Don't let that intimidate you. Even people who have used Rust for years don't always get what generic code does immediately. In that case, it helps to think of concrete uses. In our case, the behavior that we abstract over is "write to it". Examples for the types that implement (`impl`) it include the terminal's standard output, files, a buffer in memory, or TCP network connections. Scroll down in the documentation for `std::io::Write` to see a list of "Implementors".

With that knowledge, let's change our function to accept a third parameter. It can be any type that implements `Write`. This way, we can supply a simple string in our tests and make assertions on it. Here is how we can write this version of `find_matches`:

```
fn find_matches(content: &str, pattern: &str, mut writer: impl std::io::Write) {
    for line in content.lines() {
        if line.contains(pattern) {
            writeln!(writer, "{}", line);
        }
    }
}
```

The new parameter is `mut writer`, i.e., a mutable thing we call "writer". Its type is `impl std::io::Write`, which you can read as a placeholder for any type that implements the `Write` trait. Note how we replaced the `println!(…)` we used earlier with `writeln!(writer, …)`. `println!` works the same as `writeln!`, but it always uses standard output.

Now, we can test for the output:

```
#[test]
fn find_a_match() {
    let mut result = Vec::new();
    find_matches("lorem ipsum\ndolor sit amet", "lorem", &mut result);
    assert_eq!(result, b"lorem ipsum\n");
}
```

To use this in our application code, we have to change the call to `find_matches` in `main` by adding `&mut std::io::stdout()` as the third parameter. Here's an example of a main function that builds on what we've seen in the previous chapters and uses our extracted `find_matches` function:

```
fn main() -> Result<()> {
    let args = Cli::parse();
    let content = std::fs::read_to_string(&args.path)
        .with_context(|| format!("could not read file `{}`", args.path.display()))?;

    find_matches(&content, &args.pattern, &mut std::io::stdout());

    Ok(())
}
```

> **Note:** Since `stdout` expects bytes (not strings), we use `std::io::Write` instead of `std::fmt::Write`. As a result, we give an empty vector as `writer` in our tests (its type will be

> inferred to `Vec<u8>`), and in the `assert_eq!`, we use `b"foo"`. The `b` prefix makes this a *byte string literal*, so its type is going to be `&[u8]` instead of `&str`.

> **Note:** We could also make this function return a `String`, but that would change its behavior. Instead of writing to the terminal directly, it would then collect everything into a string, and dump all the results in one go at the end.

> **Exercise for the reader:** `writeln!` returns an `io::Result` because writing can fail (for example, when the buffer is full and cannot be expanded). Add error handling to `find_matches`.

We've just seen how to make this piece of code testable. We have:

1. Identified one of the core pieces of our application.
2. Put it into its own function.
3. Made it more flexible.

Even though the goal was to make it testable, the result we ended up with is actually a very idiomatic and reusable piece of Rust code. That's awesome!

Splitting your code into library and binary targets

We can do one more thing here. So far, we've put everything we wrote into the `src/main.rs` file. This means our current project produces a single binary, but we can also make our code available as a library like this:

1. Put the `find_matches` function into a new `src/lib.rs`.
2. Add a `pub` in front of the `fn` to make it something that users of our library can access (i.e. `pub fn find_matches`).
3. Remove `find_matches` from `src/main.rs`.
4. In `fn main`, prepend the call to `find_matches` with `grrs::` so that it's now `grrs::find_matches(…)`. This means it uses the function from the library we just wrote!

The way Rust deals with projects is quite flexible, and it's a good idea to think about what to put into the library part of your crate early on. You can, for example, think about writing a library for your application-specific logic first and then use it in your CLI just like any other library. Or, if your project has multiple binaries, you can put the common functionality into the library part of that crate.

> **Note:** Speaking of putting everything into a `src/main.rs`, if we continue to do that, it'll become difficult to read. The module system can help you structure and organize your code.

Testing CLI applications by running them

Thus far, we've gone out of our way to test the *business logic* of our application, which turned out to be the `find_matches` function. This is very valuable and is a great first step towards a well-tested code base. Usually, these kinds of tests are called "unit tests".

There is a lot of code we aren't testing: Everything that we wrote to deal with the outside world! Imagine you wrote the main function but accidentally left in a hard-coded string instead of using the argument of the user-supplied path. We should write tests for that, too! This level of testing is often called integration testing or system testing.

At its core, we are still writing functions and annotating them with `#[test]`. It's just a matter of what we do inside these functions. For example, we'll want to use the main binary of our project and run it like a regular program. We will put these tests into a new file in a new directory: `tests/cli.rs`.

> **Note:** By convention, `cargo` will look for integration tests in the `tests/` directory. Similarly, it will look for benchmarks in `benches/` and examples in `examples/`. These conventions also extend to your main source code: libraries have a `src/lib.rs` file, the main binary is `src/main.rs`, and if there are multiple binaries, cargo expects them to be in `src/bin/<name>.rs`. Following these conventions will make your code base more discoverable by people used to reading Rust code.

`grrs` is a small tool that searches for a string in a file. We have already tested that we can find a match. Let's think about what other functionality we can test.

Here is what I came up with:

- What happens when the file doesn't exist?
- What is the output when there is no match?
- Does our program exit with an error when we forget one (or both) arguments?

These are all valid test cases. Additionally, we should include one test case for the happy path: we found at least one match and we print it.

To make these kinds of tests easier, we're going to use the `assert_cmd` crate. It has a bunch of neat helpers that allow us to run our main binary and see how it behaves. We'll also add the `predicates` crate, which helps us write assertions that `assert_cmd` can test against and that have great error messages. We won't add those dependencies to the main list, but to a `dev dependencies` section in our `Cargo.toml`. They are only required when developing the crate, not when using it.

```
[dev-dependencies]
assert_cmd = "2.0.14"
predicates = "3.1.0"
```

This sounds like a lot of setup. Nevertheless, let's dive right in and create our `tests/cli.rs` file:

```
use assert_cmd::prelude::*; // Add methods on commands
use predicates::prelude::*; // Used for writing assertions
use std::process::Command; // Run programs

#[test]
fn file_doesnt_exist() -> Result<(), Box<dyn std::error::Error>> {
    let mut cmd = Command::cargo_bin("grrs")?;

    cmd.arg("foobar").arg("test/file/doesnt/exist");
    cmd.assert()
        .failure()
        .stderr(predicate::str::contains("could not read file"));

    Ok(())
}
```

You can run this test with `cargo test`, just like the tests we wrote above. It might take a little longer the first time as `Command::cargo_bin("grrs")` needs to compile your main binary.

Generating test files

The test we've just seen only checks that our program writes an error message when the input file doesn't exist. That's an important test to have, but maybe not the most important one. Let's test that we will actually print the matches we found in a file!

We'll need to have a file whose content we know so that we can know what our program *should* return and check this expectation in our code. One idea might be to add a file to the project with custom content and use that in our tests. Another would be to create temporary files in our tests. For this tutorial, we'll have a look at the latter approach. It is more flexible and will work for other cases; for example, when you are testing programs that change the files.

To create these temporary files, we'll be using the `assert_fs` crate. Let's add it to the `dev-dependencies` in our `Cargo.toml`:

```
assert_fs = "1.1.1"
```

Here is a new test case that creates a temp file (a "named" one so we can get its path), fills it with some text, and then runs our program to see if we get the correct output. You can write it below the other test case. When the variable `file` goes out of scope at the end of the function, the actual temporary file will automatically get deleted.

```
use assert_fs::prelude::*;

#[test]
fn find_content_in_file() -> Result<(), Box<dyn std::error::Error>> {
    let file = assert_fs::NamedTempFile::new("sample.txt")?;
    file.write_str("A test\nActual content\nMore content\nAnother test")?;

    let mut cmd = Command::cargo_bin("grrs")?;
    cmd.arg("test").arg(file.path());
    cmd.assert()
        .success()
        .stdout(predicate::str::contains("A test\nAnother test"));

    Ok(())
}
```

> **Exercise for the reader:** Add integration tests for passing an empty string as pattern. Adjust the program as needed.

What to test?

While it can certainly be fun to write integration tests, it will take some time to write them as well as to update them when your application's behavior changes. To make sure you use your time wisely, you should ask yourself what you should test.

In general, it's a good idea to write integration tests for all types of behavior that a user can observe. This means that you don't need to cover all edge cases. It usually suffices to have examples for the different types and rely on unit tests to cover the edge cases.

It is also a good idea not to focus your tests on things you can't actively control. It would be a bad idea to test the exact layout of `--help` since it is generated for you. Instead, you might just want to check that certain elements are present.

Depending on the nature of your program, you can also try to add more testing techniques. For example, if you have extracted parts of your program and find yourself writing a lot of example cases as unit tests while trying to come up with all the edge cases, you should look into `proptest`. If you have a program that consumes arbitrary files and parses them, try to write a fuzzer to find bugs in edge cases.

> **Note:** You can find the full, runnable source code used in this chapter in this book's repository.

Packaging and distributing a Rust tool

If you feel confident that your program is ready for other people to use, it is time to package and release it!

There are a few approaches, and we'll look at three of them from quickest to set up to most convenient for users.

Quickest: `cargo publish`

The easiest way to publish your app is with cargo. Do you remember how we added external dependencies to our project? Cargo downloaded them from its default crate registry: crates.io. With `cargo publish`, you can publish crates to crates.io, and this works for all crates, including those with binary targets.

Publishing a crate to crates.io can be done in a few steps. First, if you haven't already, create an account on crates.io, which is done by authorizing you on GitHub, so you'll need to have a GitHub account and be logged in there. Second, you log in using cargo on your local machine. For that, go to your crates.io account page, create a new token, and run `cargo login <your-new-token>`. You only need to do this once per computer. You can learn more about this in cargo's publishing guide.

Now that cargo and crates.io know you, you are ready to publish crates. Before you hastily go ahead and publish a new crate version, it's a good idea to open your `Cargo.toml` once more and make sure you added the necessary metadata. You can find all the possible fields you can set in the documentation for cargo's manifest format. Here's a quick overview of some common entries:

```
[package]
name = "grrs"
version = "0.1.0"
authors = ["Your Name <your@email.com>"]
license = "MIT OR Apache-2.0"
description = "A tool to search files"
readme = "README.md"
homepage = "https://github.com/you/grrs"
repository = "https://github.com/you/grrs"
keywords = ["cli", "search", "demo"]
categories = ["command-line-utilities"]
```

> **Note:** This example includes the mandatory license field with a common choice for Rust projects: The same license that is used for the compiler itself. It also refers to a `README.md` file. It should include a quick description of what your project is about and will be included not only on the crates.io page of your crate, but GitHub shows it by default on repository pages.

How to install a binary from crates.io

We've seen how to publish a crate to crates.io, and you might be wondering how to install it. In contrast to libraries, which cargo will download and compile for you when you run `cargo build` or a similar command, you'll need to tell it to explicitly install binaries.

This is done using `cargo install <crate-name>`. It will download the crate by default, compile all the binary targets it contains (in "release" mode, so it might take a while) and copy them into the `~/.cargo/bin/` directory. Make sure that your shell knows to look there for binaries!

It's also possible to install crates from git repositories, only install specific binaries of a crate, and specify an alternative directory to install them to. Have a look at `cargo install --help` for details.

When to use it

`cargo install` is a simple way to install a binary crate. It's very convenient for Rust developers to use but has some significant downsides: Since it will always compile your source from scratch, users of your tool will need to have Rust, cargo, and all other system dependencies that your project requires installed on their machine. Compiling large Rust codebases can take some time.

It's best to use this for distributing tools that are targeted at other Rust developers. For example, a lot of cargo subcommands like `cargo-tree` or `cargo-outdated` can be installed with it.

Distributing binaries

Rust is a language that compiles to native code and statically links all dependencies by default. When you run `cargo build` on your project that contains a binary called `grrs`, you'll end up with a binary file called `grrs`. Try it out! Using `cargo build`, it'll be `target/debug/grrs`, and when you run `cargo build --release`, it'll be `target/release/grrs`. Unless you use crates that explicitly need external libraries installed on the target system (like using the system's version of OpenSSL), this binary will only depend on common system libraries. That means, you take that one file, send it to people running the same operating system as you, and they'll be able to run it.

This is already very powerful! It works around two of the downsides we just saw for `cargo install`: There is no need to have Rust installed on the user's machine, and instead of it taking a minute to compile, they can instantly run the binary.

As we've seen, `cargo build` *already* builds binaries for us. The issue is that those are not guaranteed to work on all platforms. If you run `cargo build` on your Windows machine, you won't get a binary that works on a Mac by default. Is there a way to generate these binaries for all of the target platforms automatically?

Building binary releases on CI

If your tool is open sourced and hosted on GitHub, it's quite easy to set up a free CI (continuous integration) service like Travis CI. There are other services that offer this functionality, but Travis is very popular. This runs setup commands in a virtual machine each time you push changes to your repository. What those commands are, and the types of machines they run on, is configurable. For example, a good idea is to run `cargo test` on a machine with Rust and some common build tools installed. If this fails, you know there are issues in the most recent changes.

We can also use this to build binaries and upload them to GitHub! If we run `cargo build --release` and upload the binary somewhere, we should be all set, right? Not quite. We still need to make sure the binaries we build are compatible with as many systems as possible. For example, on Linux we can compile for the current system or the `x86_64-unknown-linux-musl` target and not depend on default system libraries. On macOS, we can set `MACOSX_DEPLOYMENT_TARGET` to `10.7` to only depend on system features present in versions 10.7 and older.

You can see one example of building binaries using this approach here for Linux and macOS and here for Windows using AppVeyor.

Another way is to use pre-built (i.e. Docker) images that contain all the tools we need to build binaries. This allows us to easily target more exotic platforms as well. The trust project contains scripts that you can include in your project and instructions on how to set this up. It also includes support for Windows using AppVeyor.

If you'd rather set this up locally and generate the release files on your own machine, have a look at trust. It uses cross internally, which works similar to cargo but forwards commands to a cargo process inside a Docker container. The definitions of the images are also available in cross' repository.

How to install these binaries

You point your users to your release page that might look something like this one, and they can download the artifacts we've just created. The release artifacts we've generated are nothing special. They are just archive files that contain our binaries! This means that users of your tool can download them with their browser, extract them (often automatically), and copy the binaries to a place they like.

This does require some experience with manually installing programs, so you want to add a section to your README file on how to install this program.

> **Note:** If you use trust to build your binaries and add them to GitHub releases, you can also tell people to run `curl -LSfs https://japaric.github.io/trust/install.sh | sh -s -- --git your-name/repo-name` if you think that makes it easier.

When to use it

Having binary releases is a good idea in general. There's hardly any downside to it. It does not solve the problem of users having to manually install and update your tools, but they can quickly get the latest release's version without the need to install Rust.

What to package in addition to your binaries

Right now, when a user downloads our release builds, they will get a `.tar.gz` file that only contains binary files. In our example project, they will just get a single `grrs` file they can run, but there are more files we already have in our repository that they might want to have. The README file that tells them how to use this tool and the license file(s), for example. Since we already have them, they are easy to add.

There are more interesting files that make sense, especially for command-line tools. How about we ship a man page in addition to that README file and config files that add completions of the possible flags to your shell? You can write these by hand, but *clap*, the argument parsing library we use (which clap builds upon) has a way to generate all these files for us. See this in-depth chapter for more details.

Getting your app into package repositories

Both approaches we've seen so far are not how you typically install software on your machine, especially for command-line tools that you install using global package managers on most operating systems. The advantages for users are quite obvious: There is no need to think about how to install your program if it can be installed the same way as they install other tools. These package managers also allow users to update their programs when a new version is available.

Sadly, supporting different systems means you'll have to look at how these different systems work. For some, it might be as easy as adding a file to your repository (e.g. adding a Formula file like this for macOS's `brew`), but for others, you'll often need to send in patches yourself and add your tool to their repositories. There are helpful tools like cargo-bundle, cargo-deb, and cargo-aur, but describing how they work and how to correctly package your tool for those different systems is beyond the scope of this chapter.

Instead, let's have a look at a tool that is written in Rust and that is available in many different package managers.

An example: ripgrep

ripgrep is an alternative to `grep` / `ack` / `ag` and is written in Rust. It's quite successful and is packaged for many operating systems: Just look at the "Installation" section of its README!

Note that it lists a few different options on how you can install it: It starts with a link to the GitHub releases, which contain the binaries so that you can download them directly, it lists how to install it using a bunch of different package managers, and you can also install it using `cargo install`.

This seems like a very good idea. Don't pick and choose one of the approaches presented here. Start with `cargo install` and add binary releases before finally distributing your tool using system package managers.

In-depth topics

A small collection of chapters covering some more details that you might care about when writing your command line application.

Signal handling

Processes like command line applications need to react to signals sent by the operating system. The most common example is probably `Ctrl+C`, the signal that typically tells a process to terminate. To handle signals in Rust programs you need to consider how you can receive these signals as well as how you can react to them.

> **Note:** If your applications does not need to gracefully shutdown, the default handling is fine (i.e. exit immediately and let the OS cleanup resources like open file handles). In that case: No need to do what this chapter tells you!
>
> However, for applications that need to clean up after themselves, this chapter is very relevant! For example, if your application needs to properly close network connections (saying "good bye" to the processes at the other end), remove temporary files, or reset system settings, read on.

Differences between operating systems

On Unix systems (like Linux, macOS, and FreeBSD) a process can receive signals. It can either react to them in a default (OS-provided) way, catch the signal and handle them in a program-defined way, or ignore the signal entirely.

Windows does not have signals. You can use Console Handlers to define callbacks that get executed when an event occurs. There is also structured exception handling which handles all the various types of system exceptions such as division by zero, invalid access exceptions, stack overflow, and so on

First off: Handling Ctrl+C

The ctrlc crate does just what the name suggests: It allows you to react to the user pressing `Ctrl+C`, in a cross-platform way. The main way to use the crate is this:

```
use std::{thread, time::Duration};

fn main() {
    ctrlc::set_handler(move || {
        println!("received Ctrl+C!");
    })
    .expect("Error setting Ctrl-C handler");

    // Following code does the actual work, and can be interrupted by pressing
    // Ctrl-C. As an example: Let's wait a few seconds.
    thread::sleep(Duration::from_secs(2));
}
```

This is, of course, not that helpful: It only prints a message but otherwise doesn't stop the program.

In a real-world program, it's a good idea to instead set a variable in the signal handler that you then check in various places in your program. For example, you can set an `Arc<AtomicBool>` (a boolean shareable between threads) in your signal handler, and in hot loops, or when waiting for a thread, you periodically check its value and break when it becomes true.

Handling other types of signals

The ctrlc crate only handles `Ctrl+C`, or, what on Unix systems would be called `SIGINT` (the "interrupt" signal). To react to more Unix signals, you should have a look at signal-hook. Its design is described in this blog post, and it is currently the library with the widest community support.

Here's a simple example:

```
use signal_hook::{consts::SIGINT, iterator::Signals};
use std::{error::Error, thread, time::Duration};

fn main() -> Result<(), Box<dyn Error>> {
    let mut signals = Signals::new([SIGINT])?;

    thread::spawn(move || {
        for sig in signals.forever() {
            println!("Received signal {:?}", sig);
        }
    });

    // Following code does the actual work, and can be interrupted by pressing
    // Ctrl-C. As an example: Let's wait a few seconds.
    thread::sleep(Duration::from_secs(2));

    Ok(())
}
```

Using channels

Instead of setting a variable and having other parts of the program check it, you can use channels: You create a channel into which the signal handler emits a value whenever the signal is received. In your application code you use this and other channels as synchronization points between threads. Using crossbeam-channel it would look something like this:

```
use std::time::Duration;
use crossbeam_channel::{bounded, tick, Receiver, select};
use anyhow::Result;

fn ctrl_channel() -> Result<Receiver<()>, ctrlc::Error> {
    let (sender, receiver) = bounded(100);
    ctrlc::set_handler(move || {
        let _ = sender.send(());
    })?;

    Ok(receiver)
}

fn main() -> Result<()> {
    let ctrl_c_events = ctrl_channel()?;
    let ticks = tick(Duration::from_secs(1));

    loop {
        select! {
            recv(ticks) -> _ => {
                println!("working!");
            }
            recv(ctrl_c_events) -> _ => {
                println!();
                println!("Goodbye!");
                break;
            }
        }
    }

    Ok(())
}
```

Using futures and streams

If you are using tokio, you are most likely already writing your application with asynchronous patterns and an event-driven design. Instead of using crossbeam's channels directly, you can enable signal-hook's `tokio-support` feature. This allows you to call `.into_async()` on signal-hook's `Signals` types to get a new type that implements `futures::Stream`.

What to do when you receive another Ctrl+C while you're handling the first Ctrl+C

Most users will press `Ctrl+C`, and then give your program a few seconds to exit, or tell them what's going on. If that doesn't happen, they will press `Ctrl+C` again. The typical behavior is to have the application quit immediately.

Using config files

Dealing with configurations can be annoying especially if you support multiple operating systems which all have their own places for short- and long-term files.

There are multiple solutions to this, some being more low-level than others.

The easiest crate to use for this is `confy`. It asks you for the name of your application and requires you to specify the config layout via a `struct` (that is `Serialize`, `Deserialize`) and it will figure out the rest!

```
#[derive(Debug, Serialize, Deserialize)]
struct MyConfig {
    name: String,
    comfy: bool,
    foo: i64,
}

fn main() -> Result<(), io::Error> {
    let cfg: MyConfig = confy::load("my_app")?;
    println!("{:#?}", cfg);
    Ok(())
}
```

This is incredibly easy to use for which you of course surrender configurability. But if a simple config is all you want, this crate might be for you!

Configuration environments

> **TODO**
>
> 1. Evaluate crates that exist
> 2. Cli-args + multiple configs + env variables
> 3. Can `configure` do all this? Is there a nice wrapper around it?

Exit codes

A program doesn't always succeed. And when an error occurs, you should make sure to emit the necessary information correctly. In addition to telling the user about errors, on most systems, when a process exits, it also emits an exit code (an integer between 0 and 255 is compatible with most platforms). You should try to emit the correct code for your program's state. For example, in the ideal case when your program succeeds, it should exit with `0`.

When an error occurs, it gets a bit more complicated, though. In the wild, many tools exit with `1` when a common failure occurs. Currently, Rust sets an exit code of `101` when the process panicked. Beyond that, people have done many things in their programs.

So, what to do? The BSD ecosystem has collected a common definition for their exit codes (you can find them here). The Rust library `exitcode` provides these same codes, ready to be used in your application. Please see its API documentation for the possible values to use.

After you add the `exitcode` dependency to your `Cargo.toml`, you can use it like this:

```
fn main() {
    // ...actual work...
    match result {
        Ok(_) => {
            println!("Done!");
            std::process::exit(exitcode::OK);
        }
        Err(CustomError::CantReadConfig(e)) => {
            eprintln!("Error: {}", e);
            std::process::exit(exitcode::CONFIG);
        }
        Err(e) => {
            eprintln!("Error: {}", e);
            std::process::exit(exitcode::DATAERR);
        }
    }
}
```

Communicating with humans

Make sure to read the chapter on CLI output in the tutorial first. It covers how to write output to the terminal, while this chapter will talk about *what* to output.

When everything is fine

It is useful to report on the application's progress even when everything is fine. Try to be informative and concise in these messages. Don't use overly technical terms in the logs. Remember: the application is not crashing so there's no reason for users to look up errors.

Most importantly, be consistent in the style of communication. Use the same prefixes and sentence structure to make the logs easily skimmable.

Try to let your application output tell a story about what it's doing and how it impacts the user. This can involve showing a timeline of steps involved or even a progress bar and indicator for long-running actions. The user should at no point get the feeling that the application is doing something mysterious that they cannot follow.

When it's hard to tell what's going on

When communicating non-nominal state it's important to be consistent. A heavily logging application that doesn't follow strict logging levels provides the same amount, or even less information than a non-logging application.

Because of this, it's important to define the severity of events and messages that are related to it; then use consistent log levels for them. This way users can select the amount of logging themselves via `--verbose` flags or environment variables (like `RUST_LOG`).

The commonly used `log` crate defines the following levels (ordered by increasing severity):

- trace
- debug
- info
- warning
- error

It's a good idea to think of *info* as the default log level. Use it for, well, informative output. (Some applications that lean towards a more quiet output style might only show warnings and errors by default.)

Additionally, it's always a good idea to use similar prefixes and sentence structure across log messages, making it easy to use a tool like `grep` to filter for them. A message should provide enough context by itself to be useful in a filtered log while not being *too* verbose at the same time.

```
error: could not find `Cargo.toml` in `/home/you/project/`
```

```
=> Downloading repository index
=> Downloading packages...
```

The following log output is taken from wasm-pack:

```
 [1/7] Adding WASM target...
 [2/7] Compiling to WASM...
 [3/7] Creating a pkg directory...
 [4/7] Writing a package.json...
 > [WARN]: Field `description` is missing from Cargo.toml. It is not necessary, but
recommended
 > [WARN]: Field `repository` is missing from Cargo.toml. It is not necessary, but
recommended
 > [WARN]: Field `license` is missing from Cargo.toml. It is not necessary, but recommended
 [5/7] Copying over your README...
 > [WARN]: origin crate has no README
 [6/7] Installing WASM-bindgen...
 > [INFO]: wasm-bindgen already installed
 [7/7] Running WASM-bindgen...
 Done in 1 second
```

When panicking

One aspect often forgotten is that your program also outputs something when it crashes. In Rust, "crashes" are most often "panics" (i.e., "controlled crashing" in contrast to "the operating system killed the process"). By default, when a panic occurs, a "panic handler" will print some information to the console.

For example, if you create a new binary project with `cargo new --bin foo` and replace the content of `fn main` with `panic!("Hello World")`, you get this when you run your program:

```
thread 'main' panicked at 'Hello, world!', src/main.rs:2:5
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

This is useful information to you, the developer. (Surprise: the program crashed because of line 2 in your `main.rs` file). But for a user who doesn't even have access to the source code, this is not very valuable. In fact, it most likely is just confusing. That's why it's a good idea to add a custom panic handler, that provides a bit more end-user focused output.

One library that does just that is called human-panic. To add it to your CLI project, you import it and call the `setup_panic!()` macro at the beginning of your `main` function:

```
use human_panic::setup_panic;

fn main() {
   setup_panic!();

   panic!("Hello world")
}
```

This will now show a very friendly message, and tells the user what they can do:

```
Well, this is embarrassing.

foo had a problem and crashed. To help us diagnose the problem you can send us a crash
report.

We have generated a report file at "/var/folders/n3/dkk459k908lcmkzwcmq0tcv00000gn/T/report-
738e1bec-5585-47a4-8158-f1f7227f0168.toml". Submit an issue or email with the subject of
"foo Crash Report" and include the report as an attachment.

- Authors: Your Name <your.name@example.com>

We take privacy seriously, and do not perform any automated error collection. In order to
improve the software, we rely on people to submit reports.

Thank you kindly!
```

Communicating with machines

The power of command-line tools really comes to shine when you are able to combine them. This is not a new idea: In fact, this is a sentence from the Unix philosophy:

> Expect the output of every program to become the input to another, as yet unknown, program.

If our programs fulfill this expectation, our users will be happy. To make sure this works well, we should provide not just pretty output for humans, but also a version tailored to what other programs need. Let's see how we can do this.

> **Note:** Make sure to read the chapter on CLI output in the tutorial first. It covers how to write output to the terminal.

Who's reading this?

The first question to ask is: Is our output for a human in front of a colorful terminal, or for another program? To answer this, we can use the IsTerminal trait:

```
use std::io::IsTerminal;

if std::io::stdout().is_terminal() {
    println!("I'm a terminal");
} else {
    println!("I'm not");
}
```

Depending on who will read our output, we can then add extra information. Humans tend to like colors, for example, if you run `ls` in a random Rust project, you might see something like this:

```
$ ls
CODE_OF_CONDUCT.md   LICENSE-APACHE       examples
CONTRIBUTING.md      LICENSE-MIT          proptest-regressions
Cargo.lock           README.md            src
Cargo.toml           convey_derive        target
```

Because this style is made for humans, in most configurations it'll even print some of the names (like `src`) in color to show that they are directories. If you instead pipe this to a file, or a program like `cat`, `ls` will adapt its output. Instead of using columns that fit my terminal window it will print every entry on its own line. It will also not emit any colors.

```
$ ls | cat
CODE_OF_CONDUCT.md
CONTRIBUTING.md
Cargo.lock
Cargo.toml
LICENSE-APACHE
LICENSE-MIT
README.md
convey_derive
examples
proptest-regressions
src
target
```

Easy output formats for machines

Historically, the only type of output command-line tools produced were strings. This is usually fine for people in front of terminals, who are able to read text and reason about its meaning. Other programs usually don't have that ability, though: The only way for them to understand the output of a tool like `ls` is if the author of the program included a parser that happens to work for whatever `ls` outputs.

This often means that output was limited to what is easy to parse. Formats like TSV (tab-separated values), where each record is on its own line, and each line contains tab-separated content, are very popular. These simple formats based on lines of text allow tools like `grep` to be used on the output of tools like `ls`. `| grep Cargo` doesn't care if your lines are from `ls` or file, it will just filter line by line.

The downside of this is that you can't use an easy `grep` invocation to filter all the directories that `ls` gave you. For that, each directory item would need to carry additional data.

JSON output for machines

Tab-separated values is a simple way to output structured data but it requires the other program to know which fields to expect (and in which order) and it's difficult to output messages of different types. For example, let's say our program wanted to message the consumer that it is currently waiting for a download, and afterwards output a message describing the data it got. Those are very different kinds of messages and trying to unify them in a TSV output would require us to invent a way to differentiate them. Same when we wanted to print a message that contains two lists of items of varying lengths.

Still, it's a good idea to choose a format that is easily parsable in most programming languages/environments. Thus, over the last years a lot of applications gained the ability to output their data in JSON. It's simple enough that parsers exist in practically every language yet powerful enough to be useful in a lot of cases. While its a text format that can be read by humans, a lot of people have also worked on implementations that are very fast at parsing JSON data and serializing data to JSON.

In the description above, we've talked about "messages" being written by our program. This is a good way of thinking about the output: Your program doesn't necessarily only output one blob of data but may in fact emit a lot of different information while it is running. One easy way to support this approach when outputting JSON is to write one JSON document per message and to put each JSON document on new line

(sometimes called Line-delimited JSON). This can make implementations as simple as using a regular `println!`.

Here's a simple example, using the `json!` macro from serde_json to quickly write valid JSON in your Rust source code:

```
use clap::Parser;
use serde_json::json;

/// Search for a pattern in a file and display the lines that contain it.
#[derive(Parser)]
struct Cli {
    /// Output JSON instead of human readable messages
    #[arg(long = "json")]
    json: bool,
}

fn main() {
    let args = Cli::parse();
    if args.json {
        println!(
            "{}",
            json!({
                "type": "message",
                "content": "Hello world",
            })
        );
    } else {
        println!("Hello world");
    }
}
```

And here is the output:

```
$ cargo run -q
Hello world
$ cargo run -q -- --json
{"content":"Hello world","type":"message"}
```

(Running `cargo` with `-q` suppresses its usual output. The arguments after `--` are passed to our program.)

Practical example: ripgrep

ripgrep is a replacement for *grep* or *ag*, written in Rust. By default it will produce output like this:

```
$ rg default
src/lib.rs
37:    Output::default()

src/components/span.rs
6:    Span::default()
```

But given `--json` it will print:

```
$ rg default --json
{"type":"begin","data":{"path":{"text":"src/lib.rs"}}}
{"type":"match","data":{"path":{"text":"src/lib.rs"},"lines":{"text":"
Output::default()\n"},"line_number":37,"absolute_offset":761,"submatches":[{"match":
{"text":"default"},"start":12,"end":19}]}}
{"type":"end","data":{"path":{"text":"src/lib.rs"},"binary_offset":null,"stats":{"elapsed":
{"secs":0,"nanos":137622,"human":"0.000138s"},"searches":1,"searches_with_match":1,"bytes_sea
{"type":"begin","data":{"path":{"text":"src/components/span.rs"}}}
{"type":"match","data":{"path":{"text":"src/components/span.rs"},"lines":{"text":"
Span::default()\n"},"line_number":6,"absolute_offset":117,"submatches":[{"match":
{"text":"default"},"start":10,"end":17}]}}
{"type":"end","data":{"path":{"text":"src/components/span.rs"},"binary_offset":null,"stats":
{"elapsed":
{"secs":0,"nanos":22025,"human":"0.000022s"},"searches":1,"searches_with_match":1,"bytes_sear
{"data":{"elapsed_total":{"human":"0.006995s","nanos":6994920,"secs":0},"stats":
{"bytes_printed":533,"bytes_searched":11285,"elapsed":
{"human":"0.000160s","nanos":159647,"secs":0},"matched_lines":2,"matches":2,"searches":2,"sea
```

As you can see, each JSON document is an object (map) containing a `type` field. This would allow us to write a simple frontend for `rg` that reads these documents as they come in and show the matches (as well the files they are in) even while *ripgrep* is still searching.

> **Note:** This is how Visual Studio Code uses *ripgrep* for its code search.

How to deal with input piped into us

Let's say we have a program that reads the number of words in a file:

```
use clap::Parser;
use std::path::PathBuf;

/// Count the number of lines in a file
#[derive(Parser)]
#[command(arg_required_else_help = true)]
struct Cli {
    /// The path to the file to read
    file: PathBuf,
}

fn main() {
    let args = Cli::parse();
    let mut word_count = 0;
    let file = args.file;

    for line in std::fs::read_to_string(&file).unwrap().lines() {
        word_count += line.split(' ').count();
    }

    println!("Words in {}: {}", file.to_str().unwrap(), word_count)
}
```

It takes the path to a file, reads it line by line, and counts the number of words separated by a space.

When you run it, it outputs the total words in the file:

```
$ cargo run README.md
Words in README.md: 47
```

But what if we wanted to count the number of words piped into the program? Rust programs can read data passed in via stdin with the Stdin struct which you can obtain via the stdin function from the standard library. Similar to reading the lines of a file, it can read the lines from stdin.

Here's a program that counts the words of what's piped in via stdin

```
use clap::{CommandFactory, Parser};
use std::{
    fs::File,
    io::{stdin, BufRead, BufReader, IsTerminal},
    path::PathBuf,
};

/// Count the number of lines in a file or stdin
#[derive(Parser)]
#[command(arg_required_else_help = true)]
struct Cli {
    /// The path to the file to read, use - to read from stdin (must not be a tty)
    file: PathBuf,
}

fn main() {
    let args = Cli::parse();

    let word_count;
    let mut file = args.file;

    if file == PathBuf::from("-") {
        if stdin().is_terminal() {
            Cli::command().print_help().unwrap();
            ::std::process::exit(2);
        }

        file = PathBuf::from("<stdin>");
        word_count = words_in_buf_reader(BufReader::new(stdin().lock()));
    } else {
        word_count = words_in_buf_reader(BufReader::new(File::open(&file).unwrap()));
    }

    println!("Words from {}: {}", file.to_string_lossy(), word_count)
}

fn words_in_buf_reader<R: BufRead>(buf_reader: R) -> usize {
    let mut count = 0;
    for line in buf_reader.lines() {
        count += line.unwrap().split(' ').count()
    }
    count
}
```

If you run that program with text piped in, with `-` representing the intent to read from `stdin`, it'll output the word count:

```
$ echo "hi there friend" | cargo run -- -
Words from stdin: 3
```

It requires that stdin is not interactive because we're expecting input that's piped through to the program, not text that's typed in at runtime. If stdin is a tty, it outputs the help docs so that it's clear why it doesn't work.

Rendering documentation for your CLI apps

Documentation for CLIs usually consists of a `--help` section in the command and a manual (`man`) page.

Both can be automatically generated when using `clap`, via `clap_mangen` crate.

```
#[derive(Parser)]
pub struct Head {
    /// file to load
    pub file: PathBuf,
    /// how many lines to print
    #[arg(short = "n", default_value = "5")]
    pub count: usize,
}
```

Secondly, you need to use a `build.rs` to generate the manual file at compile time from the definition of your app in code.

There are a few things to keep in mind (such as how you want to package your binary) but for now we simply put the `man` file next to our `src` folder.

```
use clap::CommandFactory;

#[path="src/cli.rs"]
mod cli;

fn main() -> std::io::Result<()> {
    let out_dir = std::path::PathBuf::from(std::env::var_os("OUT_DIR").ok_or_else(||
std::io::ErrorKind::NotFound)?);
    let cmd = cli::Head::command();

    let man = clap_mangen::Man::new(cmd);
    let mut buffer: Vec<u8> = Default::default();
    man.render(&mut buffer)?;

    std::fs::write(out_dir.join("head.1"), buffer)?;

    Ok(())
}
```

When you now compile your application there will be a `head.1` file in your project directory.

If you open that in `man` you'll be able to admire your free documentation.

Resources

Collaboration / help

- cli-and-tui Discord Channel

Crates referenced in this book

- anyhow - provides `anyhow::Error` for easy error handling
- assert_cmd - simplifies integration testing of CLIs
- assert_fs - Setup input files and test output files
- clap-verbosity-flag - adds a `--verbose` flag to clap CLIs
- clap - command line argument parser
- confy - boilerplate-free configuration management
- crossbeam-channel - provides multi-producer multi-consumer channels for message passing
- ctrlc - easy ctrl-c handler
- env_logger - implements a logger configurable via environment variables
- exitcode - system exit code constants
- human-panic - panic message handler
- indicatif - progress bars and spinners
- log - provides logging abstracted over implementation
- predicates - implements boolean-valued predicate functions
- proptest - property testing framework
- serde_json - serialize/deserialize to JSON
- signal-hook - handles UNIX signals
- tokio - asynchronous runtime
- wasm-pack - tool for building WebAssembly

Other crates

Due to the constantly-changing landscape of Rust crates, a good place to find crates is the lib.rs crate index, including:

- Command-line interface
- Configuration
- Database interfaces
- Encoding
- Filesystem
- HTTP Client
- Operating systems

Other resources:

- Rust Cookbook

- rosetta-rs