PNA Rust Project 1: The Rust toolbox

Task: Create an in-memory key/value store that passes simple tests and responds to command-line arguments.

Goals:

- Install the Rust compiler and tools
- Learn the project structure used throughout this course
- Use `cargo init` / `run` / `test` / `clippy` / `fmt`
- Learn how to find and import crates from crates.io
- Define an appropriate data type for a key-value store

Topics: testing, the `clap` crate, `CARGO_VERSION` etc., the `clippy` and `rustfmt` tools.

Extensions: the `structopt` crate.

Introduction

In this project you will create a simple in-memory key/value store that maps strings to strings, and that passes some tests and responds to command line arguments. The focus of this project is on the tooling and setup that goes into a typical Rust project.

If this sounds basic to you, please do the project anyway as it discusses some general patterns that will be used throughout the course.

Project spec

The cargo project, `kvs`, builds a command-line key-value store client called `kvs`, which in turn calls into a library called `kvs`.

The `kvs` executable supports the following command line arguments:

- `kvs set <KEY> <VALUE>`

  Set the value of a string key to a string
- `kvs get <KEY>`

  Get the string value of a given string key
- `kvs rm <KEY>`

  Remove a given key
- `kvs -V`

  Print the version

The `kvs` library contains a type, `KvStore`, that supports the following methods:

- `KvStore::set(&mut self, key: String, value: String)`

  Set the value of a string key to a string
- `KvStore::get(&mut self, key: String) -> Option<String>`

  Get the string value of the a string key. If the key does not exist, return `None`.
- `KvStore::remove(&mut self, key: String)`

  Remove a given key.

The `KvStore` type stores values in-memory, and thus the command-line client can do little more than print the version. The `get`/ `set` / `rm` commands will return an "unimplemented" error when run from the command line. Future projects will store values on disk and have a working command line interface.

Installation

At this point in your Rust programming experience you should know how to install Rust via [rustup].

If you haven't already, do so now by running

```
curl https://sh.rustup.rs -sSf | sh
```

(If you are running Windows then follow the instructions on rustup.rs. Note though that you will face more challenges than others during this course, as it was developed on Unix. In general, Rust development on Windows is as less polished experience than on Unix).

Verify that the toolchain works by typing `rustc -V`. If that doesn't work, log out and log in again so that changes to the login profile made during installation can take effect.

Project setup

You will do the work for this project in your own git repository, with your own Cargo project. You will import the test cases for the project from the source repository for this course.

Note that within that repository, all content related to this course is within the `rust` subdirectory. You may ignore any other directories.

The projects in this course contain both libraries and executables. They are executables because we are developing an application that can be run. They are libraries because the supplied test cases must link to them.

We'll use the same setup for each project in this course.

The directory layout we will use is:

```
├── Cargo.toml
├── src
│   ├── bin
│   │   └── kvs.rs
│   └── lib.rs
└── tests
    └── tests.rs
```

The `Cargo.toml`, `lib.rs` and `kvs.rs` files look as follows:

`Cargo.toml`:

```toml
[package]
name = "kvs"
version = "0.1.0"
authors = ["Brian Anderson <andersrb@gmail.com>"]
description = "A key-value store"
edition = "2018"
```

`lib.rs`:

```rust
// just leave it empty for now
```

`kvs.rs`:

```
fn main() {
    println!("Hello, world!");
}
```

The author should be yourself, but the name needs to be `kvs` in order for the test cases to work. That's because the project name is also the name of the library it contains. Likewise the name of the binary (the command line application) needs to be `kvs`. In the above setup it will be `kvs` implicitly based on the file name, but you could name the file whatever you wanted by putting the appropriate information in the manifest (`Cargo.toml`).

You may set up this project with `cargo new --lib`, `cargo init --lib` (in a clean directory), or manually. You'll probably also want to initialize a git repository in the same directory.

Finally, the `tests` directory is copied from the course materials. In this case, copy from the course repository the file `rust/projects/project-1/tests` into your own repository, as `tests`.

At this point you should be able to run the program with `cargo run`.

Try it now.

You are set up for this project and ready to start hacking.

Part 1: Make the tests compile

You've been provided with a suite of unit tests in `tests/tests.rs`. Open it up and take a look.

Try to run the tests with `cargo test`. What happens? Why?

Your first task for this project is to make the tests *compile*. Fun!

If your project is like mine you probably saw a huge spew of build errors. Look at the first few. In general, when you see a bunch of errors, the first are the most important — `rustc` will keep trying to compile even after hitting errors, so errors can cascade, the later ones being pretty meaningless. Your first few errors probably look like:

```
error[E0433]: failed to resolve: use of undeclared type or module
`assert_cmd`
 --> tests/tests.rs:1:5
  |
1 | use assert_cmd::prelude::*;
  |     ^^^^^^^^^^ use of undeclared type or module `assert_cmd`

error[E0432]: unresolved import
 --> tests/tests.rs:3:5
  |
3 | use predicates::str::contains;
  |     ^^^^^^^^^^^^^^^^^^^^^^^^^
```

(If you are seeing something else, please file an issue).

These two errors are quite hard to diagnose to a new Rust programmer so I'll just tell you what's going on here: you are missing [dev-dependency] crates in your manifest.

For this project your `Cargo.toml` file needs to contain these lines:

```
[dev-dependencies]
assert_cmd = "0.11.0"
predicates = "1.0.0"
```

The details of these dependencies are not important to you completing the project, but you might want to investigate them on your own. We didn't tell you about the need for dev-deps earlier just so you would experience these errors yourself. In future projects, the setup text will tell you the dev-deps you need.

One quick note: how can you figure out that these errors are due to missing dependencies in your manifest and not due to errors in your source code? Here's one big clue, from the error shown previously:

```
1 | use assert_cmd::prelude::*;
  |     ^^^^^^^^^^ use of undeclared type or module `assert_cmd`
```

In `use` statements the first path element is always the name of a crate. The exception to this is when the first path element references a name that was previously brought into scope with *another* `use` statement. In other words, if there had been another `use` statement in this file like `use foo::assert_cmd`, then use `assert_cmd::prelude::*` would refer to *that* `assert_cmd`. There is more that could be said about this but we shouldn't go deeper into the nuances of path resolution here. Just know that, in general, in a `use` statement, if the first element in the path isn't found (i.e. cannot be resolved), the problem is probably that the crate hasn't been named in the manifest.

Whew. That is an unfortunate diversion in the very first project. But hopefully instructive.

Go ahead and add the appropriate dev-deps to your manifest.

Try again to run the tests with `cargo test`. What happens? Why?

Hopefully those *previous* errors are gone. Now the errors are all about the test cases not being able to find all the code it expects in your own code.

So now your task is to outline all the types, methods, etc. necessary to make the tests build.

During this course you will read the test cases a lot. The test cases tell you exactly what is expected of your code. If the text and the tests don't agree, the tests are right (file a bug!). This is true in the real world too. The test cases demonstrate what the software *actually* does. They are reality. Get used to reading test cases.

And, bonus — test cases are often the poorest-written code in any project, sloppy and undocumented.

Again, try to run the tests with `cargo test`. What happens? Why?

In `src/lib.rs` write the type and method definitions necessary to make `cargo test --no-run` complete successfully. Don't write any method bodies yet — instead write `panic!()`. This is the way to sketch out your APIs without knowing or caring about the implementation (there's also the [`unimplemented!`] macro, but since typing it is longer, it's common to simply use `panic!`, a possible exception being if you are releasing software that contains unimplemented methods).

Do that now before moving on.

Once that is done, if you run `cargo test` (without `--no-run`), you should see that some of your tests are failing, like

```
    Finished dev [unoptimized + debuginfo] target(s) in 2.32s
     Running target/debug/deps/kvs-b03a01e7008067f6

running 0 tests

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

     Running target/debug/deps/kvs-a3b5a004932c6715

running 0 tests

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

     Running target/debug/deps/tests-5e1c2e20bd1fa377

running 13 tests
test cli_get ... FAILED
test cli_invalid_get ... FAILED
test cli_invalid_rm ... FAILED
test cli_invalid_set ... FAILED
test cli_no_args ... FAILED
test cli_invalid_subcommand ... FAILED
... more lines of spew ...
```

... followed by many more lines. That's great! That's all we need right now. You'll make those pass throughout the rest of this project.

Aside: Testing tips

If you look again at the output from `cargo test` you'll see something interesting:

```
     Running target/debug/deps/kvs-b03a01e7008067f6

running 0 tests

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

```

```
     Running target/debug/deps/kvs-a3b5a004932c6715

running 0 tests

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

     Running target/debug/deps/tests-5e1c2e20bd1fa377


running 13 tests
test cli_get ... FAILED
```

Cargo says "Running ..." three times. And the first two of those times it in fact did not run any tests. And furthermore, if all those tests hadn't failed, cargo would have run *yet another* set of tests.

Why is this?

Well, it is because there are many places you can write tests in Rust:

- Inside the source of your library
- Inside the source of each of your binaries
- Inside each test file
- In the doc comments of your library

And cargo doesn't know which of these actually contain tests, so it just builds and runs them all.

So those two sets of empty tests:

```
     Running target/debug/deps/kvs-b03a01e7008067f6
running 0 tests
     Running target/debug/deps/kvs-a3b5a004932c6715
running 0 tests
```

Well, this is a bit confusing, but one of them is your library, compiled for testing, and the other is your binary, compiled for testing. Neither contains any tests. The reason both have "kvs" in their names is because both your library and your binary are called "kvs".

All this test spew gets annoying, and there are two ways to quiet cargo: with command line arguments, and with changes to the manifest.

Here are the relevant command line flags:

- `cargo test --lib` — test just the tests inside the library
- `cargo test --doc` — test the doc tests in the library
- `cargo test --bins` — test all the bins in the project
- `cargo test --bin foo` — test just the `foo` bin
- `cargo test --test foo` — test the tests in test file `foo`

These are convenient to quickly hide test spew, but if a project doesn't contain a type of tests it's probably best to just never deal with them. If you recall from The Cargo Book's manifest description, there are two keys that can be applied: `test = false` and `doctest = false`. They go in the `[lib]` and `[[bin]]` sections. Consider updating your manifest.

Another quick thing to do if you haven't before. Run this:

```
cargo test -- --help
```

Just do it. It's cool. What you are seeing there is the help information for *the executable containing your compiled tests* (that `--` surrounded by spaces tells cargo to pass all following arguments to the test binary). It's not the same info displayed when you run `cargo test --help`. They are two different things: cargo is running your test bin by passing it all these various arguments.

If you want you can do exactly the same thing. Let's go back one more time to our `cargo test` example. We saw this line:

```
     Running target/debug/deps/kvs-b03a01e7008067f6
```

That's cargo telling you the name of the test binary. You can run it yourself, like `target/debug/deps/kvs-b03a01e7008067f6 --help`.

The `target` directory contains lots of cool stuff. Digging through it can teach you a lot about what the Rust toolchain is actually doing.

In practice, particularly with large projects, you won't run the entire test suite while developing a single feature. To narrow down the set of tests to the ones we care about, run the following:

```
cargo test cli_no_args
```

This will run the test called `cli_no_args`. In fact, it will run any test containing `cli_no_args` in the name, so if, e.g., you want to run all the CLI tests, you can run `cargo test cli`. That's probably how you will be running the tests yourself as you work through the project, otherwise you will be distracted by the many failing tests that you have not yet fixed. Unfortunately that pattern is a simple substring match, not something fancy like a regular expression.

Note that, as of this writing, the test cases for the projects in this course are not organized in a way that makes it clear which test cases should complete for any particular section of a project — only that by the end the entire suite should pass. You'll need to read the names and implementations of the tests to figure out which you *think* should pass at any particular time.

Part 2: Accept command line arguments

The key / value stores throughout this course are all controlled through a command-line client. In this project the command-line client is very simple because the state of the key-value store is only stored in memory, not persisted to disk.

In this part you will make the `cli_*` test cases pass.

Recall how to run individual test cases from previous sections of

Again, the interface for the CLI is:

- `kvs set <KEY> <VALUE>`

Set the value of a string key to a string

- `kvs get <KEY>`

  Get the string value of a given string key

- `kvs rm <KEY>`

  Remove a given key

- `kvs -V`

  Print the version

In this iteration though, the `get` and `set` commands will print to stderr the string, "unimplemented", and exiting with a non-zero exit code, indicating an error.

You will use the `clap` crate to handle command-line arguments.

Find the latest version of the `clap` crate and add it to your dependencies in `Cargo.toml`. There are a number of ways to find and import a crate, but pro-tip: check out the built-in [`cargo search`] and the plug-in [`cargo edit`].

Next use [crates.io], [lib.rs], or [docs.rs] to find the documentation for the `clap` crate, and implement the command line interface such that the `cli_*` test cases pass.

When you are testing, use `cargo run`; do not run the executable directly from the `target/` directory. When passing arguments to the program, separate them from the `cargo run` command with two dashes, `--`, like `cargo run -- get key1`.

Part 3: Cargo environment variables

When you set up `clap` to parse your command line arguments, you probably set the name, version, authors, and description (if not, do so). This information is redundant w/ values provided in `Cargo.toml`. Cargo sets environment variables that can be accessed through Rust source code, at build time.

Modify your `clap` setup to set these values from standard cargo environment variables.

Part 4: Store values in memory

Now that your command line scaffolding is done, let's turn to the implementation of `KvStore`, and make the remaining test cases pass.

The behavior of `KvStore`'s methods are fully-defined through the test cases themselves — you don't need any further description to complete the code for this project.

Make the remaining test cases pass by implementing methods on `KvStore`.

Part 5: Documentation

You have implemented the project's functionality, but there are still a few more things to do before it is a polished piece of Rust software, ready for contributions or publication.

First, public items should generally have doc comments.

Doc comments are displayed in a crate's API documentation. API documentation can be generated with the command, `cargo doc`, which will render them as HTML to the `target/doc` folder. Note though that `target/doc` folder does not contain an `index.html`. In this project, your crate's documentation will be located at `target/doc/kvs/index.html`. You can launch a web browser at that location with `cargo doc --open`. `cargo doc --open` doesn't always work, e.g. if you are ssh'd into a cloud instance. If it doesn't though the command will print the name of the html file it couldn't open — useful simply for finding the location of your API docs.

Good doc comments do not just repeat the name of the function, nor repeat information gathered from the type signature. They explain why and how one would use a function, what the return value is on both success and failure, error and panic conditions. The library you have written is very simple so the documentation can be simple as well. If you truly cannot think of anything useful to add through doc comments then it can be ok to not add a doc comment (this is a matter of preference). With no doc comments it should be obvious how the type or function is used from the name and type signature alone.

Doc comments contain examples, and those examples can be tested with `cargo test --doc`.

Add `#![deny(missing_docs)]` to the top of `src/lib.rs` to enforce that all public items have doc comments. Then add doc comments to the types and methods in your library. Follow the documentation guidelines. Give each an example and make sure they pass `cargo test --doc`.

Part 6: Ensure good style with `clippy` and `rustfmt`

[`clippy`] and [`rustfmt`] are tools for enforcing common Rust style. `clippy` helps ensure that code uses modern idioms, and prevents patterns that commonly lead to errors. `rustfmt` enforces that code is formatted consistently. It's not necessary right now, but you might click those links and read their documentation. They are both sophisticated tools capable of much more than described below.

Both tools are included in the Rust toolchain, but not installed by default. They can be installed with the following [`rustup`] commands:

```
rustup component add clippy
rustup component add rustfmt
```

Do that now.

Both tools are invoked as cargo subcommands, `clippy` as `cargo clippy` and `rustfmt` as `cargo fmt`. Note that `cargo fmt` modifies your source code, so commit your work before making before running it to avoid accidentally making unwanted changes, after which you can either include the changes as part of the previous commit with `git commit --amend`. Or just commit them as their own formatting commit — it's common to rust both `clippy` and `rustfmt` after a series of commits, e.g. before submitting a pull request.

Run `cargo clippy` against your project and make any suggested changes. Run `cargo fmt` against your project and commit any changes it makes.

It's worth reading the [`rustup`], [`clippy`], and [`rustfmt`] documentation, as these are tools you will be using frequently.

Congratulations, you are done with your first project! If you like you may complete the remaining "extensions". They are optional.

Nice coding, friend. Enjoy a nice break.

<!--

TODO ## Aside: exploring Rust toolchain components

rust component list rust component list | rg -v std # opportunity to introduce rg

-->

Extension 1: `structopt`

In this project we used `clap` to parse command line arguments. It's typical to represent a program's parsed command line arguments as a struct, perhaps named `Config` or `Options`. Doing so requires calling the appropriate methods on `clap`'s `ArgMatches` type. Both steps, for larger programs, require *a lot* of boilerplate code. The `structopt` crate greatly reduces boilerplate by allowing you to define a `Config` struct, annotated to automatically produce a `clap` command line parser that produces that struct. Some find this approach nicer than writing the `clap` code explicitly.

Modify your program to use `structopt` for parsing command line arguments instead of using `clap` directly.

<!--

TODOs

- set the binary's name
- ask about pros / cons of this main.rs setup
  - explain why we're doing this setup (makes main testable) though this will become evident as they work through the tests
- doc comments
- make sure there's enough background reading to support the project
- resources (whether / where to put these?)

    - https://docs.rs/clap/2.32.0/clap/
    - https://doc.rust-lang.org/cargo/reference/environment-variables.html#environment-variables-cargo
    - https://rust-lang-nursery.github.io/api-guidelines/documentation.html#documentation
    - https://doc.rust-lang.org/std/macro.env.html
    - https://github.com/rust-lang/rust-clippy/blob/master/README.md
    - https://github.com/rust-lang/rustfmt/blob/master/README.md
- do range lookups (`scan`)?
- README.md?
- GitHub CI setup?
- Add suggestions to read clippy and rustfmt documentation
- Make clippy / rustfmt docs readings?

-->