Packages, Crates, and Modules

Crate

The smallest unit that the Rust compiler will work with. When you run `rustc some_file.rs` the file `some_file.rs` is treated as a crate file.

A crate when being compiled can be compiled into two forms, a binary crate or a library crate. Binary crate are programs that after being compiled are turned into an executable that you can run. Binary crate must have a function called `main` that gets called when the executable is ran.

Library crate don't have a `main` function and they do not get compiled into an executable binary. Instead, they defined functions that are meant to be shared with multiple projects, much like exporting some common functions that you are going to be using in other projects.

Hence in Rust, when you refer to "crates" it is usually library crate, and refer to binary crate as just the executable or binary.

Exporting and importing functionality

We will go through how Rust does it's import and export system via an example, assume we have a directory setup as such:

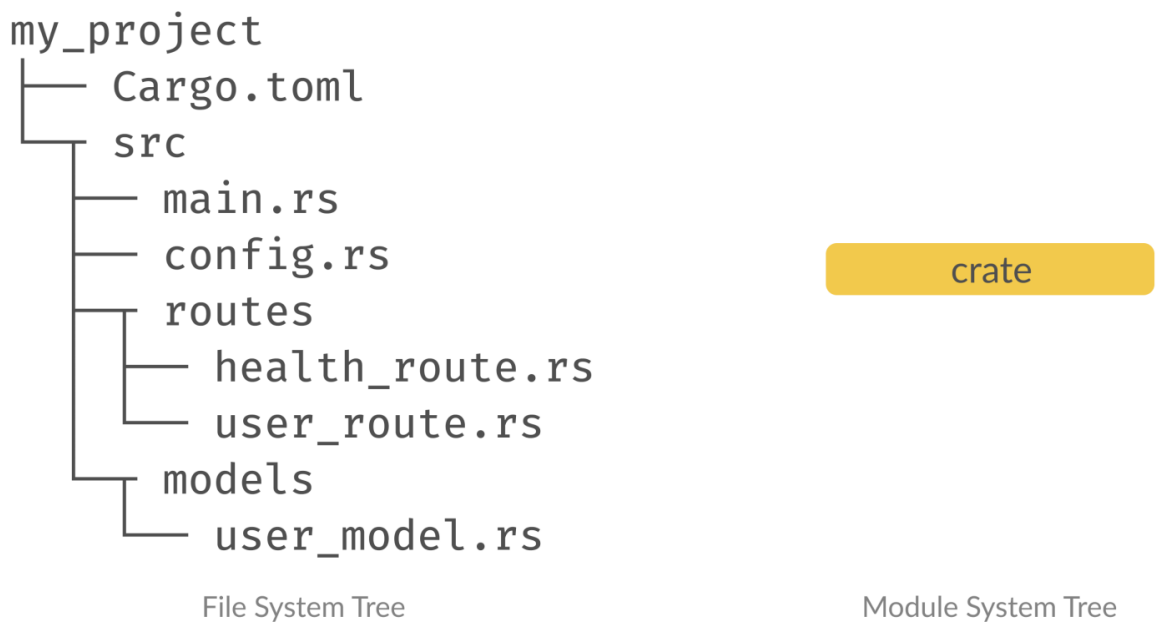
```
my_project
├── Cargo.toml
└─┬ src
  ├── main.rs
  ├── config.rs
  ├─┬ routes
  │ ├── health_route.rs
  │ └── user_route.rs
  └─┬ models
    └── user_model.rs
```

We have functions written in `config.rs, routes/health_route.rs, routes/user_route.rs, and modules/user_module.rs` that we want our `main.rs` use. How do we do that?

Importing `config.rs`

Rust does not build the module tree for you even though the files with functions that you want your `main.rs` to use is under the same directory, Rust by default only sees the crate module which is

`main.rs`.



So what do we do? We will have to explicitly build the module tree in Rust, there is no implicit mapping between the directory tree and the module tree!

In order to add files to the module tree we have to declare that file as a submodule using the `mod` keyword. Where do you declare submodule? Where you are using file, in this case we want to call those functions in `main.rs` hence you will have to declare the submodule in `main.rs` by writing `mod my_module;`.

By writing `mod my_module` the compiler will look for `my_module.rs` or `my_module/mod.rs` in the same directory.

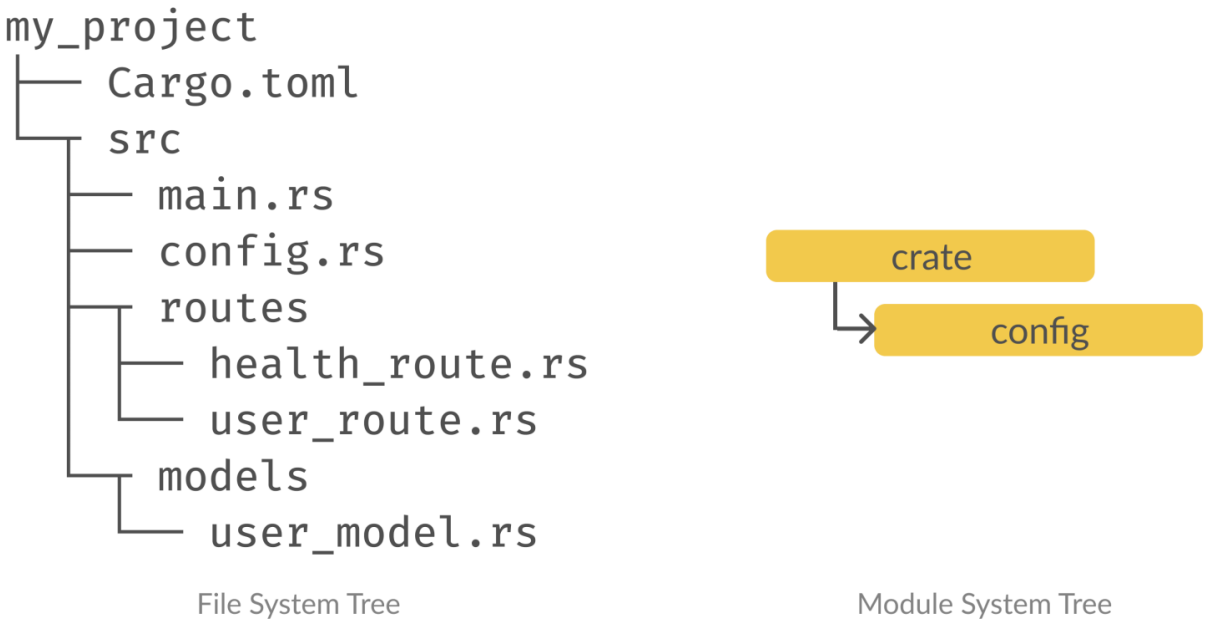
In this case because we are importing `config.rs` which is a file in the same directory as `main.rs` you can just write `mod config;`

```
// main.rs
mod config;

fn main() {
  config::print_config();
  println!("main");
}
```

After you import the module the functions can be called by referring to them using `::` under the submodule namespace.

After you have declare the submodule the module tree looks something like this:



File System Tree

Module System Tree

But wait it still doesn't work?!

After you have successful declare the `config` module, it isn't enough to call the function because almost everything in Rust is private by default. In order to call `print_config` you have to mark it as a public function that other file can call by using the `pub` keyword.

```
// config.rs
pub fn print_config() {
  println!("config");
}
```

Now you will be able to run `main.rs` without a problem.

Importing `routes/health_route.rs`

Now here we are importing another file under another directory the `routes` directory. The `mod` keyword is only for `my_module.rs or my_module/mod.rs` in the same directory. In order to call functions inside `routes/health_route.rs` from `main.rs` here are the things we need to do

1. Make a file named `routes/mod.rs`
2. Declare the `routes` submodule in `main.rs`, this will import the file `routes/mod.rs`
3. Then in `routes/mod.rs` we will declare the submodule `health_route` and make it public by prefixing it with `pub` keyword
4. Then in addition we also have to make the function inside `health_routes.rs` public as well and we are finally done

```
my_project
├── Cargo.toml
```

```
└┬ src
  ├── main.rs
  ├── config.rs
  ├┬ routes
  │ ├── mod.rs
  │ ├── health_route.rs
  │ └── user_route.rs
  └┬ models
    └── user_model.rs
```

```
// main.rs
mod config;
mod routes;

fn main() {
  routes::health_route::print_health_route();
  config::print_config();
  println!("main");
}
```

```
// routes/mod.rs
pub mod health_route;
```

```
// routes/health_route.rs
pub fn print_health_route() {
  println!("health_route");
}
```

> The idea is that if you are going to declare a submodule under another directory, you will import a submodule that has the same directory name. i.e. `another_directory/mod.rs`, and inside that file you will declare the public submodule that you are declaring. Finally make the function of the nested submodule you want to export public as well.

When you call it, you will have to go by the submodule names you have set up including the directory name submodule.

Revision #4
Created 2023-01-29 23:12:42 UTC by Tamarine
Updated 2023-07-27 01:54:13 UTC by Tamarine