Week 04 Weekly Exercises

Objectives

- Introduction to rustdoc, doctests and rust documentation conventions
- Implement standard behaviours (traits) on arbitrary types
- Practice basic use of rust modules, and understand it's goals

Activities To Be Completed

The following is a list of all the markedactivities available to complete this week...

- **Doctor Who?**
- **Vector Operations**
- **Modularity**
- **Assignment One!**

The following practice activities are optional and are not **marked, or required** to be completed for the week.

None - all exercises this week are marked.

Preparation

Before attempting the weekly exercises you should re-read the relevant lecture slides and their accompanying examples.

Getting Started

Create a new directory for this week's exercises called `lab04`, change to this directory, and fetch the provided code for this week by running these commands:

```
$ mkdir lab04
$ cd lab04
$ 6991 fetch lab 04
```

Or, if you're not working on CSE, you can download the provided code as a tar file.

EXERCISE:

Doctor Who?

The standard Rust distribution ships with a command-line tool called `rustdoc`. Its job is to generate documentation for Rust projects. On a fundamental level, `rustdoc` takes as an argument either a crate root or a Markdown file, and produces HTML, CSS, and JavaScript.

Cargo also has integration with rustdoc to make it easier to generate docs. In any of our cargo projects, we can run `6991 cargo doc` to generate the relevant html/css/js for our rust documentation page.

In this exercise, you've been given the solution crate to an optional exercise from week03, an exercised named `my_caesar`. It's been slightly modified to be a library crate.

NOTE:

> Take a look at what changes were made compared to the solution! Why might these changes have been made? What benefits do they give?

Your goal is to finish creating this library crate that provides public functions for the user to use, and to document/test the crate and its functions.

You task is to complete the following:

- Expose the relevant entry function to the user by adding `pub` to one of the functions.
- Document each function with
    - Description of the function. (example).
    - One passing doctest (if the function is public - you cannot doctest private functions, have a think about why not!).
- Add crate level documentation, describing what the crate does.
- Add a line of documentation for each constant.

> NOTE:
>
> Note that we're only asking you to test the **public** interface using doctests.
>
> If this was a real library crate, what other testing mechanisms might you want to use?
>
> Hint - unit tests!

To test your documentation is working correctly, you can run

```
$ 6991 cargo doc
```

And open the file: `target/doc/doctor_who/index.html`
You can test your doctests pass by running

```
$ 6991 cargo test --doc
   Compiling doctor_who v0.1.0 (/doctor_who/)
    Finished test [unoptimized + debuginfo] target(s) in 0.16s
   Doc-tests doctor_who

running 1 tests
test src/lib.rs - caesar_shift (line 8) ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.16s
```

> HINT:
>
> Hint 1
>
> We don't typically need to write paremater documentation for functions, as the compiler types should be sufficient. However, if you want to write parameter documentation, you can do so by adding a line of documentation for each parameter, like so:
>
> ```
> /// # Parameters
> /// * `x` - The first number to add.
> /// * `y` - The second number to add.
> ```

When you think your program is working, you can use `autotest` to run some simple automated tests:

```
$ 6991 autotest
```

When you are finished working on this exercise, you must submit your work by running `give`:

```
$ 6991 give-crate
```

The due date for this exercise is **Week 5 Wednesday 21:00:00**.

Note that this is an individual exercise; the work you submit with `give` must be entirely your own.

EXERCISE:

Vector Operations

NOTE:

Outcomes:

- Understanding that a trait is a set of common behaviours
- Implementing traits on custom structs
- Appreciating where traits might be applied for assignment 1

In this exercise, you will be implementing a set of standard behaviours (the technical word for this is `traits`, but we will go into more detail about those in week 5) on a custom struct, `Vec3`.

`Vec3` is a struct used to represent a 3D Vector. 3D Vectors are commonly used to represent graphics and physics, but in order for them to be useful, we need to be able to do standard mathematical operations on them!

For each of the operations, you can assume that the operation on two vectors, is the same as performing the operations on the discrete components of each vector.

```
$ 6991 cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.00s
     Running `target/debug/vector_operations`
v1 = Vec3 { x: 1.0, y: 2.0, z: 3.0 }
v2 = Vec3 { x: 4.0, y: 5.0, z: 6.0 }
v1 + v2 = Vec3 { x: 5.0, y: 7.0, z: 9.0 }
v1 - v2 = Vec3 { x: -3.0, y: -3.0, z: -3.0 }
v1 * v2 = Vec3 { x: 4.0, y: 10.0, z: 18.0 }
v1 / v2 = Vec3 { x: 0.25, y: 0.4, z: 0.5 }
```

NOTE:

You can assume that you will only be given basic input. i.e. - no division by zero!
This should make your implementation simpler :)

When you think your program is working, you can use `autotest` to run some simple automated tests:

```
$ 6991 autotest
```

When you are finished working on this exercise, you must submit your work by running `give`:

```
$ 6991 give-crate
```

The due date for this exercise is **Week 5 Wednesday 21:00:00**.

Note that this is an individual exercise; the work you submit with `give` must be entirely your own.

EXERCISE:
Modularity

You've been given a simple binary crate `Tribonacci` (a modified solution to a previous weeks exercise) which does the job of both taking user input (as a command-line argument) and subsequently running the tribonacci sequence on said input.

Your task is to take this single crate, and split it up into a library crate (perhaps containing data structures, and utility functions), and a binary crate (that a user can run).

In your binary crate, you can specify a dependancy on the library crate by using a relative cargo dependancy. For example, in your binary crate's `Cargo.toml`

```
my_new_lib_crate = { path = "../my_new_lib_crate" }
```

This exercise will not have autotests (as it's up to your design!) - but there will be discussion on how we seperate libraries and binaries in an upcoming workshop, you can ask for feedback if you wish, or discuss on the forum!

Submissions that compile will obtain full marks - but make sure to attempt this properly, as use of modules, seperate files and seperate crates will be significant for design marks in your first assignment :)

When you are finished working on this exercise, you must submit your work by running `give`:

This exercise cannot be submitted with `6991 give-crate`. Instead, please package your crate(s) (along with all other files intended for submission) up into a tar file named `crate.tar`.

> NOTE:
>
> Before tar'ing, please make sure you run `6991 cargo clean` on all crates you intend to submit first in order to delete any unnecessary build cruft that may push you over the submission size limit.

For example:

```
$ tar cvf crate.tar <path1> <path2> <path3> ...
# e.g.:
$ tar cvf crate.tar ./my_crate1/ ./my_crate2/
```

Finally, submit with the following command:

```
$ give cs6991 lab04_modularity crate.tar
```

The due date for this exercise is **Week 5 Wednesday 21:00:00**.

Note that this is an individual exercise; the work you submit with `give` must be entirely your own.

EXERCISE:

Assignment One!

This week's exercises are a bit less work, because assignment will be released! Use this time to work on starting your assignment, or if it hasnt been released, relax! Best of luck, and have fun :)

Submission

When you are finished each exercise make sure you submit your work by running `give`.

You can run `give` multiple times.

Don't submit any exercises you haven't attempted.

If you are working at home, you may find it more convenient to upload your work via give's web interface.

The due date for this week's exercises is **Week 5 Wednesday 21:00:00**.

You cannot obtain marks by e-mailing your code to tutors or lecturers.

Automarking will be run continuously throughout the term, using test cases different to those `autotest` runs for you. (Hint: do your own testing as well as running `autotest`.)

After automarking is run you can view your results here or by running this command on a CSE machine:

```
$ 6991 classrun -sturec
```
