Single chapter compiled via latex.

April 10, 2022

git commit: c68d1f4

Shing Lyu

■ ■ ■

Building a command-line program

2.1 Introduction

Command-line programs, also known as CLIs (Command-line Interfaces), are probably one of the most natural applications of Rust. When you compile your first Hello World program, you are already building a command-line program. A typical command-line program usually takes arguments, flags, and sometimes standard input as the input; then, it executes its main algorithm and output to the standard output or file. All these operations are well supported by the Rust standard library and third-party crates on crates.io.

There are a few advantages to building a CLI in Rust. First, the rich collection of libraries on crates.io will enable you to achieve many things you need. Second, its high performance and safety guarantees can let you mitigate many performance bottlenecks and bugs, comparing to other popular scripting languages like Python or Ruby. Finally, Rust programs can be compiled into a single, small binary containing platform-specific machine code for easy distribution, so users don't need to have a language runtime on their system.

One example of this space is the ripgrep[1] project. It is a line-oriented search tool like GNU grep, ack, or The Silver Searcher. Ripgrep has exceptional performance. It outperforms C-based GNU grep in many benchmarks[2]. But at the same time, it doesn't need to reinvent every wheel. Ripgrep builds on top of may existing libraries like the `regex` crate (regular expression parsing, compiling, and execution) and the `clap` crate (command-line argument parsing). This is a perfect example of how Rust can be fast and easy to write at the same time.

[1] https://github.com/BurntSushi/ripgrep
[2] https://blog.burntsushi.net/ripgrep/

2.2 What are we building?

`Cowsay` is a funny little command-line program originally written in Perl. It takes a text message and renders an ASCII-art cow (looks more like a horse to me, to be honest) saying that message in a speech bubble (Figure 2.1). Although this program seems pretty useless, it's still quite popular on Unix servers, where the system administrator can use it to print a light-hearted welcome message to the user.

```
% cowsay "Hello world"
 _____________
< Hello world >
 -------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

Figure 2.1: Example output of `cowsay`

`Cowsay` has a very simple algorithm, so by using it as an example, we can focus on the mechanisms and tooling to build a command-line program, instead of focusing on the business logic. We all know that a cat is the "unofficial mascot of the internet", so we are going to build a `catsay` tool that makes a cat say our message. The features will include:

- Take a string as the positional argument.
- Take a `-d`/`--dead` flag that makes the cat's eyes become `xx`, which is the comical expression of dead eyes.
- Take a `-h`/`--help` flag to print a help message.
- Take a `-v`/`--version` flag to print the version information.
- Print in color.
- Error handling: printing the error message to STDERR.
- Piping: accept STDIN as input and allow the output to be piped to other programs

- Integration tests
- Package and publish to crates.io

2.3 Creating a binary project

Although we can simply write a `.rs` file and compile it with `rustc`, handling dependencies will be a nightmare. Therefore we are going to use Cargo, Rust's package manager, to create a project and handle the dependencies for us. Cargo is capable of creating two kinds of projects, binaries and libraries. Libraries are used to build packages that are intended to be used as building blocks for other programs. Binaries are what we are trying to build in this chapter: a single program that is going to be used independently. To create a binary program, run the following command in your terminal:

```
$ cargo new --bin catsay
    Created binary (application) `catsay` package
```

The `--bin` flag stands for "binary", which tells Cargo to create the package as a binary program. You can also omit the flag because it's the default.

The command creates a `catsay` folder and some basic files:

```
catsay
|-- Cargo.toml
+-- src
    +-- main.rs
```

If you open `main.rs`, there will be a hello world program template already created for you. To run the hello world example, simply run this in the terminal inside the `catsay` folder:

```
$ cargo run
```

This will compile your project and execute the code in your `main.rs` file. (To be more precise, run the `fn main()` function in your `main.rs`).

2.4 Reading command-line arguments with `std::env::args`

The first thing we want to implement is when we pass a string to the program, print an ASCII-art cat that says it:

```
$ cargo run -- "Hello I'm a cat"
   Compiling catsay v0.1.0 (/path/to/catsay)
    Finished dev [unoptimized + debuginfo] target(s) in 1.18s
     Running `target/debug/catsay 'Hello I'\''m a cat'`
Hello I'm a cat
 \
```

```
  \
   /\_/\
  ( o o )
  =( I )=
```

NOTE The `--` following `cargo run` signifies the end of options (to `cargo`); all the arguments following the `--` will be passed to the main program in `main.rs`, which is the compiled binary `target/debug/catsay`, as you can see from the "Running..." line in the output. Also, keep in mind that the "Compiling ...", "Finished ...", and "Running ..." lines are the logs from Cargo itself. Our program's output starts after the "Running ... " line.

Printing the text and the cat is pretty straightforward with `println!()`, but how do we actually read the command-line argument? Thanks to the Rust standard library, we can use `std::env::args()`. Replace the code in `src/main.rs` as follows:

```
fn main() {
    let message = std::env::args().nth(1)
        .expect("Missing the message. Usage: catsay <message>")
    ;
    println!("{}", message);
    println!("  \\");
    println!("   \\");
    println!("    /\\_/\\");
    println!("   ( o o )");
    println!("   =( I )=");
}
```

The `std::env::args()` function returns an iterator to the arguments. The 0th argument is the name of the binary itself, `catsay`, and the string we are looking for is the next, so we can call the `nth(1)` function on the iterator to get the first argument. The `nth()` function might fail (e.g. `n` is larger than the size of the iterator) and returns an `Option::None`, so we can use `unwrap` or `expect` to get the contained value. Then we assign this value to a variable named `message` and print it out using `println!()`.

2.5 Handling complex arguments with `StructOpt`

The `std::env::args` works well for small programs with only a few options. But once we have more and more options, it becomes cumbersome to parse them by hand. For example, we might have flags that have a long and short version, e.g., `--version` and `-v`. Or we might have optional arguments that take values (e.g. `--option value`). These types of arguments are prevalent in command-line tools, but implementing them from scratch every time is a real pain. One solution to this is to use the `clap` library[3]. `Clap` can help you generate a parser for the arguments, you just have to declare the arguments you need in Rust code or YAML, then `clap` generates the command line parser and a nice-looking `--help` message for you.

To make our life even simpler, there is the great library `StructOpt`[4] that combines `clap` and custom derive. Custom derive is a feature in Rust that allows us to automatcially generate a default implmentation of a trait by annotating a struct. We can define a struct containing the arguments we want, and annotate it with `#[derive(StructOpt)]`. A macro defined by the `StructOpt` automatically `impl` the `StructOpt` trait for our struct. This implementation will contain the necessary `clap` code for parsing the arguments. We can get the parsed arguments in the struct format we defined. It's much more declarative, and the parsed struct is very easy to manipulate. To use `StructOpt` to parse our string input, we first need to add it to the `[dependencies]` section in the `Cargo.toml` file:

```
[package]
name = "catsay"
version = "0.1.0"
authors = ["Shing Lyu"]
edition = "2021"

[dependencies]
structopt = "0.3.26"
```

Then we can change `src/main.rs` to:

```
extern crate structopt;

use structopt::StructOpt;

#[derive(StructOpt)]
struct Options {
    message: String // [1]
}

fn main() {
    let options = Options::from_args(); // [2]
```

[3] Clap stands for Command Line Argument Parser. https://clap.rs/
[4] https://crates.io/crates/structopt

```
    let message = options.message;
    println!("{}", message);
    // ... print the cat
}
```

In [1], we define a struct named `Options` that has one `String` field called `message`. Then we annotate the struct with the custom derive attribute `#[derive(StructOpt)]`. This way `StructOpt` will take the struct as our argument definition and generate the argument parsers accordingly. To actually parse the arguments in `main()`, we call `Options::from_args()`, which parses the arguments and fill them into the `Options` struct and return it. We can then access the individual fields like a normal Rust struct (e.g., `options.message`).

Let's see how that looks like in action. One nice thing we get for free from using `StructOpt` is that it gives us a `--help` command for free:

```
$ cargo run -- --help
    Finished dev [unoptimized + debuginfo] target(s) in 21.02s
     Running `target/debug/catsay --help`
catsay 0.1.0

USAGE:
    catsay [message]

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

ARGS:
    <message>
```

As you can see in the help message, we have a positional argument named `<message>`.

If we forget to provide the `message` argument, `StructOpt` is smart enough to show an error message:

```
% cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.04s
     Running `target/debug/catsay`
error: The following required arguments were not provided:
    <message>

USAGE:
    catsay <message>

For more information try --help
```

However, in the current implementation, we only know there is an argument called `message`, but a new user will not know what it is for. They also don't know what kind

of value it should be. To improve the help message, we can include a description for that field and add a default value for it. If we don't provide the `message` argument, the default value will be used (Listing 2.5):

```
struct Options{
    #[structopt(default_value = "Meow!")]
    /// What does the cat say?
    message: String,
}
```

We annotated the field with `#[structopt(default_value="<default>")]`, this sets a default value for the field. The next line looks like a comment in Rust, but it starts with a triple "/" instead of double. These kinds of comments are documentation comments, which are usually used for Rust documentation (e.g., the `rustdoc` tools use it). `StructOpt` will pick that up and use it as the description for that field.

The help message will then become:

```
ARGS:
    <message>    What does the cat say? [default: Meow!]
```

2.6 Adding binary flags (a.k.a switches)

`StructOpt` also makes it really easy to add binary flags, also known as toggles or switches. `Cowsay` has a flag called `--dead` (`-d`), which will change the cow's eye from an "o" symbol to "x", a classic comical expression of dead characters. We can easily implement this by adding the code to our `Options` struct as in Listing 2.6.

```
#[derive(StructOpt)]
struct Options {
    message: String,

    #[structopt(short = "d", long = "dead")]
    /// Make the cat appears dead
    dead: bool,
}
```

We add a field of type `bool` named `dead`. We can assign the long and short version of the flag by annotating the field with `#[structopt(short = "d", long= "dead")]`. The help message will now look like:

```
FLAGS:
    -d, --dead       Make the cat appears dead
    -h, --help       Prints help information
    -V, --version    Prints version information
```

To use the flag, we can modify the `main()` function as Listing 2.6.

```
let options = Options::from_args();
let message = options.message;

let eye = if options.dead { "x" } else { "o" }; // [1]

println!("{}", message);
println!("  \\");
println!("   \\");
println!("      /\\_/\\");
println!("     ( {eye} {eye} )", eye=eye); // [2]
println!("     =( I )=");
```

When a flag has the `bool` type, its values are determined by the presence of it. If the flag is not present, it will be set to `false` and vice versa. In [1] we assign the `eye` variable to either `"o"` or `"x"` based on whether `options.dead` is `true` or `false`. Then in [2] we use `println!()` and interpolate the `"{eye}"` part into the desired eye character.

There are other types of arguments called options, which can take a value (e.g. `--value myvalue`), but we'll cover them in section 2.9. Let's shift our focus to how to handle the output.

2.7 Printing to STDERR

Up until now, we are only printing using `println!()`, which prints to the standard output (STDOUT). However, there is also the standard error (STDERR) stream that we can and should print errors to. Rust provides a STDERR equivalent of `println!()`, called `eprintln!()`. The `e-` prefix stands for error. We can demonstrate this by printing an error if the user tries to make the cat say "Woof". (Listing 2.7)

```
fn main() {
    // ...
    if message.to_lowercase() == "woof" {
        eprintln!("A cat shouldn't bark like a dog.")
    }
    // ...
}
```

We can test this by redirecting the STDOUT and STDERR to separate files:

```
cargo run "woof" 1> stdout.txt 2> stderr.txt
```

The two files will look like Listing 2.7. An interesting fact is that `cargo run` actually prints its log (i.e., the "Compiling..., Finished..." message) to STDERR. If you wish to print without a newline at the end of each line, you can use `print!()` and `eprint!()`.

```
$ cat stdout.txt
```

```
woof
 \
  \
      /\_/\
     ( o o )
    =( I )=

$ cat stderr.txt
   Compiling catsay v0.1.0 (/home/shinglyu/workspace/
    practical_rust/chap_1_cli/catsay)
    Finished dev [unoptimized + debuginfo] target(s) in 1.89s
     Running `target/debug/catsay woof`
A cat shouldn't bark like a dog.
```

2.8 Printing with color

Nowadays terminals (or terminal emulators) are ususally capable of printing in color. So we are going to make our `catsay` more colorful using the `colored` crate. First we need to add `colored` to the `Cargo.toml` file:

TODO: Use cargo add to add it instead

```
[dependencies]
//...
colored = "2.0.0"
```

Then in the `main.rs` file, we need to extern the crate and bring everything into the namespace by `use colored::*`. (Listing 2.8)

```
extern crate colored;
use colored::*;

//...

fn main() {
    //...
    println!("{}", message.bright_yellow().underline().
    ↪ on_purple());
    println!("  \\");
    println!("   \\");
    println!("      /\\_/\\");
    println!("     ( {eye} {eye} )", eye=eye.red().bold());
    println!("     =( I )=");
}
```

The `colored` crate defines a `Colorize` trait, which is implemented on a `&str` and `String`. This trait provides various chainable coloring functions:

- Coloring the text: `red()`, `green()`, `blue()`, etc.
- Coloring the background: `on_red()` (i.e. text on red background), `on_green()`, `on_blue()`, etc.
- Brighter version: `bright_red()`, `on_bright_green()`, etc.
- Styling: `bold()`, `underline()`, `italic()`, etc.

So Listing2.8 will make our message be shown in bright yellow text on a purple background, with an underline. And we make the cat's eye bloody red and bold. The end result will look like Figure 2.2.

TODO: Crop this screenshot better

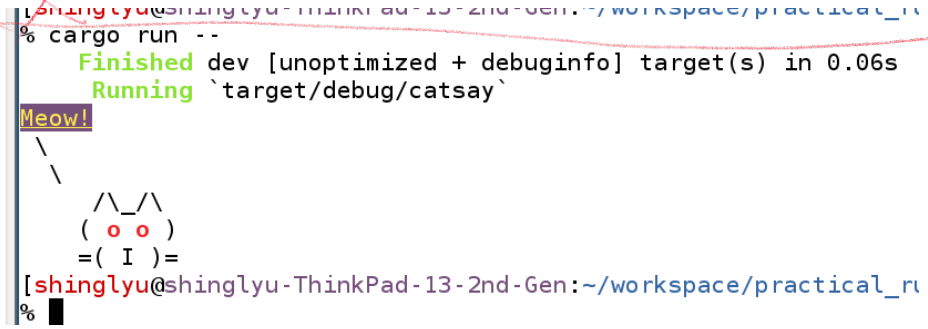

Figure 2.2: Example output of the colored `catsay`

2.9 Reading the cat picture from a file

Another common operation in command-line applications is reading from file. `Cowsay` has a `-f` option allowing you to pass in a custom cow picture file. We are going to implement a simplified version of it to demonstrate how to read files in Rust.

First, we need to add the option for reading the file as in Listing 2.9.

```
#[derive(StructOpt)]
struct Options {
    // ...

    #[structopt(short = "f", long = "file", parse(from_os_str))
    ]
    /// Load the cat picture from the specified file
    catfile: Option<std::path::PathBuf>,
}
```

There are a few things to pay attention to in Listing 2.9:

- In the `#[structopt(...)]` annotation, the short and long version of the option (`-f`/`--file`) is named differently from the field name (`catfile`) in the `Options` struct. You can name the options and flags in user-friendy terms while keeping the variable naming consistent in the code.
- In the second part of the `#[structopt(...)]` annotation, we define a custom parser scheme. By default `StructOpt` will use the `from_str` scheme, which uses the function signature `fn(&str) -> T`. But in our case, we are passing a string of path name, which might be represented differently in different operating systems[5], Therefore, we need to parse from an `&OsStr` instead.
- The type we defined for `catfile` is wrapped in an `Option<T>`, this is how we indicate that this field is optional. If the field is not provided, it will simply be an `Option::None`. There are other options like `Vec<T>`, which represents a list of arguments, and `u64`, which indicates that we want to count the occurences of a parameter, for example `-v`, `-vv` and `-vvv` that is commonly used to set verbosity level.
- Inside the `Option` we use a `std::path::PathBuf` instead of a raw string. `PathBuf` can help us handle path to files more robustly because it hides away many differences in how the operating systems represent paths.

Now if we run `cargo run -- --help` again, we'll see that a new section called "OPTIONS" has been added:

```
catsay 0.1.0

USAGE:
    catsay [FLAGS] [OPTIONS] [message]

FLAGS:
    -d, --dead       Make the cat appears dead
    -h, --help       Prints help information
    -V, --version    Prints version information

OPTIONS:
    -f, --file <catfile>    Load the cat picture from the
    ↳specified file

ARGS:
    <message>    What does the cat say? [default: Meow!]
```

Once we have the options in place, we can use the function in our `main()` function to load the external file and renders it. (Listing 2.9.)

[5]See the `OsString` documentation for why we need it: https://doc.rust-lang.org/std/ffi/struct.OsString.html

```
let options = Options::from_args();

// ...

let eye = if options.dead { "x" } else { "o" };

println!("{}", message);

match &options.catfile {
    Some(path) => {
        let cat_template = std::fs::read_to_string(path)
            .expect(&format!("could not read file {:?}", path))
    ↪ ;
        let cat_picture = cat_template.replace("{eye}", eye);
        println!("{}", &cat_picture);
    },
    None => {
        // ... print the cat as before
    }
}
```

In Listing 2.9, we use a `match` statement to check whether the `options.catfile` is a `Some(PathBuf)` or `None`. If it's `None`, we just print out the cat as before. But if it's a `Some(PathBuf)`, we use `std::fs::read_to_string(path)` to read the file content to string. An example catfile will look like Listing 2.9. To support different eyes, we put a placeholder `{eye}` in place of the eyes. But we can't simply use `format!()` to replace it with `o` or `x`, this is because `format!()` needs to know the formatting string at compile time, but our catfile string is loaded at runtime. Therefore, we need to use the `String.replace()` function to replace the eye placeholder with the actual string we are using. Alternatively, use libraries like `strfmt`[6] to have a more `format!`-flavor code.

```
 \
  \                   / )
   \ (\__/)          ( (
     ){eye} {eye} (           ) )
   ={  Y   }=         / /
     )     `-------/ /
    (                /
     \               |
    ,'\        ,    ,'
    `-'\  ,---\   | \
       _) )    `. \ /
      (__/        ) )
                  (_/
```

[6] https://github.com/vitiral/strfmt

2.10 Better error handling

Until now, we are always using `unwrap()` or `expect()` on functions that might fail, e.g. `std::fs::read_to_string`. When the return value is a `Result::Err(E)`, unwrapping it will cause the program to crash with `panic!()`. This is not always desirable because you lose the ability to recover from the failure or provide a better error message so the user can figure out what happened. The `human_panic` crate[7] can help you produce a more human-readable panic message, but it's still hard-crashing the program. A better way is to use Rust's `?` operator.

If we change the `std::fs::read_to_string(path).expect(...)` to `std::fs::read_to_string(path)?` (the `?` at the end is an operator, not a typo), it will be equivalent to Listing 2.10.

```
let cat_template = std::fs::read_to_string(path)?
// will be equivalent to
let cat_template = match std::fs::read_to_string(path) {
    Ok(file_content) => file_content,
    Err(e) => return e, // e: std::io::Error
};
```

The `?` operator performs a `match` on the `Result` returned by `read_to_string()`. If the value is `Ok(...)`, it simply unwraps the value inside it. If it's an `Err(...)`, it early returns the `Err(...)`. This is particularly useful if you have multiple potential points of failure in your function. Anyone of them failing will cause an early return with the `Err` and the function caller can then handle the error or further escalate the error to its caller.

■ **NOTE** If you are familiar with the `try!` macro in the earlier version of Rust, you might notice that the `?` operator does exactly the same thing as `try!`. This is because the `?` operator is just syntactic sugar for `try!`. It was introduced in Rust 1.13.

But you might notice that our `main()` function returns nothing yet. By using the `?` operator, our `main()` function might return a `std::io::Error`. So we have to fix our function signature to be `fn main() -> Result<(), Box<dyn std::error::Error>>`. We don't really care about the return value in the `Ok` case, so we pass a `()` for it; for the `Err` case we pass a weird looking `Box<dyn std::error::Error>`. This is a trait object, which means any type that implements the `std::error::Error` trait can be used here. Also don't forget to return an `Ok(())` at the end of the function to satisfy the function signature. (Listing 2.10)

```
fn main() -> Result<(), Box<dyn std::error::Error>> {
    // ...
    std::fs::read_to_string(path)?
```

[7] https://github.com/rust-cli/human-panic

```
    //...
    Ok(())
}
```

If we trigger an error now by providing a file path that doesn't exist, we'll get this not so-user-friendly message:

```
cargo run -- -f no/such/file.txt
    Finished dev [unoptimized + debuginfo] target(s) in 0.05s
    Running `target/debug/catsay -f no/such/file.txt`
Error: Os { code: 2, kind: NotFound, message: "No such file or
    directory" }
```

If we want to define a more user-friendly error, we can use the experimental `failure` crate (Listing 2.10).

TODO: Use cargo add here

```
[package]
//...

[dependencies]
structopt = "0.3.26"
colored = "2.0.0"
failure = "0.1.8"
```

The `failure` crate provides a `Context` struct, which wraps the original error with a human-readable and user-facing explanation of the error, called context. This is more flexible than returning just the original `Err` or a `String` error message because both of them are returned in a package. You can choose to look into the machine-readable error to recover or simply print the human-readable context. To use `Context`, we can rely on the `failure::ResultExt` extension trait, which adds a `with_context()` function on `Result`, so we can define a context message as in Listing 2.10. Now the error we return actually wraps the `io::Error` from `read_to_string()` and the "could not read file *filename*" error message (i.e., the context).

TODO: Check if failure and exitfailure is still mainstream? TODO: Clean up this example

```
// main.rs
//...
use failure::ResultExt;

//...

fn main() -> Result<(), failure::Error> {
    //...
    std::fs::read_to_string(path)
        .with_context(|_| format!("could not read file {:?}",
        ↪ path))?
```

```
    //...
    Ok(())
}
```

To print out the context in a human-friendly way when the program exits with an error, we can use the `exitfailure` crate (Listing 2.10), which is just a small wrapper for the `failure::Error` type that prints the message in a human-friendly way (Listing 2.10).

TODO: Cargo code highlighting

```
[package]
//...

[dependencies]
structopt = "0.3.26"
colored = "2.0.0"
failure = "0.1.8"
exitfailure = "0.5.1"
```

```
//...
use failure::ResultExt;
use exitfailure::ExitFailure;

// main() return ExitFailure instead of failure::Error
fn main() -> Result<(), ExitFailure> {
    //...
    std::fs::read_to_string(path)
        .with_context(
            |_| format!("could not read file {:?}", path)
        )?
    //...
    Ok(())
}
```

Now the error message looks much better:

```
$ cargo run -- -f no/such/file.txt
    Finished dev [unoptimized + debuginfo] target(s) in 0.00s
     Running `target/debug/grrs -f no/such/file.txt`
//... regular cargo compile output
Error: could not read file "no/such/file.txt"
Info: caused by No such file or directory (os error 2)
```

■ **TIP** The `failure` crate and its context are much more than just printing a human-friendly error message. If we have a function call chain, we can have a nested chain of errors; each has a context that is relevant to the layer of abstraction. It also gives ergonomic ways to manipulate the chain of errors and backtraces. It also allows easier

downcasting from a generic `Error` to a specific `Fail` type, comparing to the built-in `std::error::Error`. It's worth considering using `failure` if your command-line program grows more and more complex.

2.11 Piping to other commands

Piping is one of the most powerful designs in Unix-like operating systems, in which the output of one command can be sent directly to another command as inputs. This allows a command-line program to be designed in a modular way and work together easily. To make our `catsay` tool pipe-friendly, we need to take care of our input and output format.

Piping to STDOUT without color

We already discussed how to print separately to STDOUT and STDERR. Normally we pipe the STDOUT to another program as input. But we also added colors to our output. The way coloring works is that we add ANSI color escape codes to our output, and our terminal will interpret that color code and applies the color onto the text. We can see the raw color codes by piping the output to a file with `cargo␣run␣>␣output.txt`. The content of the `output.txt` will look like Listing 2.11.

```
^[[4;45;93mMeow!^[[0m
␣\
␣␣\
␣␣␣␣␣␣/\_/\
␣␣␣␣␣(␣^[[1;31mo^[[0m␣^[[1;31mo^[[0m␣)
␣␣␣␣␣=(␣I␣)=
```

Although many existing tools can handle these codes properly, some might still fail to recognize these color codes and treat them as raw characters. To avoid this kind of situation, we can set the `NO_COLOR` environment variable to `1` to turn off the coloring. This `NO_COLOR` environment variable is an informal standard[8] to toggle coloring on and off. The `colored` crate and many other command-line tools or libraries have already implemented this standard.

If we run `NO_COLOR=1␣cargo␣run`, you'll see there is no color anymore. If you pipe the output to a file, you'll also notice that the color code is no longer present. This should come in handy if you wish to pipe a colored output to other command-line programs.

[8] https://no-color.org/

Accepting STDIN

Taking input from STDIN is another way to interact with other programs. We can make our `catsay` takes a string from STDIN as the message argument. We can create a switch `--stdin` that enables this behavior:

```
cat "Hello world" | catsay --stdin
```

We can add one flag `--stdin` into our struct:

```
#[derive(StructOpt)]
struct Options {
    // ...
    #[structopt(short = "i", long = "stdin")]
    /// Read the message from STDIN instead of the argument
    stdin: bool,
}
```

Then in our `main()` function, whenever we see the `options.stdin` is `true`, we need to read the message from STDIN. Otherwise, we'll keep the old behavior and read from the argument `options.message`. The code is illustrated in Listing 2.11. Notice that we `use std::io` and `std::io::Read`, this is used on [2] to read the STDIN into a string. The `read_to_string()` function does not return a string. Instead, it fills the `&mut String` argument passed to it. Because it has to be mutable, we have to add a `mut` on [1].

```
// ...
use std::io::{self, Read};

fn main() -> Result<(), ExitFailure> {
    let options = Options::from_args();
    let mut message = String::new(); // [1]
    if options.stdin {
        io::stdin().read_to_string(&mut message)?; // [2]
    } else {
        message = options.message;
    };

    // print the message and cat picture...
}
```

This allows us to read the message from the standard input. Being able to interact with other programs through piping will make your program much more flexible and expandable.

2.12 Integration testing

Automated testing is a vital tool to improve our code quality. We touched quite a few topics on how to implement features into our program, but we haven't mentioned how to test these features. We now write everything in the `main()` function in the `main.rs` file. But that's not very testable. To unit test our business logic, it's better to split the functionality into a separate crate, and let our `main.rs` file import and use that crate. Then we can unit test the other crate, which contains most the business logic. These kinds of unit tests are relatively easy to write, you can follow the official Rust documentation or any introductory Rust book/course to learn how to unit test your code. In this section, we are going to focus on how to write an integration test that is specific to command-line programs.

Testing a command-line program usually involves running the command, then verify it's return code and STDOUT/STDERR output. This can be easily done by writing a shell script. But writing a shell script means that we have to implement our own assertion and test result aggregation and reporting, which Rust already supports in its unit testing framework. So we are going to use the `std::process::Command` and `assert_cmd`[9] crate to test our program in Rust.

First, let's create a folder `tests` in our project's root directory, and create a file named `integration_test.rs` as in Listing 2.12. We also need to add the `assert_cmd` crate to our Cargo.toml (Listing 2.12)

```
use std::process::Command; // Run programs
use assert_cmd::prelude::*; // Add methods on commands

#[test]
fn run_with_defaults() -> Result<(), Box<dyn std::error::Error
    ↪ >> {
    Command::cargo_bin("catsay")
        .expect("binary exists")
        .assert()
        .success();

    Ok(())
}
```

```
[package]
// ...

[dependencies]
// ...
predicates = "2.1.1"
```

[9] https://crates.io/crates/assert_cmd

We used two crates, the `std::process::Command` and `assert_cmd::prelude::*`. The `std::process::Command` crate give us a `Command` struct that can help us run a program in a newly spawned process. The `assert_cmd::prelude::*` module imports a few useful traits that extend `Command` to be more suitable for integration testing, like `cargo_bin()`, `assert()`, and `success()`.

In the main test function `run_with_defaults()`, we first initialize the command using `Command::cargo_bin`, which takes a cargo-built binary name, in our case it's "catsay". We then use `expect()` to handle cases like the binary doesn't exist, which will return an `Err(CargoError)`. Then we call `assert()` on the command, which produces an `Assert` struct, on which we can run various assertions of the status and output of the executed command. We only run a very basic assertion `success()`, which checks if the command succeeded or not.

We can run this test with `cargo test`, and we should get an output like Listing 2.12

TODO: Wrap lines

```
$ cargo test
zsh: correct 'test' to 'tests' [nyae]? n
   Compiling catsay v0.1.0
    Finished dev [unoptimized + debuginfo] target(s) in 1.04s
       Running target/debug/deps/catsay-bf24a9cbada6cbf2

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 ↵
    ↳filtered out

     Running target/debug/deps/integration_test-↵
    ↳cce770f212f0b7be

running 1 test
test run_with_defaults ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 ↵
    ↳filtered out
```

The test we just wrote was not very exciting, nor did it test much more than making sure the code runs. The next step we can take is to check the STDOUT and see if it contains the expected output. When we call our `catsay` program without any argument, it prints out a cat saying "Meow!", so we can verify if there is the string "Meow!" in the standard output. To do this, we can use the `stdout()` function from `assert_cmd`, and to check for the string we can use the `predicates` module (Listing 2.12) to build a predicate that checks for the string as in Listing 2.12

```
[package]
// ...
```

```
[dependencies]
// ...
assert_cmd = "2.0.4"
predicates = "2.1.1"
```

TODO: Explain this listing

```
// ...
use predicates::prelude::*; // Used for writing assertions

#[test]
fn run_with_defaults() -> Result<(), Box<std::error::Error>> {
    Command::cargo_bin("catsay")
        .expect("binary exists")
        .assert()
        .success()
        .stdout(predicate::str::contains("Meow!")); // Check
        ↪ stdout
    Ok(())
}
```

We can test not only positive cases but also test negative cases and error handling. For example, in Listing 2.12, we check if an invalid `-f` argument will be handled by the program correctly.

```
#[test]
fn fail_on_non_existing_file()
    -> Result<(), Box<std::error::Error>> {
    Command::cargo_bin("catsay")
        .expect("binary exists")
        .args(&["-f", "no/such/file.txt"])
        .assert()
        .failure();
    Ok(())
}
```

We pass an invalid file `no/such/file.txt` to the `-f` argument using the `.args()` function. This is equivalent to calling `catsay -f no/such/file.txt`. We expect that the program will exit with an error because it fails to read the file. Therefore, we call `.assert().failure()` to check if it actually fails.

2.13 Publishing and distributing the program

So once we are happy with our program, we want to package it in a way that anyone can easily install and use it as a command in their shell. There are several ways we can do this; each method has some trade-off between the effort on the user's side (easy to install) and the effort on the author's side (easy to build and package).

Install from source

If we run `cargo install --path ./` in our project folder, we can see cargo compiling our code in release mode, then "install" it into the `~/.cargo/bin` folder. We can then append this path to our `PATH` environment variable, and the `catsay` command should be available in the shell.

■ **TIP** The location where cargo installs your program can be overridden by setting the `CARGO_HOME` environment variable. By default its set to `$HOME/.cargo`.

You can publish the code onto any public code hosting service like GitHub or Bitbucket, or even publish a tarball, then ask your user to download the source code and run `cargo install --path ./`. But there are several drawbacks to this method:

- It's hard for the user to find our program.
- It requires knowledge on how to download the source code and compile it.
- The user needs the full Rust toolchain and a powerful computer to compile the source code.

Publish to crates.io

Nowadays, most Rust programmers search packages on crates.io. So to make our program easier to find, we can publish it to crates.io. It's very easy to publish our program on crates.io, and users can easily run `cargo install <crate name>` to download and install it.

To be able to publish on crates.io, we need to have an account and get an access token. Here are the steps to acquire one:

- Go to https://crates.io
- Click the "Log in with GitHub" (You need a GitHub account).TODO: Link to create github account
- Once logged in, click on your user name and select "Account Settings".
- Under the "API Access" section, you can generate a token. Copy that token and keep it handy.

Once you got the token, you can run `cargo login <token>` (replace `<token>` with the token you just created.) to allow Cargo to access crates.io on your behalf. Then you can run `cargo package`, which will package your program into a format that crates.io accepts. You can check the `target/package` folder to see what was generated. Once the package is ready, simply run `cargo publish` to publish it to crates.io.

Keep in mind that once the code is uploaded to crates.io, they stay there forever and can't be removed or overwritten. To update the code, you need to increase the version number in `Cargo.toml` and publish a new version. If you accidentally publish a broken version, you can use the `cargo yank` command to "yank" it. That means no new dependencies can be created against that version, but existing ones will still work. And even though the version is yanked, the code still stays public. So never publish any secret (e.g., password, access token, personal information) in your crates.io package.

Although publishing to crates.io solves the discoverability issue and takes away the burden for the user to manually download your code, the code is still compiled from scratch every time a user installs. So the user still needs to have the full Rust toolchain installed on their machine. To make it even easier for the users, we can pre-compile the project into binaries and release them directly.

Building binaries for distribution

TODO: Mention zip cross compile

Rust compiles to native code and by default links statically, so it doesn't require a heavy runtime like a Java Virtual Machine or a Python interpreter. If you run `cargo build --release`, then Cargo will compile your program in release mode, which means a higher level of optimization and less verbose logging than the default debug mode. You'll find the built binary in `target/release/catsay`. This binary can then be sent to a user using the same platform as you, and they can execute it directly without installing anything.

Notice that we said "using the same platform", this is because the binary might not run on another CPU architecture and operating system combination. In theory, we can do a cross-compilation to compile our binary for a different target platform. For example, if we are running a Linux machine with x86_64 CPU, we can compile it for an iPhone running on an ARM processor. This usually requires you to install cross-compilers and linkers, which might be tedious to set up on your own machine. But thankfully, `cross`[10] project solves it by wrapping all the cross-compilation environments into Docker images. This spins up a lightweight virtual machine in Docker, with all the cross-compilation toolchain and libraries configured to cross-compile the most portable binaries.

However, `cross` only works on x86_64 Linux machines. If you don't wish to set up a Linux machine with Docker just for the compilation, you can easily offload that task to a hosted continuous integration (CI) services. Nowadays, you can easily get free

[10] https://github.com/rust-embedded/cross

access to CI services like Travis CI and AppVeyor (for Windows) and connect them with GitHub. The `trust`[11] project provides templates to set up Travis CI and AppVeyor CI pipelines to build your binaries. For Linux builds, it actually uses `cross` underneath. For Windows builds, it relies on the Windows-based AppVeyor CI.

Once the binaries are built, we can put the binaries online for users to download. But usually, different platforms have their specific package format, which comes with package repositories and package managers. Users can effortlessly search, download, install, and update binaries using them. For example, macOS has brew formulae, Debian has Deb, and RedHat Linux has RPM. It's a good idea to submit your program into each package repositories for discoverability and easier update. But different platforms have different ways of packaging and submission, so we don't' plan to cover all of them in this book. You can find some helper tools on crates.io to help you pack for a specific format, for example, `cargo-deb` and `cargo-rpm`.

2.14 Conclusion

In this chapter, we talked about how to build a command-line program in Rust. We started with how to create a binary project and reading simple command-line arguments. Then we improved the command-line parser and started to parse more complex arguments with `StructOpt`. We looked at how to add positional arguments, binary flags, and options; and how to add description and default values to them. We also discussed how to build common command-line features like coloring, reading from a file, accepting standard input, and output to standard output and standard error. Then we showed how to run integration tests on our command-line program. Finally, we discussed various ways to publish and distribute our program.

[11] https://github.com/japaric/trust