Chapter 4

Parsing Arguments with clap

> *A command-line interface is a promise: this is what I accept, and this is what I will do with it. The sooner you stop breaking that promise by accident, the better your tool becomes.*

In Chapter 3 you built argument handling by hand. You walked `std::env::args()`, matched on strings, converted numbers with `parse()`, and printed your own usage messages. It worked, and it taught you what a command line really is: a list of strings that your program must interpret.

It also probably felt tedious. Every flag needed a branch. Every number needed an error message. Help text drifted out of sync with the code the moment you added an option. And the first time someone typed `-vvv` or `--count=3` instead of `--count 3`, your parser had nothing to say.

This chapter replaces all of that with **clap** (Command Line Argument Parser), the de facto standard for argument parsing in Rust. By the end of it you will be able to:

- Define a complete command-line interface as a plain Rust struct
- Parse flags, options, positional arguments, counts, and repeated values
- Validate input at the boundary, so the rest of your program only sees clean, typed data
- Build tools with subcommands in the style of `git` and `cargo`
- Read defaults from environment variables
- Generate shell completions
- Test your parser without ever spawning a process

You will finish with two runnable binaries: `wcr`, a word-count clone, and `tasks`, a small task tracker with subcommands.



Versions used in this chapter. All examples target `clap 4.5` and `clap_complete 4.5`, pinned in the preface's crate table and in each project's `Cargo.toml`. clap 4 changed several APIs from clap 3 (notably `ArgAction` and `value_parser`). If you find older tutorials using `#[clap(...)]` or `.takes_value(true)`, they are describing clap 3.

4.1 Why Not Parse by Hand?

Here is a hand-rolled parser for a tiny greeting tool, in the style of Chapter 3:

```rust
use std::env;
use std::process;

fn main() {
    let args: Vec<String> = env::args().skip(1).collect();
    let mut name: Option<String> = None;
    let mut count: u8 = 1;

    let mut i = 0;
    while i < args.len() {
        match args[i].as_str() {
            "-c" | "--count" => {
                i += 1;
                count = match args.get(i).and_then(|s| s.parse().ok()) {
                    Some(n) => n,
                    None => {
                        eprintln!("error: --count needs a number");
                        process::exit(2);
                    }
                };
            }
            "-h" | "--help" => {
                println!("usage: greet [-c N] NAME");
                return;
            }
            other if other.starts_with('-') => {
                eprintln!("error: unknown option {other}");
                process::exit(2);
            }
            other => name = Some(other.to_string()),
        }
        i += 1;
    }

    let Some(name) = name else {
        eprintln!("error: missing NAME");
        process::exit(2);
    };
    for _ in 0..count {
        println!("Hello, {name}!");
    }
}
```

That is 35 lines for two arguments, and it is still wrong in several ways:

- `--count=3` is not recognized.
- `-c3` is not recognized.
- A literal `--` (meaning “everything after this is positional”) is not handled.
- There is no `--version`.
- The help text is a single hand-typed line that will go stale.
- Nothing suggests `--count` when the user mistypes `--cuont`.

- Shell completion is impossible without writing more code.

Now the same tool in clap:

```rust
use clap::Parser;

/// Print a friendly greeting
#[derive(Parser, Debug)]
#[command(version, about)]
struct Cli {
    /// Who to greet
    name: String,

    /// How many times to greet
    #[arg(short, long, default_value_t = 1)]
    count: u8,
}

fn main() {
    let cli = Cli::parse();
    for _ in 0..cli.count {
        println!("Hello, {}!", cli.name);
    }
}
```

Run it:

```
$ greet Ada --count=2
Hello, Ada!
Hello, Ada!

$ greet Ada -c 300
error: invalid value '300' for '--count <COUNT>': 300 is not in 0..=255

For more information, try '--help'.

$ greet Ada --cuont 2
error: unexpected argument '--cuont' found

  tip: a similar argument exists: '--count'

Usage: greet [OPTIONS] <NAME>

For more information, try '--help'.
```

Every one of the missing features comes for free. More importantly, the struct *is* the specification. The type of `count` is `u8`, so clap knows to reject `300`. There is no separate place where the interface is described, so it cannot drift away from the code.

That is the core idea of this chapter: **declare the interface, let the library parse it, and receive typed data.**

4.2 Your First clap Program

Create a new project and add clap with the `derive` feature:

```
$ cargo new greet
$ cd greet
$ cargo add clap --features derive
```

`cargo add` writes this into `Cargo.toml`:

```
[dependencies]
clap = { version = "4.5", features = ["derive"] }
```

The `derive` feature enables the `#[derive(Parser)]` macro. Without it you only get the builder API (Section 4.9).

What the derive macro generates

When you write `#[derive(Parser)]` on a struct, clap's macro inspects every field and generates code that:

1. Builds a description of your command (its name, arguments, help text)
2. Parses `std::env::args_os()` against that description
3. Converts each matched value into the field's type
4. On any failure, prints a formatted error and exits the process

`Cli::parse()` is the entry point. It either returns a fully populated `Cli` or never returns at all. That "never returns on error" behavior is exactly what you want in `main`, and Section 4.11 shows the non-exiting variant for tests.

Where the help text comes from

Look again at the struct:

```
/// Print a friendly greeting
#[derive(Parser, Debug)]
#[command(version, about)]
struct Cli {
    /// Who to greet
    name: String,
    ...
}
```

Three things are happening here.

- **Doc comments become help text.** The `///` comment on the struct becomes the command's description, and the comment on each field becomes that argument's help.
- **`#[command(version)]`** reads the version from your `Cargo.toml` and wires up `-V` / `--version`.
- **`#[command(about)]`** with no value tells clap to use the struct's doc comment as the short description.

The result:

```
$ greet --help
Print a friendly greeting

Usage: greet [OPTIONS] <NAME>
```

```
Arguments:
  <NAME>  Who to greet

Options:
  -c, --count <COUNT>  How many times to greet [default: 1]
  -h, --help           Print help
  -V, --version        Print version
```



Write doc comments as if they are your documentation, because they are. The first line becomes the short help shown by `-h`. If you add a blank line and more text, the longer text appears only with `--help`. clap treats the two as distinct, which gives you a compact summary and a detailed manual from the same source.

4.3 The Anatomy of Arguments

Command-line arguments come in a few shapes, and clap infers which one you mean from the **field's type**. Learning this mapping is most of what you need to be productive.

Field type	Behavior	Example invocation
`String`, `PathBuf`, `u32`, …	Required value	`tool value`
`Option<T>`	Optional value	`tool` or `tool value`
`bool`	Flag, off by default	`--verbose`
`Vec<T>`	Zero or more values	`tool a b c`
`Option<Vec<T>>`	Zero or more, with presence tracking	`tool` or `tool a b`
`u8` with `ArgAction::Count`	Counted flag	`-vvv`

Positional arguments

A field with no `#[arg(short)]` or `#[arg(long)]` attribute is positional. Its position in the struct is its position on the command line.

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

#[derive(Parser, Debug)]
struct Cli {
    /// Source file
    source: PathBuf,
    /// Destination file
    dest: PathBuf,
}
```

```
$ cp-lite a.txt b.txt
```

Options and flags

Add `short`, `long`, or both to turn a field into a named argument:

```
#[derive(Parser, Debug)]
struct Cli {
    /// Print extra detail
    #[arg(short, long)]
    verbose: bool,

    /// Output file
    #[arg(short, long)]
    output: Option<PathBuf>,
}
```

- `short` with no value derives a single-letter flag from the field name: `-v` and `-o`.
- `long` with no value derives a long flag from the field name: `--verbose` and `--output`.
- You can override either: `#[arg(short = 'O', long = "out")]`.

clap accepts all the common spellings for you:

```
$ tool -o out.txt
$ tool -oout.txt
$ tool --output out.txt
$ tool --output=out.txt
$ tool -vo out.txt          # combined short flags
```

Defaults

There are two attributes for defaults, and the difference matters:

```
#[arg(long, default_value = "info")]       // a string, parsed like user input
level: String,

#[arg(long, default_value_t = 8080)]       // a typed value, shown in --help
port: u16,
```

`default_value` takes a *string* and runs it through the same parser as user input. `default_value_t` takes a *Rust value* of the field's type (requiring `Display`) and converts it for you. Prefer `default_value_t` when you can, because the compiler checks it.

Counting flags

Many tools let the user repeat a flag to increase its effect: `-v`, `-vv`, `-vvv`. In clap 4 this is spelled with `ArgAction::Count`:

```
use clap::{ArgAction, Parser};

#[derive(Parser, Debug)]
struct Cli {
    /// Increase verbosity (-v, -vv, -vvv)
    #[arg(short, long, action = ArgAction::Count)]
    verbose: u8,
}
```

```
fn main() {
    let cli = Cli::parse();
    let level = match cli.verbose {
        0 => "warn",
        1 => "info",
        2 => "debug",
        _ => "trace",
    };
    println!("log level: {level}");
}
```

You will reuse this exact pattern in Chapter 14 to drive `tracing` filters.

Multiple values

A `Vec<T>` field collects repeated values. How it collects them depends on whether the field is positional or an option:

```
#[derive(Parser, Debug)]
struct Cli {
    /// Input files
    files: Vec<PathBuf>,

    /// Exclude a pattern (repeatable)
    #[arg(short, long)]
    exclude: Vec<String>,
}
```

```
$ tool a.txt b.txt c.txt --exclude '*.log' --exclude '*.tmp'
```

The positional `files` swallows every bare argument. The option `--exclude` takes **one value per occurrence**, and you repeat the flag for more. This is usually the behavior you want, because it avoids the ambiguity of `--exclude a b c` where it is unclear whether `b` and `c` belong to the option or are positional.

> ⚠ **A `Vec` positional is optional by default.** If your tool needs at least one file, say so: `#[arg(required = true)] files: Vec<PathBuf>`. Otherwise `tool` with no arguments will parse successfully with an empty vector, and your own code must handle it.

The `--` separator

clap honors the Unix convention that `--` ends option parsing. Everything after it is positional, even if it starts with a dash:

```
$ tool -- -weird-filename.txt
```

This is how users pass files whose names begin with `-`. You do not need to write any code for it.

4.4 Validating Values at the Boundary

A recurring theme of this book is **validate once, at the edge**. If your `main` function only ever receives well-formed data, the rest of your program does not need to defend itself. clap is the edge for command-line input, and it gives you several levels of validation.

Level 1: the type itself

The field type is already a validator. `u16` rejects negatives, text, and anything over 65535. `PathBuf` accepts any string. `bool` flags take no value at all.

```
#[arg(long)]
retries: u8,   // rejects "abc", "-1", "256"
```

Level 2: ranges

For numbers with a narrower valid range than their type, use `value_parser!` with `.range()`:

```
use clap::{value_parser, Parser};

#[derive(Parser, Debug)]
struct Cli {
    /// Port to listen on (1-65535)
    #[arg(long, default_value_t = 8080, value_parser = value_parser!(u16).range(1..=65535))]
    port: u16,
}
```

```
$ server --port 0
error: invalid value '0' for '--port <PORT>': 0 is not in 1..=65535
```

Level 3: closed sets with `ValueEnum`

When an argument must be one of a fixed list of choices, model it as an enum and derive `ValueEnum`. This is almost always better than accepting a `String` and matching on it yourself.

```
use clap::{Parser, ValueEnum};

#[derive(Copy, Clone, Debug, PartialEq, Eq, ValueEnum)]
enum Format {
    Text,
    Json,
    Csv,
}

#[derive(Parser, Debug)]
struct Cli {
    /// Output format
    #[arg(long, value_enum, default_value_t = Format::Text)]
    format: Format,
}
```

```
$ tool --format yaml
error: invalid value 'yaml' for '--format <FORMAT>'
  [possible values: text, json, csv]
```

```
  tip: a similar value exists: 'json'
```

Three benefits arrive at once:

1. The user gets the list of valid values in the error and in `--help`.
2. Shell completions (Section 4.10) will offer the choices.
3. Your code uses `match cli.format { Format::Text => ..., Format::Json => ..., Format::Csv => ... }` with exhaustive checking. When you add `Format::Yaml` later, the compiler points at every place that must handle it.

You can customize the spelling of individual variants with `#[value(name = "ndjson")]` and add aliases with `#[value(alias = "j")]`.

Level 4: custom parser functions

For anything else, write a function that takes `&str` and returns `Result<T, E>`. Here is a parser for durations such as `30s`, `5m`, or `2h`:

```
use clap::Parser;
use std::time::Duration;

fn parse_duration(input: &str) -> Result<Duration, String> {
    let input = input.trim();
    let split = input
        .find(|c: char| !c.is_ascii_digit())
        .ok_or_else(|| format!("'{input}' is missing a unit (use s, m, or h)"))?;

    let (digits, unit) = input.split_at(split);
    let value: u64 = digits
        .parse()
        .map_err(|_| format!("'{input}' does not start with a number"))?;

    let seconds = match unit {
        "s" => value,
        "m" => value * 60,
        "h" => value * 3600,
        other => return Err(format!("unknown unit '{other}' (use s, m, or h)")),
    };
    Ok(Duration::from_secs(seconds))
}

#[derive(Parser, Debug)]
struct Cli {
    /// How long to wait (e.g. 30s, 5m, 2h)
    #[arg(long, value_parser = parse_duration, default_value = "10s")]
    timeout: Duration,
}

fn main() {
    let cli = Cli::parse();
    println!("waiting {:?}", cli.timeout);
}
```

```
$ tool --timeout 5m
```

```
waiting 300s

$ tool --timeout 5x
error: invalid value '5x' for '--timeout <TIMEOUT>': unknown unit 'x' (use s, m, or h)
```

The error string you return becomes part of clap's message, so write it for the person at the keyboard. The field type is `Duration`, not `String`, so your program never re-parses it.



> **Parse, don't validate.** A validator checks that a string looks right and hands the string back, so later code must trust that it was checked. A parser returns a *new type* that can only exist if the check passed. clap's `value_parser` is a parser in this sense, and it is the reason `Duration`, `Format`, and `u16` can travel through your program without defensive code.

4.5 Subcommands

Tools like `git commit`, `cargo build`, and `docker run` organize their behavior into **subcommands**. Each subcommand has its own arguments and its own help. In clap, the natural model is an enum: one variant per subcommand.

```rust
use clap::{Parser, Subcommand};

#[derive(Parser, Debug)]
#[command(version, about = "A tiny task tracker")]
struct Cli {
    #[command(subcommand)]
    command: Commands,
}

#[derive(Subcommand, Debug)]
enum Commands {
    /// Add a task
    Add {
        /// What needs doing
        title: String,
    },
    /// List tasks
    List {
        /// Include completed tasks
        #[arg(short, long)]
        all: bool,
    },
    /// Mark a task complete
    Done {
        /// Task id
        id: u32,
    },
}
```

```
fn main() {
    let cli = Cli::parse();
    match cli.command {
        Commands::Add { title } => println!("adding: {title}"),
        Commands::List { all } => println!("listing (all = {all})"),
        Commands::Done { id } => println!("completing #{id}"),
    }
}
```

```
$ tasks --help
A tiny task tracker

Usage: tasks <COMMAND>

Commands:
  add   Add a task
  list  List tasks
  done  Mark a task complete
  help  Print this message or the help of the given subcommand(s)

Options:
  -h, --help     Print help
  -V, --version  Print version

$ tasks add "Write chapter 5"
adding: Write chapter 5

$ tasks list --all
listing (all = true)
```

The design has a pleasant property: because `Commands` is an enum, the `match` in `main` is **exhaustive**. You cannot add a subcommand and forget to handle it.

Variants can be structs or tuples

For larger subcommands, define a separate struct and wrap it. This keeps `main` tidy and lets each command live in its own module (a pattern you will rely on in Chapter 8):

```
use clap::{Args, Subcommand};

#[derive(Subcommand, Debug)]
enum Commands {
    Add(AddArgs),
    List(ListArgs),
}

#[derive(Args, Debug)]
struct AddArgs {
    title: String,
    #[arg(short, long)]
    tag: Vec<String>,
}
```

```
#[derive(Args, Debug)]
struct ListArgs {
    #[arg(short, long)]
    all: bool,
}
```

`#[derive(Args)]` is the sibling of `#[derive(Parser)]`. It defines a *group of arguments* that can be embedded in a command or subcommand, and the same struct can be reused in several places.

Global arguments

Some options apply everywhere: `--verbose`, `--config`, `--color`. Mark them `global = true` and clap will accept them before or after the subcommand name:

```
#[derive(Parser, Debug)]
struct Cli {
    /// Path to the task store
    #[arg(short, long, global = true, default_value = "tasks.json")]
    file: PathBuf,

    #[command(subcommand)]
    command: Commands,
}
```

```
$ tasks --file work.json list
$ tasks list --file work.json     # also works
```

Showing help when nothing is given

By default, running `tasks` with no subcommand produces an error. Most users prefer to see the help instead. Add `arg_required_else_help`:

```
#[command(version, about, arg_required_else_help = true)]
```

If you want a *default* subcommand instead, make the field an `Option<Commands>` and handle `None` yourself.

4.6 Relationships Between Arguments

Real tools have rules like “you can’t use `--json` and `--csv` together” or “`--password` requires `--user`”. clap lets you state those rules declaratively, so the parser rejects bad combinations before your code runs.

Conflicts and requirements

```
#[derive(Parser, Debug)]
struct Cli {
    /// Suppress all output
    #[arg(short, long, conflicts_with = "verbose")]
    quiet: bool,
```

```
    /// Print extra detail
    #[arg(short, long)]
    verbose: bool,

    /// Username for authentication
    #[arg(long, requires = "password")]
    user: Option<String>,

    /// Password for authentication
    #[arg(long, requires = "user")]
    password: Option<String>,
}
```

```
$ tool --quiet --verbose
error: the argument '--quiet' cannot be used with '--verbose'

$ tool --user ada
error: the following required arguments were not provided:
  --password <PASSWORD>
```

The string passed to `conflicts_with` and `requires` is the **field name** (the argument's ID), not the flag spelling.

Argument groups

When the rule involves more than two arguments, such as "provide exactly one of these", use an `ArgGroup`:

```
use clap::{ArgGroup, Parser};
use std::path::PathBuf;

#[derive(Parser, Debug)]
#[command(group(
    ArgGroup::new("source")
        .required(true)
        .args(["file", "url"]),
))]
struct Cli {
    /// Read from a local file
    #[arg(long)]
    file: Option<PathBuf>,

    /// Read from a URL
    #[arg(long)]
    url: Option<String>,
}
```

By default, members of a group are mutually exclusive. With `.required(true)`, the user must supply one. If you want "at least one, but several allowed," add `.multiple(true)`.



Rules in the parser, not in `main`. The alternative to groups is checking `cli.file.is_some() == cli.url.is_some()` in your own code and printing your own error. That works, but you lose the consistent error format, the usage line in the message, and the information in `--help`. Prefer the declarative form whenever clap can express the rule.

4.7 Environment Variables

A well-behaved CLI often lets users set a default through the environment, which is useful for CI systems, containers, and secrets that should not appear in shell history. Enable the `env` feature:

```
clap = { version = "4.5", features = ["derive", "env"] }
```

Then name the variable:

```
#[derive(Parser, Debug)]
struct Cli {
    /// API token
    #[arg(long, env = "MYTOOL_TOKEN", hide_env_values = true)]
    token: String,

    /// Server address
    #[arg(long, env = "MYTOOL_SERVER", default_value = "https://example.com")]
    server: String,
}
```

The precedence is the one users expect, from highest to lowest:

1. A value on the command line
2. The environment variable
3. The default value

`hide_env_values = true` stops `--help` from printing the *current* value of the variable, which matters for tokens. Without it, running `--help` in a shell where `MYTOOL_TOKEN` is set would echo the secret into your terminal scrollback.



Never make a secret available *only* as a flag. Command-line arguments are visible to every user on the machine through `ps`, and they land in shell history. Offer an environment variable or a file path alongside any option that carries a credential, and document that in the help text.

4.8 Polishing the Interface

A CLI is a user interface, so small details compound. These are the finishing touches that make a tool feel professional.

Meaningful value names

By default, clap labels a value with its field name in capitals: `--output <OUTPUT>`. When a better name exists, use it:

```
#[arg(short, long, value_name = "FILE")]
output: Option<PathBuf>,
```

This shows `-o, --output <FILE>` in the help, which tells the user what to type.

Short and long descriptions

```
/// Search files quickly
///
/// fsearch walks a directory tree in parallel and prints every path
/// matching the given pattern. It respects .gitignore files by default.
#[derive(Parser, Debug)]
#[command(version)]
struct Cli { /* ... */ }
```

`-h` shows only “Search files quickly”. `--help` shows the full text.

Examples and trailing text

Real examples in `--help` are worth more than paragraphs of prose:

```
#[command(after_help = "\
EXAMPLES:
  fsearch 'TODO' src/
  fsearch --ext rs --ext toml 'serde'
")]
```

`after_long_help` does the same but appears only with `--help`.

Help headings

When a command has many options, group them:

```
#[arg(long, help_heading = "Output")]
format: Format,
```

Exit codes

clap follows the convention you learned in Chapter 3:

Situation	Exit code
Successful parse, `--help`, `--version`	`0`
Invalid usage (bad value, unknown flag, missing argument)	`2`

Your own errors, such as “file not found,” are your responsibility. Chapter 7 shows how to map them to exit codes cleanly.

Where output goes

Parse errors go to **stderr**. `--help` and `--version` go to **stdout**, so that `tool --help | less` and `tool --version > version.txt` behave correctly. clap handles both for you.

4.9 The Builder API (and When to Use It)

Everything so far used the **derive API**. clap has a second interface, the **builder API**, which constructs the same command at runtime from method calls:

```
use clap::{Arg, ArgAction, Command};

fn main() {
    let matches = Command::new("greet")
        .version("1.0")
        .about("Print a friendly greeting")
        .arg(Arg::new("name").help("Who to greet").required(true))
        .arg(
            Arg::new("shout")
                .short('s')
                .long("shout")
                .help("Use upper case")
                .action(ArgAction::SetTrue),
        )
        .get_matches();

    let name = matches.get_one::<String>("name").unwrap();
    let shout = matches.get_flag("shout");

    if shout {
        println!("HELLO, {}!", name.to_uppercase());
    } else {
        println!("Hello, {name}!");
    }
}
```

Notice what is lost: arguments are retrieved **by string name** and cast to a type at the call site. A typo in `"shout"` compiles fine and panics at runtime. The derive API moves those checks to compile time.

Choose the builder API when the interface is **not known until runtime**, for example when a plugin system adds commands dynamically, or when arguments are generated from a config file. For everything else, use derive. The two can be mixed: `Cli::command()` (from the `CommandFactory` trait) hands you the builder object for a derived parser, which you will use in the next section.

4.10 Shell Completions and Man Pages

Tab completion is the difference between a tool people tolerate and a tool people enjoy. Because clap already knows every flag, subcommand, and enum value, it can generate completion scripts with the companion crate `clap_complete`.

```
$ cargo add clap_complete
```

A common pattern is to add a hidden or explicit `completions` subcommand:

```rust
use clap::{CommandFactory, Parser, Subcommand};
use clap_complete::{generate, Shell};
use std::io;

#[derive(Parser)]
#[command(name = "tasks", version)]
struct Cli {
    #[command(subcommand)]
    command: Commands,
}

#[derive(Subcommand)]
enum Commands {
    /// Print a shell completion script
    Completions {
        /// Target shell
        shell: Shell,
    },
    // ... other subcommands
}

fn main() {
    let cli = Cli::parse();
    match cli.command {
        Commands::Completions { shell } => {
            let mut cmd = Cli::command();
            let name = cmd.get_name().to_string();
            generate(shell, &mut cmd, name, &mut io::stdout());
        }
    }
}
```

`Shell` already implements `ValueEnum`, so `tasks completions bash`, `zsh`, `fish`, `powershell`, and `elvish` all work with no extra code. Users install the output once:

```
$ tasks completions bash > ~/.local/share/bash-completion/completions/tasks
$ tasks completions zsh  > ~/.zfunc/_tasks
$ tasks completions fish > ~/.config/fish/completions/tasks.fish
```

For **man pages**, the sibling crate `clap_mangen` renders a `Command` to roff format in a few lines, typically from a `build.rs` or an `xtask`. Chapter 20 covers packaging both completions and man pages into release artifacts.

4.11 Testing Your Parser

Your argument definitions are code, and code can be wrong. A flag can be declared twice, a `requires` can name a field that does not exist, or two options can claim the same short letter. The good news is that you can test all of this without spawning a process.

Verify the definition itself

```
#[cfg(test)]
mod tests {
    use super::*;
    use clap::CommandFactory;

    #[test]
    fn verify_cli() {
        Cli::command().debug_assert();
    }
}
```

`debug_assert()` runs clap's internal consistency checks: duplicate short flags, invalid references in `requires`/`conflicts_with`, groups naming missing arguments, and more. Some of these are **only** checked in debug builds at runtime. Without this test, a misconfigured parser might not complain until a user triggers the broken path. **Add this test to every CLI you write.**

Parse from a slice

`try_parse_from` behaves like `parse` but returns a `Result` instead of exiting. The first element is the program name, as in `argv[0]`.

```
#[cfg(test)]
mod tests {
    use super::*;
    use clap::error::ErrorKind;

    #[test]
    fn parses_count() {
        let cli = Cli::try_parse_from(["greet", "Ada", "-c", "3"]).unwrap();
        assert_eq!(cli.name, "Ada");
        assert_eq!(cli.count, 3);
    }

    #[test]
    fn count_defaults_to_one() {
        let cli = Cli::try_parse_from(["greet", "Ada"]).unwrap();
        assert_eq!(cli.count, 1);
    }

    #[test]
    fn missing_name_is_an_error() {
        let err = Cli::try_parse_from(["greet"]).unwrap_err();
        assert_eq!(err.kind(), ErrorKind::MissingRequiredArgument);
```

```
    }

    #[test]
    fn rejects_unknown_flag() {
        let err = Cli::try_parse_from(["greet", "Ada", "--nope"]).unwrap_err();
        assert_eq!(err.kind(), ErrorKind::UnknownArgument);
    }
}
```

These tests run in microseconds, need no files, and document the interface as precisely as the help text does. Chapter 15 extends this approach to full end-to-end tests of the compiled binary with `assert_cmd`.

4.12 Mini Project: `wcr`, a Word Counter

Time to put the pieces together. `wcr` is a simplified clone of the Unix `wc` utility. It counts lines, words, bytes, and characters in files or standard input.

Features

- Zero or more file arguments; with none, or with `-`, it reads from stdin
- `-l`/`--lines`, `-w`/`--words`, `-c`/`--bytes`, `-m`/`--chars` select what to count
- With no selection flags, it shows lines, words, and bytes
- With several files, it prints a total
- An unreadable file is reported, counting continues, and the exit status becomes `1`

Setup

```
$ cargo new wcr
$ cd wcr
$ cargo add clap --features derive
$ cargo add anyhow
```

`Cargo.toml`:

```
[package]
name = "wcr"
version = "0.1.0"
edition = "2021"

[dependencies]
anyhow = "1"
clap = { version = "4.5", features = ["derive"] }
```

`src/main.rs`

```
use std::fs::File;
use std::io::{self, BufRead, BufReader};
use std::ops::AddAssign;
use std::path::{Path, PathBuf};
use std::process::ExitCode;
```

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

/// Count lines, words, bytes, and characters
///
/// With no FILE, or when FILE is -, read standard input.
#[derive(Parser, Debug)]
#[command(name = "wcr", version)]
struct Cli {
    /// Files to read ('-' means standard input)
    files: Vec<PathBuf>,

    /// Print the line count
    #[arg(short, long)]
    lines: bool,

    /// Print the word count
    #[arg(short, long)]
    words: bool,

    /// Print the byte count
    #[arg(short = 'c', long)]
    bytes: bool,

    /// Print the character count
    #[arg(short = 'm', long, conflicts_with = "bytes")]
    chars: bool,
}

/// Which columns to print.
#[derive(Debug, Clone, Copy)]
struct Show {
    lines: bool,
    words: bool,
    bytes: bool,
    chars: bool,
}

impl Show {
    fn from_cli(cli: &Cli) -> Self {
        let any = cli.lines || cli.words || cli.bytes || cli.chars;
        if any {
            Show {
                lines: cli.lines,
                words: cli.words,
                bytes: cli.bytes,
                chars: cli.chars,
            }
        } else {
            // Same defaults as the classic wc.
```

```
            Show { lines: true, words: true, bytes: true, chars: false }
        }
    }
}

#[derive(Debug, Default, Clone, Copy)]
struct Counts {
    lines: usize,
    words: usize,
    bytes: usize,
    chars: usize,
}

impl AddAssign for Counts {
    fn add_assign(&mut self, rhs: Self) {
        self.lines += rhs.lines;
        self.words += rhs.words;
        self.bytes += rhs.bytes;
        self.chars += rhs.chars;
    }
}

/// Count everything in one pass over the reader.
fn count<R: BufRead>(mut reader: R) -> io::Result<Counts> {
    let mut counts = Counts::default();
    let mut buf = Vec::new();

    loop {
        buf.clear();
        let n = reader.read_until(b'\n', &mut buf)?;
        if n == 0 {
            break;
        }
        counts.bytes += n;
        if buf.ends_with(b"\n") {
            counts.lines += 1;
        }
        // Lossy conversion keeps us going on files that aren't valid UTF-8.
        let text = String::from_utf8_lossy(&buf);
        counts.chars += text.chars().count();
        counts.words += text.split_whitespace().count();
    }
    Ok(counts)
}

fn count_path(path: &Path) -> Result<Counts> {
    if path.as_os_str() == "-" {
        Ok(count(io::stdin().lock())?)
    } else {
        let file = File::open(path)
```

```
            .with_context(|| format!("cannot open '{}'", path.display()))?;
        Ok(count(BufReader::new(file))?)
    }
}

fn print_row(counts: &Counts, show: Show, label: &str) {
    let mut row = String::new();
    if show.lines {
        row += &format!("{:>8}", counts.lines);
    }
    if show.words {
        row += &format!("{:>8}", counts.words);
    }
    if show.bytes {
        row += &format!("{:>8}", counts.bytes);
    }
    if show.chars {
        row += &format!("{:>8}", counts.chars);
    }
    if !label.is_empty() {
        row += &format!(" {label}");
    }
    println!("{row}");
}

fn main() -> ExitCode {
    let cli = Cli::parse();
    let show = Show::from_cli(&cli);

    let inputs = if cli.files.is_empty() {
        vec![PathBuf::from("-")]
    } else {
        cli.files.clone()
    };

    let mut total = Counts::default();
    let mut failed = false;

    for path in &inputs {
        match count_path(path) {
            Ok(counts) => {
                total += counts;
                let label = if path.as_os_str() == "-" {
                    String::new()
                } else {
                    path.display().to_string()
                };
                print_row(&counts, show, &label);
            }
            Err(err) => {
```

```
                eprintln!("wcr: {err:#}");
                failed = true;
            }
        }
    }

    if inputs.len() > 1 {
        print_row(&total, show, "total");
    }

    if failed {
        ExitCode::from(1)
    } else {
        ExitCode::SUCCESS
    }
}
```

Try it

```
$ cargo run -- --help
Count lines, words, bytes, and characters

With no FILE, or when FILE is -, read standard input.

Usage: wcr [OPTIONS] [FILES]...

Arguments:
  [FILES]...  Files to read ('-' means standard input)

Options:
  -l, --lines    Print the line count
  -w, --words    Print the word count
  -c, --bytes    Print the byte count
  -m, --chars    Print the character count
  -h, --help     Print help
  -V, --version  Print version

$ printf 'one two\nthree\n' | cargo run -q
       2       3      14

$ cargo run -q -- -l src/main.rs Cargo.toml
     172 src/main.rs
      11 Cargo.toml
     183 total

$ cargo run -q -- -c -m src/main.rs
error: the argument '--bytes' cannot be used with '--chars'

Usage: wcr --bytes [FILES]...
```

What to notice

1. **The interface is 20 lines.** Everything else is the program's actual logic. The struct definition is the entire argument-handling layer.
2. **The short letters are explicit.** A bare `short` derives the letter from the field name, so `bytes` would become `-b` and `chars` would become `-c`, the opposite of what classic `wc` users expect. Spelling out `short = 'c'` and `short = 'm'` avoids that surprise, and `debug_assert()` would catch any genuine duplicates.
3. **`conflicts_with`** gives us validation with a single attribute.
4. **Defaults live in one place.** `Show::from_cli` is the only function that knows the "no flags means lines, words, bytes" rule, and it is trivially testable.
5. **`ExitCode`** gives `main` an honest exit status without calling `process::exit`, so destructors still run.
6. **`{err:#}`** prints an `anyhow` error with its whole cause chain on one line, which is what you want for a CLI.

4.13 Mini Project: `tasks`, a Task Tracker with Subcommands

The second project is larger and exercises subcommands, enums, repeated options, global arguments, environment variables, and completions together. `tasks` stores its data as JSON.

```
$ tasks add "Write chapter 5" --priority high --tag book
Added task 1
$ tasks add "Buy coffee"
Added task 2
$ tasks list
[ ]   1  high   Write chapter 5 [book]
[ ]   2  normal Buy coffee
$ tasks done 1
Completed task 1
$ tasks list --all --format json
```

Setup

```
$ cargo new tasks
$ cd tasks
$ cargo add clap --features derive,env
$ cargo add clap_complete
$ cargo add anyhow serde_json
$ cargo add serde --features derive
```

`Cargo.toml`:

```toml
[package]
name = "tasks"
version = "0.1.0"
edition = "2021"

[dependencies]
anyhow = "1"
```

```
clap = { version = "4.5", features = ["derive", "env"] }
clap_complete = "4.5"
serde = { version = "1", features = ["derive"] }
serde_json = "1"
```

src/main.rs

```
use std::fs;
use std::io;
use std::path::{Path, PathBuf};

use anyhow::{bail, Context, Result};
use clap::{CommandFactory, Parser, Subcommand, ValueEnum};
use clap_complete::{generate, Shell};
use serde::{Deserialize, Serialize};

// ---------------------------------------------------------------
// Command-line interface
// ---------------------------------------------------------------

/// A tiny task tracker
#[derive(Parser, Debug)]
#[command(name = "tasks", version, arg_required_else_help = true)]
struct Cli {
    /// Path to the task store
    #[arg(
        short,
        long,
        global = true,
        env = "TASKS_FILE",
        default_value = "tasks.json",
        value_name = "FILE"
    )]
    file: PathBuf,

    #[command(subcommand)]
    command: Cmd,
}

#[derive(Subcommand, Debug)]
enum Cmd {
    /// Add a task
    Add {
        /// What needs doing
        title: String,

        /// Task priority
        #[arg(short, long, value_enum, default_value_t = Priority::Normal)]
        priority: Priority,

        /// Attach a tag (repeatable)
        #[arg(short, long = "tag", value_name = "TAG")]
        tags: Vec<String>,
    },

    /// List tasks
    List {
        /// Include completed tasks
        #[arg(short, long)]
        all: bool,
```

```
        /// Output format
        #[arg(long, value_enum, default_value_t = Format::Text)]
        format: Format,
    },

    /// Mark a task complete
    Done {
        /// Task id
        id: u32,
    },

    /// Remove a task
    Rm {
        /// Task id
        id: u32,
    },

    /// Print a shell completion script
    Completions {
        /// Target shell
        shell: Shell,
    },
}

#[derive(Serialize, Deserialize, Debug, Clone, Copy, PartialEq, Eq, ValueEnum)]
#[serde(rename_all = "lowercase")]
enum Priority {
    Low,
    Normal,
    High,
}

#[derive(Debug, Clone, Copy, ValueEnum)]
enum Format {
    Text,
    Json,
}

// ---------------------------------------------------------------------
// Data model and storage
// ---------------------------------------------------------------------

#[derive(Serialize, Deserialize, Debug)]
struct Task {
    id: u32,
    title: String,
    priority: Priority,
    tags: Vec<String>,
    done: bool,
}

fn load(path: &Path) -> Result<Vec<Task>> {
    match fs::read_to_string(path) {
        Ok(text) => serde_json::from_str(&text)
            .with_context(|| format!("'{}' does not contain valid task data", path.display())),
        // A missing file just means "no tasks yet".
        Err(e) if e.kind() == io::ErrorKind::NotFound => Ok(Vec::new()),
        Err(e) => Err(e).with_context(|| format!("cannot read '{}'", path.display())),
    }
}
```

```rust
fn save(path: &Path, tasks: &[Task]) -> Result<()> {
    let json = serde_json::to_string_pretty(tasks)?;
    fs::write(path, json).with_context(|| format!("cannot write '{}'", path.display()))
}

// ---------------------------------------------------------------
// Commands
// ---------------------------------------------------------------

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

    match cli.command {
        Cmd::Add { title, priority, tags } => {
            let mut tasks = load(&cli.file)?;
            let id = tasks.iter().map(|t| t.id).max().unwrap_or(0) + 1;
            tasks.push(Task { id, title, priority, tags, done: false });
            save(&cli.file, &tasks)?;
            println!("Added task {id}");
        }

        Cmd::List { all, format } => {
            let tasks = load(&cli.file)?;
            let shown: Vec<&Task> = tasks.iter().filter(|t| all || !t.done).collect();

            match format {
                Format::Json => println!("{}", serde_json::to_string_pretty(&shown)?),
                Format::Text => {
                    if shown.is_empty() {
                        println!("No tasks.");
                    }
                    for t in shown {
                        let mark = if t.done { "x" } else { " " };
                        let priority = format!("{:?}", t.priority).to_lowercase();
                        let tags = if t.tags.is_empty() {
                            String::new()
                        } else {
                            format!(" [{}]", t.tags.join(", "))
                        };
                        println!("[{mark}] {:>3}  {priority:<6} {}{tags}", t.id, t.title);
                    }
                }
            }
        }

        Cmd::Done { id } => {
            let mut tasks = load(&cli.file)?;
            let Some(task) = tasks.iter_mut().find(|t| t.id == id) else {
                bail!("no task with id {id}");
            };
            task.done = true;
            save(&cli.file, &tasks)?;
            println!("Completed task {id}");
        }

        Cmd::Rm { id } => {
            let mut tasks = load(&cli.file)?;
            let before = tasks.len();
            tasks.retain(|t| t.id != id);
            if tasks.len() == before {
```

```
                bail!("no task with id {id}");
            }
            save(&cli.file, &tasks)?;
            println!("Removed task {id}");
        }

        Cmd::Completions { shell } => {
            let mut cmd = Cli::command();
            let name = cmd.get_name().to_string();
            generate(shell, &mut cmd, name, &mut io::stdout());
        }
    }

    Ok(())
}

// ---------------------------------------------------------------
// Tests
// ---------------------------------------------------------------

#[cfg(test)]
mod tests {
    use super::*;
    use clap::error::ErrorKind;

    #[test]
    fn verify_cli() {
        Cli::command().debug_assert();
    }

    #[test]
    fn add_with_tags_and_priority() {
        let cli = Cli::try_parse_from([
            "tasks", "add", "Ship it", "-p", "high", "-t", "book", "-t", "urgent",
        ])
        .unwrap();

        match cli.command {
            Cmd::Add { title, priority, tags } => {
                assert_eq!(title, "Ship it");
                assert_eq!(priority, Priority::High);
                assert_eq!(tags, ["book", "urgent"]);
            }
            other => panic!("expected Add, got {other:?}"),
        }
    }

    #[test]
    fn global_file_flag_works_after_subcommand() {
        let cli = Cli::try_parse_from(["tasks", "list", "--file", "work.json"]).unwrap();
        assert_eq!(cli.file, PathBuf::from("work.json"));
    }

    #[test]
    fn rejects_unknown_priority() {
        let err = Cli::try_parse_from(["tasks", "add", "x", "-p", "urgent"]).unwrap_err();
        assert_eq!(err.kind(), ErrorKind::InvalidValue);
    }

    #[test]
    fn rejects_non_numeric_id() {
```

```
        let err = Cli::try_parse_from(["tasks", "done", "abc"]).unwrap_err();
        assert_eq!(err.kind(), ErrorKind::ValueValidation);
    }
}
```

Try it

```
$ cargo run -q -- add "Write chapter 5" -p high -t book
Added task 1
$ cargo run -q -- add "Buy coffee"
Added task 2
$ cargo run -q -- list
[ ]   1  high   Write chapter 5 [book]
[ ]   2  normal Buy coffee
$ cargo run -q -- done 1
Completed task 1
$ cargo run -q -- list
[ ]   2  normal Buy coffee
$ TASKS_FILE=/tmp/other.json cargo run -q -- list
No tasks.
$ cargo run -q -- done 99
Error: no task with id 99
$ cargo test
```

Design decisions worth discussing

- **`Priority` derives both `ValueEnum` and `Serialize`.** One type serves as the parsed CLI input *and* the stored data format. For a small tool this is a fair trade. In a larger application, you would likely separate the CLI types from the domain types so that changing a flag name does not change your file format. Chapter 8 discusses where that boundary belongs.
- **`Cmd::Completions` lives beside the real commands.** Completion generation needs no task file, but because `file` is a global argument with a default, nothing breaks. If it did, you would reach for `#[command(hide = true)]` or handle the subcommand before loading anything.
- **Errors use `anyhow::bail!`.** The `main` function returns `Result<()>`, so any `Err` prints `Error: ...` to stderr and exits with status `1`. Chapter 7 refines this into typed errors and consistent exit codes.
- **Parsing is tested; storage is not (yet).** The tests above cover everything clap is responsible for. Testing `load` and `save` against a temporary directory is one of this chapter's exercises.

4.14 Common Mistakes

1. **Forgetting the `derive` feature.**

`cargo add clap` alone gives you the builder API only. You will get a confusing "cannot find derive macro `Parser`" error. Use `cargo add clap --features derive`.

2. **Using `#[clap(...)]` instead of `#[command(...)]` and `#[arg(...)]`.**

This is clap 3 syntax from older tutorials. In clap 4, struct-level options use `#[command(...)]`, field-level options use `#[arg(...)]`, and subcommand fields use `#[command(subcommand)]`.

3. **Declaring a `bool` and expecting it to take a value.**

A `bool` field is a flag: `--verbose` sets it to `true`, and there is no `--verbose=false`. If you need an explicit on/off value, use a `ValueEnum` or `#[arg(action = ArgAction::Set)]` and let the user write `--color=false`.

4. **Letting two arguments share a short flag.**

Two fields starting with the same letter both want the same derived short. clap panics in debug builds, but only when that code path runs. Always include the `debug_assert()` test.

5. **Parsing strings that should have been types.**

`format: String` followed by `match format.as_str() { "json" => ..., _ => unreachable!() }` throws away everything clap could do for you. Use `ValueEnum`.

6. **Doing real work inside a `value_parser`.**

A parser runs at parse time, possibly several times, and before `--help` or `--version` are handled. Keep it a pure, cheap function. Do not open files or make network calls there. Take a `PathBuf` and open the file in `main`.

7. **Calling `parse()` inside library code or tests.**

`Cli::parse()` exits the process on error. Use it only in `main`. Everywhere else, use `try_parse_from`.

8. **Printing help text by hand.**

If you ever write `println!("usage: ...")`, you are duplicating what clap generates. Use `Cli::command().print_help()` when you need to show help programmatically.

9. **Putting secrets in flags only.**

See the warning in Section 4.7. Provide an environment variable or file alternative.

10. **Ignoring `Vec` semantics on options.**

`--tag a b c` does not mean three tags. It means `--tag a` plus two positional arguments. Repeat the flag, or opt in to multi-value behavior explicitly with `num_args`, and understand the ambiguity that you are accepting.

4.15 Best Practices

- **Let the types carry the rules.** Prefer `u16`, `PathBuf`, `Duration`, and enums over `String`. Every rule you move into a type is one fewer `if` in `main`.
- **Parse once, then pass typed data.** Do not hand `Cli` to every function in your program. Destructure it in `main` and pass only the values each function needs, or convert it into a purpose-built configuration struct.
- **Write the help text first.** If you cannot describe an option in one clear line, the option is probably poorly designed.
- **Follow platform conventions.** Use `-h`/`--help`, `-V`/`--version`, `-v` for verbose, `-q` for quiet, `-o` for output, `-n` for dry-run or count. Users carry expectations from every other tool they use.
- **Use `ValueEnum` for every closed set.** You get validation, help, completions, and exhaustive `match` together.
- **Use `global = true` for cross-cutting options** such as `--verbose`, `--config`, and `--color`.
- **Make destructive commands explicit.** Offer `--dry-run` and require `--force` or a confirmation for anything irreversible.

- **Test the parser.** Include one `debug_assert()` test and a few `try_parse_from` tests in every CLI you build.
- **Ship completions.** It takes ten lines, and users notice.
- **Keep parsing separate from doing.** Parsing arguments and executing commands are different responsibilities. This separation is what makes the later chapters on testing and architecture straightforward.

4.16 Challenge Exercises

Exercise 1: Add `--head N` to `wcr`. Add an option that stops reading each input after `N` lines. Use `value_parser!(usize).range(1..)` so that `0` is rejected at parse time. Make sure the option appears under its own help heading.

Exercise 2: Duration-aware `wcr`. Take the `parse_duration` function from Section 4.4 and add a `--timeout` option to a program that reads from stdin. (Implementing the timeout itself is hard with blocking I/O, since you will return to it in Chapter 11. For now, parse it and print the value.)

Exercise 3: Tag filtering in `tasks`. Add `--tag <TAG>` (repeatable) to `tasks list` so that only tasks having *all* given tags are shown. Add `--any-tag` as a flag that changes the meaning to "at least one." Use `requires` so that `--any-tag` is rejected unless at least one `--tag` is present.

Exercise 4: Edit command. Add `tasks edit <ID>` with optional `--title`, `--priority`, and `--add-tag` / `--remove-tag` arguments. Use an `ArgGroup` so that at least one change must be provided.

Exercise 5: Test the storage layer. Write tests for `load` and `save` using `std::env::temp_dir()`. Test the missing-file case, the corrupt-file case, and a round trip. (Chapter 15 introduces the `tempfile` crate, which makes this cleaner.)

Exercise 6: Rewrite `greet` with the builder API. Take the derive version from Section 4.1 and rewrite it with `Command::new(...)`. Compare the two. Which errors does the compiler catch in each?

Exercise 7: Color control. Add a global `--color <auto|always|never>` option as a `ValueEnum`. Then use clap's `ColorChoice` or the `NO_COLOR` environment variable to decide whether `tasks list` prints ANSI colors. (Hint: Chapter 12 covers terminal styling in depth.)

4.17 Recap

In this chapter you replaced hand-written argument parsing with a declarative approach, and you saw how much follows from that one change.

- **The struct is the interface.** With `#[derive(Parser)]`, fields become arguments, doc comments become help, and types become validation.
- **Field types map to argument shapes:** `T` is required, `Option<T>` optional, `bool` a flag, `Vec<T>` repeated, and `u8` with `ArgAction::Count` a counted flag like `-vvv`.
- **Validate at the boundary** with ranges, `ValueEnum`, and custom `value_parser` functions, so that the rest of your program works with data that is already trustworthy.
- **Subcommands are enums.** The compiler checks that every subcommand is handled, and `#[derive(Args)]` lets you share argument groups.
- **Relationships between arguments** (`conflicts_with`, `requires`, `ArgGroup`) belong in the parser, where they produce consistent errors and accurate help.

- **Environment variables** act as a defaults layer, with the precedence of flag, then environment, then default.
- **clap generates more than parsing:** `--help`, `--version`, "did you mean" suggestions, shell completions, and man pages.
- **Parsers are testable.** `debug_assert()` validates your definition and `try_parse_from` tests your behavior without spawning a process.
- You built two runnable tools: **`wcr`**, a stdin-aware word counter, and **`tasks`**, a subcommand-driven task tracker with JSON storage, environment configuration, and completions.

In **Chapter 5**, you will move from the command line to the file system itself: reading directories, walking trees, working with paths, and handling the edge cases that make file operations harder than they look. The `Vec<PathBuf>` arguments you parsed in this chapter will become the starting point for real work.

4.18 Further Reading

- **clap documentation** (docs.rs/clap): the derive tutorial and the derive reference are the best next stops. The reference lists every `#[arg(...)]` and `#[command(...)]` attribute.
- **clap cookbook** (in the repository's `examples/` directory): complete, runnable examples modeled on `git`, `cargo`, and `pacman`.
- **clap_complete** (docs.rs/clap_complete): dynamic completions and the supported shells.
- **clap_mangen** (docs.rs/clap_mangen): generating man pages from your `Command`.
- **Command Line Interface Guidelines** (clig.dev): a practical, language-independent guide to flag naming, help text, output, and error design. Excellent companion reading for this entire Part.
- **POSIX Utility Conventions and the GNU Coding Standards**: the origin of `-x`, `--long`, and `--` conventions that clap implements.
- **The Rust API Guidelines** (rust-lang.github.io/api-guidelines): useful when your CLI also exposes a library crate, as the capstone projects do.