Crate clap

`clap` is a simple-to-use, efficient, and full-featured library for parsing command line arguments and subcommands when writing console/terminal applications.

About

`clap` is used to parse *and validate* the string of command line arguments provided by the user at runtime. You provide the list of valid possibilities, and `clap` handles the rest. This means you focus on your *applications* functionality, and less on the parsing and validating of arguments.

`clap` also provides the traditional version and help switches (or flags) 'for free' meaning automatically with no configuration. It does this by checking list of valid possibilities you supplied and adding only the ones you haven't already defined. If you are using subcommands, `clap` will also auto-generate a `help` subcommand for you in addition to the traditional flags.

Once `clap` parses the user provided string of arguments, it returns the matches along with any applicable values. If the user made an error or typo, `clap` informs them of the mistake and exits gracefully (or returns a `Result` type and allows you to perform any clean up prior to exit). Because of this, you can make reasonable assumptions in your code about the validity of the arguments.

Quick Example

The following examples show a quick example of some of the very basic functionality of `clap`. For more advanced usage, such as requirements, conflicts, groups, multiple values and occurrences see the documentation, examples/ directory of this repository or the video tutorials.

NOTE: All of these examples are functionally the same, but show different styles in which to use `clap`

The first example shows a method that allows more advanced configuration options (not shown in this small example), or even dynamically generating arguments when desired. The downside is it's more verbose.

```
// (Full example with detailed comments in examples/01b_quick_example.rs)
//
// This example demonstrates clap's full 'builder pattern' style of creating arguments whic
// more verbose, but allows easier editing, and at times more advanced options, or the poss
// to generate arguments dynamically.
extern crate clap;
use clap::{Arg, App, SubCommand};

fn main() {
    let matches = App::new("My Super Program")
                          .version("1.0")
                          .author("Kevin K. <kbknapp@gmail.com>")
                          .about("Does awesome things")
                          .arg(Arg::with_name("config")
                               .short("c")
                               .long("config")
                               .value_name("FILE")
                               .help("Sets a custom config file")
                               .takes_value(true))
                          .arg(Arg::with_name("INPUT")
                               .help("Sets the input file to use")
                               .required(true)
                               .index(1))
```

```
                            .arg(Arg::with_name("v")
                                 .short("v")
                                 .multiple(true)
                                 .help("Sets the level of verbosity"))
                            .subcommand(SubCommand::with_name("test")
                                        .about("controls testing features")
                                        .version("1.3")
                                        .author("Someone E. <someone_else@other.com>")
                                        .arg(Arg::with_name("debug")
                                            .short("d")
                                            .help("print debug information verbosely")))
                            .get_matches();

    // Gets a value for config if supplied by user, or defaults to "default.conf"
    let config = matches.value_of("config").unwrap_or("default.conf");
    println!("Value for config: {}", config);

    // Calling .unwrap() is safe here because "INPUT" is required (if "INPUT" wasn't
    // required we could have used an 'if let' to conditionally get the value)
    println!("Using input file: {}", matches.value_of("INPUT").unwrap());

    // Vary the output based on how many times the user used the "verbose" flag
    // (i.e. 'myprog -v -v -v' or 'myprog -vvv' vs 'myprog -v'
    match matches.occurrences_of("v") {
        0 => println!("No verbose info"),
        1 => println!("Some verbose info"),
        2 => println!("Tons of verbose info"),
        3 | _ => println!("Don't be crazy"),
    }

    // You can handle information about subcommands by requesting their matches by name
    // (as below), requesting just the name used, or both at the same time
    if let Some(matches) = matches.subcommand_matches("test") {
        if matches.is_present("debug") {
            println!("Printing debug info...");
        } else {
            println!("Printing normally...");
        }
    }

    // more program logic goes here...
}
```

The next example shows a far less verbose method, but sacrifices some of the advanced configuration options (not shown in this small example). This method also takes a *very* minor runtime penalty.

```
// (Full example with detailed comments in examples/01a_quick_example.rs)
//
// This example demonstrates clap's "usage strings" method of creating arguments
// which is less verbose
extern crate clap;
use clap::{Arg, App, SubCommand};

fn main() {
    let matches = App::new("myapp")
                            .version("1.0")
                            .author("Kevin K. <kbknapp@gmail.com>")
                            .about("Does awesome things")
                            .args_from_usage(
```

```
                                "-c, --config=[FILE] 'Sets a custom config file'
                                <INPUT>              'Sets the input file to use'
                                -v...                'Sets the level of verbosity'")
                            .subcommand(SubCommand::with_name("test")
                                        .about("controls testing features")
                                        .version("1.3")
                                        .author("Someone E. <someone_else@other.com>")
                                        .arg_from_usage("-d, --debug 'Print debug information
                            .get_matches();

    // Same as previous example...
}
```

This third method shows how you can use a YAML file to build your CLI and keep your Rust source tidy or support multiple localized translations by having different YAML files for each localization.

First, create the `cli.yml` file to hold your CLI options, but it could be called anything we like:

```
name: myapp
version: "1.0"
author: Kevin K. <kbknapp@gmail.com>
about: Does awesome things
args:
    - config:
        short: c
        long: config
        value_name: FILE
        help: Sets a custom config file
        takes_value: true
    - INPUT:
        help: Sets the input file to use
        required: true
        index: 1
    - verbose:
        short: v
        multiple: true
        help: Sets the level of verbosity
subcommands:
    - test:
        about: controls testing features
        version: "1.3"
        author: Someone E. <someone_else@other.com>
        args:
            - debug:
                short: d
                help: print debug information
```

Since this feature requires additional dependencies that not everyone may want, it is *not* compiled in by default and we need to enable a feature flag in Cargo.toml:

Simply change your `clap = "~2.27.0"` to `clap = {version = "~2.27.0", features = ["yaml"]}`.

At last we create our `main.rs` file just like we would have with the previous two examples:

```
// (Full example with detailed comments in examples/17_yaml.rs)
//
// This example demonstrates clap's building from YAML style of creating arguments which i
// more clean, but takes a very small performance hit compared to the other two methods.
#[macro_use]
extern crate clap;
```

```
use clap::App;

fn main() {
    // The YAML file is found relative to the current file, similar to how modules are fou
    let yaml = load_yaml!("cli.yml");
    let matches = App::from_yaml(yaml).get_matches();

    // Same as previous examples...
}
```

Finally there is a macro version, which is like a hybrid approach offering the speed of the builder pattern (the first example), but without all the verbosity.

```
#[macro_use]
extern crate clap;

fn main() {
    let matches = clap_app!(myapp =>
        (version: "1.0")
        (author: "Kevin K. <kbknapp@gmail.com>")
        (about: "Does awesome things")
        (@arg CONFIG: -c --config +takes_value "Sets a custom config file")
        (@arg INPUT: +required "Sets the input file to use")
        (@arg debug: -d ... "Sets the level of debugging information")
        (@subcommand test =>
            (about: "controls testing features")
            (version: "1.3")
            (author: "Someone E. <someone_else@other.com>")
            (@arg verbose: -v --verbose "Print test information verbosely")
        )
    ).get_matches();

    // Same as before...
}
```

If you were to compile any of the above programs and run them with the flag `--help` or `-h` (or `help` subcommand, since we defined `test` as a subcommand) the following would be output

```
$ myprog --help
My Super Program 1.0
Kevin K. <kbknapp@gmail.com>
Does awesome things

USAGE:
    MyApp [FLAGS] [OPTIONS] <INPUT> [SUBCOMMAND]

FLAGS:
    -h, --help       Prints this message
    -v               Sets the level of verbosity
    -V, --version    Prints version information

OPTIONS:
    -c, --config <FILE>    Sets a custom config file

ARGS:
    INPUT    The input file to use

SUBCOMMANDS:
```

```
    help    Prints this message
    test    Controls testing features
```

NOTE: You could also run `myapp test --help` to see similar output and options for the `test` subcommand.

Try it!

Pre-Built Test

To try out the pre-built example, use the following steps:

- Clone the repository `$ git clone https://github.com/kbknapp/clap-rs && cd clap-rs/tests`
- Compile the example `$ cargo build --release`
- Run the help info `$ ./target/release/claptests --help`
- Play with the arguments!

BYOB (Build Your Own Binary)

To test out `clap`'s default auto-generated help/version follow these steps:

- Create a new cargo project `$ cargo new fake --bin && cd fake`
- Add `clap` to your `Cargo.toml`

```
[dependencies]
clap = "2"
```

- Add the following to your `src/main.rs`

```
extern crate clap;
use clap::App;

fn main() {
  App::new("fake").version("v1.0-beta").get_matches();
}
```

- Build your program `$ cargo build --release`
- Run with help or version `$ ./target/release/fake --help` or `$ ./target/release/fake --version`

Usage

For full usage, add `clap` as a dependency in your `Cargo.toml` (it is **highly** recommended to use the `~major.minor.patch` style versions in your `Cargo.toml`, for more information see Compatibility Policy) to use from crates.io:

```
[dependencies]
clap = "~2.27.0"
```

Or get the latest changes from the master branch at github:

```
[dependencies.clap]
git = "https://github.com/kbknapp/clap-rs.git"
```

Add `extern crate clap;` to your crate root.

Define a list of valid arguments for your program (see the documentation or examples/ directory of this repo)

Then run `cargo build` or `cargo update && cargo build` for your project.

Optional Dependencies / Features

Features enabled by default

- `suggestions`: Turns on the `Did you mean '--myoption'?` feature for when users make typos. (builds dependency `strsim`)
- `color`: Turns on colored error messages. This feature only works on non-Windows OSs. (builds dependency `ansi-term` and `atty`)
- `wrap_help`: Wraps the help at the actual terminal width when available, instead of 120 characters. (builds dependency `textwrap` with feature `term_size`)

To disable these, add this to your `Cargo.toml`:

```
[dependencies.clap]
version = "~2.27.0"
default-features = false
```

You can also selectively enable only the features you'd like to include, by adding:

```
[dependencies.clap]
version = "~2.27.0"
default-features = false

# Cherry-pick the features you'd like to use
features = [ "suggestions", "color" ]
```

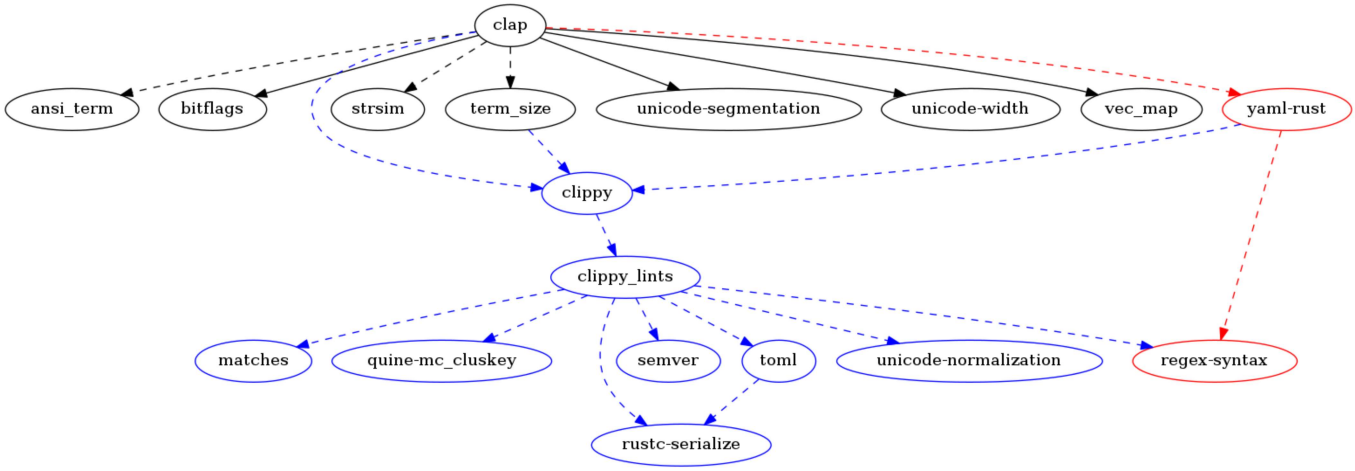
Opt-in features

- **"yaml"**: Enables building CLIs from YAML documents. (builds dependency `yaml-rust`)
- **"unstable"**: Enables unstable `clap` features that may change from release to release

Dependencies Tree

The following graphic depicts `clap`s dependency graph (generated using cargo-graph).

- **Dashed** Line: Optional dependency
- **Red** Color: **NOT** included by default (must use cargo `features` to enable)
- **Blue** Color: Dev dependency, only used while developing.



More Information

You can find complete documentation on the docs.rs for this project.

You can also find usage examples in the examples/ directory of this repo.

Video Tutorials

There's also the video tutorial series Argument Parsing with Rust v2.

These videos slowly trickle out as I finish them and currently a work in progress.

How to Contribute

Contributions are always welcome! And there is a multitude of ways in which you can help depending on what you like to do, or are good at. Anything from documentation, code cleanup, issue completion, new features, you name it, even filing issues is contributing and greatly appreciated!

Another really great way to help is if you find an interesting, or helpful way in which to use `clap`. You can either add it to the examples/ directory, or file an issue and tell me. I'm all about giving credit where credit is due :)

Please read CONTRIBUTING.md before you start contributing.

Testing Code

To test with all features both enabled and disabled, you can run theese commands:

```
$ cargo test --no-default-features
$ cargo test --features "yaml unstable"
```

Alternatively, if you have `just` installed you can run the prebuilt recipes. *Not* using `just` is perfectly fine as well, it simply bundles commands automatically.

For example, to test the code, as above simply run:

```
$ just run-tests
```

From here on, I will list the appropriate `cargo` command as well as the `just` command.

Sometimes it's helpful to only run a subset of the tests, which can be done via:

```
$ cargo test --test <test_name>

# Or

$ just run-test <test_name>
```

Linting Code

During the CI process `clap` runs against many different lints using `clippy`. In order to check if these lints pass on your own computer prior to submitting a PR you'll need a nightly compiler.

In order to check the code for lints run either:

```
$ rustup override add nightly
$ cargo build --features lints
$ rustup override remove

# Or

$ just lint
```

Debugging Code

Another helpful technique is to see the `clap` debug output while developing features. In order to see the debug output while running the full test suite or individual tests, run:

```
$ cargo test --features debug

# Or for individual tests
$ cargo test --test <test_name> --features debug
```

```
# The corresponding just command for individual debugging tests is:
$ just debug <test_name>
```

Goals

There are a few goals of `clap` that I'd like to maintain throughout contributions. If your proposed changes break, or go against any of these goals we'll discuss the changes further before merging (but will *not* be ignored, all contributes are welcome!). These are by no means hard-and-fast rules, as I'm no expert and break them myself from time to time (even if by mistake or ignorance).

- Remain backwards compatible when possible
  - If backwards compatibility *must* be broken, use deprecation warnings if at all possible before removing legacy code - This does not apply for security concerns
- Parse arguments quickly
  - Parsing of arguments shouldn't slow down usage of the main program - This is also true of generating help and usage information (although *slightly* less stringent, as the program is about to exit)
- Try to be cognizant of memory usage
  - Once parsing is complete, the memory footprint of `clap` should be low since the main program is the star of the show
- `panic!` on *developer* error, exit gracefully on *end-user* error

Compatibility Policy

Because `clap` takes `SemVer` and compatibility seriously, this is the official policy regarding breaking changes and previous versions of Rust.

`clap` will pin the minimum required version of Rust to the CI builds. Bumping the minimum version of Rust is considered a minor breaking change, meaning *at a minimum* the minor version of `clap` will be bumped.

In order to keep from being surprised by breaking changes, it is **highly** recommended to use the `~major.minor.patch` style in your `Cargo.toml`:

```
[dependencies] clap = "~2.27.0"
```

This will cause *only* the patch version to be updated upon a `cargo update` call, and therefore cannot break due to new features, or bumped minimum versions of Rust.

Minimum Version of Rust

`clap` will officially support current stable Rust, minus two releases, but may work with prior releases as well. For example, current stable Rust at the time of this writing is 1.21.0, meaning `clap` is guaranteed to compile with 1.19.0 and beyond. At the 1.22.0 release, `clap` will be guaranteed to compile with 1.20.0 and beyond, etc.

Upon bumping the minimum version of Rust (assuming it's within the stable-2 range), it *must* be clearly annotated in the `CHANGELOG.md`

License

`clap` is licensed under the MIT license. Please read the LICENSE-MIT file in this repository for more information.

Macros

_clap_count_exprs	Counts the number of comma-delimited expressions passed to it. The result is a compile-time evaluable expression, suitable for use as a static array size, or the value of a `const`.
app_from_crate	Allows you to build the `App` instance from your Cargo.toml at compile time.
arg_enum	Convenience macro to generate more complete enums with variants to be used as a type when parsing arguments. This enum also provides a `variants()` function which can be used to retrieve a `Vec<&'static str>` of the variant names, as well as implementing `FromStr` and `Display` automatically.

clap_app	Build `App`, `Arg`s, `SubCommand`s and `Group`s with Usage-string like input but without the associated parsing runtime cost.
crate_authors	Allows you to pull the authors for the app from your Cargo.toml at compile time in the form: `"author1 lastname <author1@example.com>:author2 lastname <author2@example.com>"`
crate_description	Allows you to pull the description from your Cargo.toml at compile time.
crate_name	Allows you to pull the name from your Cargo.toml at compile time.
crate_version	Allows you to pull the version from your Cargo.toml at compile time as `MAJOR.MINOR.PATCH_PKGVERSION_PRE`
load_yaml	A convenience macro for loading the YAML file at compile time (relative to the current file, like modules work). That YAML object can then be passed to this function.
value_t	Convenience macro getting a typed value `T` where `T` implements `std::str::FromStr` from an argument value. This macro returns a `Result<T,String>` which allows you as the developer to decide what you'd like to do on a failed parse. There are two types of errors, parse failures and those where the argument wasn't present (such as a non-required argument). You can use it to get a single value, or a iterator as with the `ArgMatches::values_of`
value_t_or_exit	Convenience macro getting a typed value `T` where `T` implements `std::str::FromStr` or exiting upon error, instead of returning a `Result` type.
values_t	Convenience macro getting a typed value `Vec<T>` where `T` implements `std::str::FromStr` This macro returns a `clap::Result<Vec<T>>` which allows you as the developer to decide what you'd like to do on a failed parse.
values_t_or_exit	Convenience macro getting a typed value `Vec<T>` where `T` implements `std::str::FromStr` or exiting upon error.

Structs

App	Used to create a representation of a command line program and all possible command line arguments. Application settings are set using the "builder pattern" with the `App::get_matches` family of methods being the terminal methods that starts the runtime-parsing process. These methods then return information about the user supplied arguments (or lack there of).
Arg	The abstract representation of a command line argument. Used to set all the options and relationships that define a valid argument for the program.
ArgGroup	`ArgGroup`s are a family of related arguments and way for you to express, "Any of these arguments". By placing arguments in a logical group, you can create easier requirement and exclusion rules instead of having to list each argument individually, or when you want a rule to apply "any but not all" arguments.
ArgMatches	Used to get information about the arguments that where supplied to the program at runtime by the user. New instances of this struct are obtained by using the `App::get_matches` family of methods.
Error	Command Line Argument Parser Error
OsValues	An iterator for getting multiple values out of an argument via the [`ArgMatches::values_of_os`] method. Usage of this iterator allows values which contain invalid UTF-8 code points unlike [`Values`].
SubCommand	The abstract representation of a command line subcommand.
Values	An iterator for getting multiple values out of an argument via the `ArgMatches::values_of` method.
YamlLoader	

Enums

AppSettings	Application level settings, which affect how `App` operates
ArgSettings	Various settings that apply to arguments and may be set, unset, and checked via getter/setter methods `Arg::set`, `Arg::unset`, and `Arg::is_set`
ErrorKind	Command line argument parser kind of error
Shell	Describes which shell to produce a completions file for

Type Definitions

Result	Short hand for `Result` type