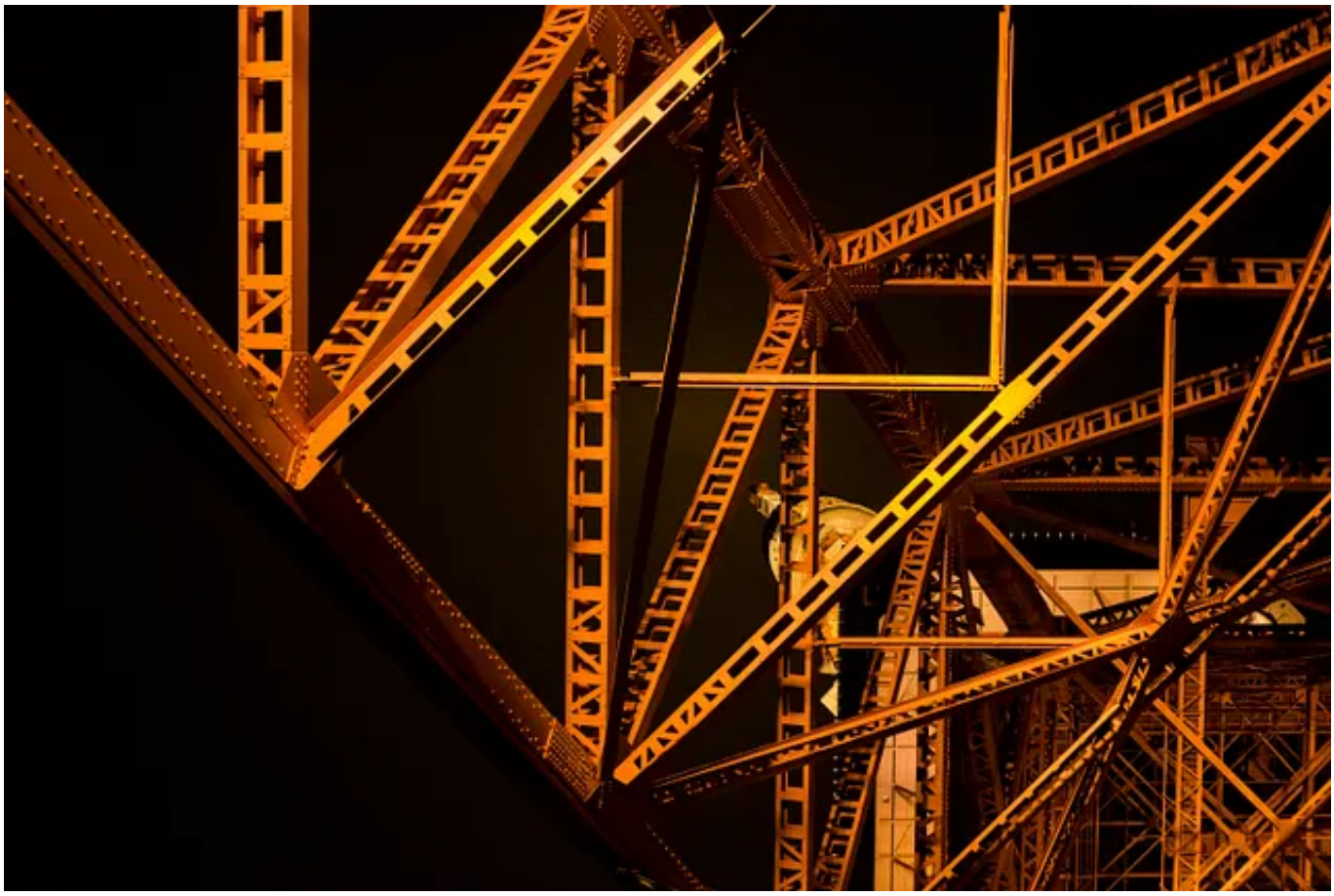

Photo by Alain Pham on Unsplash

Rust — Modules and Project Structure

Exploring the structure of a Rust project, crates, modules, visibility and what the heck is a prelude!?

Gian Lorenzetto, PhD · Follow

Published in CodeX · 10 min read · Sep 15, 2021

In the first post of this series I discussed getting Rust installed and creating new projects using the *cargo* cli tool. In this post I want to get into a bit more detail about the structure of a Rust project, and dig into the concept of crates, modules and preludes.

If you haven't, go get Rust installed and make sure you can create a new project —

```
$ cargo new hello_rust
```

As a reminder, this will create a new *binary application*, so you can run this at a terminal with —

```
$ cargo run
```

You should see cargo first compile and then run your application, with the following written to the console —

```
$ cargo run
“Hello, World!”
```

Great! In the rest of this article, I'm going to discuss —

- The default Rust project structure
- The *main.rs* file

- Rust *modules* (based on files)
- Rust modules and *visibility*
- Rust modules (based on folders)
- What's a *Prelude*?

First up, let's unpack what you've got in the default project.

The Default Rust Project

The default Rust console application is pretty basic, but the folder structure is *intentional* and should not be changed —

```
hello_rust
  - src
    - main.rs
  - .gitignore
  - Cargo.toml
```

Note you can use the `cargo check` command to validate your folder structure and *Cargo.toml* file at any time. If you do make a mistake (in this case I renamed `src` to `src1`), Cargo will helpfully tell you what you need to do —

```
error: failed to parse manifest at
`/Users/gian/_working/scratch/hello_rust/Cargo.toml`

Caused by:
 no targets specified in the manifest
```

```
 either src/lib.rs, src/main.rs, a [lib] section, or [[bin]] section
must be present
```

In our case we must have a `src/main.rs`, since we created a *binary application*. If we had created a new library (passing `--lib` to the `cargo new` command), then cargo would have created the `src/lib.rs` for us instead.

The *Cargo.lock* file is an automatically generated file and should not be edited. Since Cargo initialises a Git repo for you by default, it also includes a *.gitignore*, with one entry —

```
/target
```

The `target` folder is automatically created on `cargo build` and contains the build *artefacts*, in a *debug* or *release* folder (depending on the build configuration, recall that the default is *debug*).

If you're cross-compiling to another platform, then you will see an additional level specifying the target platform, then the build configuration.

Lastly, there is the main.rs file, which is the entry point for our application. Let's take a close look at it's contents.

The main.rs file

The default *main.rs* file is quite straight forward —

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

We have the `main()` function, the main entry point for our application, which just prints “Helo, World!” to standard output.

You may have noted the `!` in `println!` — this indicates that the `println` function is a Rust macro (an advanced Rust syntax feature) that you can safely ignore for the most part, other than to remember that it’s not a regular function.

While you could now happily write all your Rust code in the *main.rs* file, that’s generally not ideal ;) That’s where modules come in!

Modules

Let’s start off by adding a struct to the *main.rs*. We’ll progressively move this code further from the main file, but for now just change your *main.rs* to look like —

```
struct MyStruct {}

fn main() {
  let _ms = MyStruct {};    <-- Note the '_'
}
```

This is about as simple a program as you could possible write, but it will do nicely to illustrate Rust's modules. Note the `_` prefixing the variable name — Rust doesn't like unused variables (rightly so!) but by using the `_` prefix we're telling the compiler this was intentional and it will prevent the compiler emitting a warning. This is *not* a good example of when to use this feature ("ignored" pattern match), but it does have legitimate uses in other cases.

Now, let's say our code is getting out of hand and we want to move our very complex structure out into another file. We want our code to be *loosely coupled* and *highly cohesive* of course! So let's do that and create a new file called *my_struct.rs* —

```
hello_rust
  - src
    - main.rs
    - my_struct.rs
```

Note that we *must* add the file below the `src/` folder for the compiler to find it. While the name of file doesn't really matter, it's idiomatic Rust to use *snake_case* so that's what we'll do here.

Move the struct declaration from *main.rs* and place it into *my_struct.rs* —

```
// Contents of my_struct.rs
struct MyStruct {}
```

Try building the project —

```
$ cargo build
```

If you removed the structure declaration from *main.rs* you will see an error like this —

```
Compiling hello_rust v0.1.0 (/scratch/hello_rust)
error[E0422]: cannot find struct, variant or union type `MyStruct` in
this scope
 → src/main.rs:2:15
  |
2 | let _ms = MyStruct {};
  |           ^^^^^^^^ not found in this scope
error: aborting due to previous error

For more information about this error, try `rustc — explain E0422`.
error: could not compile `hello_rust`
```

Rust is telling us that it can no longer find the definition of our struct. This is where *modules* come in — unlike some other languages, you must *explicitly include code* into your application. Rust will not simply find the file and compile / include it for you.

In order to include the structure declaration we need to update our *main.rs* to add a *module* reference, like so—

In Rust, all files and folders are *modules*. In order to use the code in a module, you need to first *import* it with the `mod` syntax. Essentially this is

inserting the code from the module at the point where the `mod my_struct;` statement is found. More on folder modules in a bit.

Try building again. Wait, what's this!? It still doesn't work … hmm. Let's take a look at the error message —

```
Compiling hello_rust v0.1.0 (/scratch/hello_rust)
error[E0422]: cannot find struct, variant or union type `MyStruct` in
this scope
 → src/main.rs:4:15
  |
4 | let _ms = MyStruct {};
  |           ^^^^^^^^ not found in this scope
  |
help: consider importing this struct
  |
1 | use crate::my_struct::MyStruct;
  |
```

Although the error is the same, there is now a helpful hint about adding —

```
use crate::my_struct::MyStruct;
```

Let's give that a shot — change *main.rs* to look like this (but *don't* build yet! Spoiler, we have another issue I'll get to shortly)—

```
mod my_struct;
use crate::my_struct::MyStruct;

fn main() {
  let _ms = MyStruct {};
}
```

There's a little bit to unpack here. When you import a module with the `mod` statement, Rust *automatically* creates a module namespace for it (to avoid conflicts) and thus we cannot access our struct type directly. The module namespace is automatically taken from the file name (since the module is a file in this case), hence the `my_struct::MyStruct;` part of the `use` statement — it comes firstly from the file name *my_struct.rs* (without the file extension).

The reason for the `crate::` part of the `use` statement is that *all* Rust projects are crates. As you have now seen, Rust projects can be composed of multiple files (which are *modules*), that can be nested within folders (which are *also modules*). In order to access the root of that module tree, you can always use the `crate::` prefix.

So looking at our *main.rs* again, we have —

```
mod my_struct;                    <-- Import the module code, placing
                                      it into the 'my_struct'
                                      namespace

use crate::my_struct::MyStruct; <-- Map the fully qualified (from
                                      the crate root) struct
                                      declaration to just 'MyStruct'

fn main() {
  let _ms = MyStruct {};          <-- Yay, we found it! .. or did we?
}
```

If it seems confusing (and I must say I found this a little confusing coming from C#) just remember this —

- You *must* use `mod` to include a module (file or folder) into your application.

- The `use` keyword is a convenience to map a fully qualified type name to just it's type name (you can even rename types, but that's for another post).

Modules — Visibility

If you were impatient (go on, admit it!) then you would have tried to build the previous incarnation of *main.rs* and got another error —

```
Compiling hello_rust v0.1.0 (/scratch/hello_rust)
error[E0603]: struct `MyStruct` is private
 → src/main.rs:2:23
  |
2 | use crate::my_struct::MyStruct;
  |                       ^^^^^^^^ private struct
  |
```

This is telling us that although we've found the struct declaration, the visibility of the module is private and thus we can't access it here.

Visibility in Rust is a little different to languages like C#, but it pays to remember a couple of rules —

- Everything *inside* a module (ie, a file or subfolder within the `/src` folder) can access *anything else* within that module.
- Everything *outside* a module can *only* access public members of that module.

This may look strange at first, but it has some very appealing side effects — private functions within a module are still accessible for tests within that module (idiomatic Rust keeps unit tests within the module). Second, every module is forced to declare a public interface, defining what members are accessible outside the module.

To make a member of a module public, we must add the `pub` keyword. Let's revisit our *my_struct.rs* file again and replace the contents with —

```
pub struct MyStruct {}          <-- Add the 'pub' keyword
```

And that's it! You can now successfully build our marvellously complicated application :) Note that you can place `pub` on most declarations, including structs, struct fields, functions (associated and otherwise), constants etc.

Modules — Folders

Now let's say that our `MyStruct` structure is getting out of hand and we want to split it into multiple files. We want to collect these up into a folder to keep things nice and tidy of course.

As alluded to above, Rust treats files and folders in the same way (as modules) with one key difference.

Let's start by creating a folder called `foo/` because we've realised our `MyStruct` is really part of the foo feature of our app. Next move the file *my_struct.rs* into `/src/foo`. Ie, the new folder structure should look like —

```
- src/
  - main.rs
  - foo/
    - my_struct.rs
```

Now edit *main.rs* to include our new module `foo` replacing `my_struct` —

```
mod foo;                    <-- Change the module to match the folder
use crate::foo::MyStruct;   <-- Update the namespace to 'foo'

fn main() {
  let _ms = MyStruct {};
}
```

We can build this now (`cargo build`), but we will get an error. As always, Rust's error messages are instructive —

```
Compiling hello_rust v0.1.0 (/scratch/hello_rust)
error[E0583]: file not found for module `foo`
 → src/main.rs:1:1
  |
1 | mod foo;
  | ^^^^^^^^
  |
  = help: to create the module `foo`, create file "src/foo.rs" or
"src/foo/mod.rs"
```

When trying to import a module defined as a folder, we use the folder name (as we did for the file based module previously) but Rust expects a file named *mod.rs* exists within the folder.

In this case we can simply rename our *my_struct.rs* to *mod.rs* and voila! Our application is building again.

For completeness let's add a file to the `foo/` folder with another struct definition (imaginatively named Another) —

```
// Contents of src/foo/another.rs

pub struct Another {}   <-- We're going to expose this as public
                            from the 'foo' module so that we can
                            use it in main.rs
```

We import our new module into the *mod.rs* file —

```
// Contents of src/foo/mod.rs

pub mod another;        <-- Add the module import for 'another'
                            Note the use of 'pub' to expose the
                            module 'another' as public from the
                            module 'foo'
pub struct MyStruct {}
```

And finally try using our new Another struct in main.rs

```
mod foo;
use crate::foo::MyStruct;
use crate::foo::another::Another; <-- Note that 'another' is a
                                      module within 'foo'

fn main() {
  let _ms = MyStruct {};
  let _a = Another {};            <-- Using prefix '_' as before
}
```

If this looks a little cumbersome, that's because it is. There is however, a better way.

Preludes

Let's revisit our *mod.rs* file within the `foo/` folder. Change the contents to the following —

```
mod another;                <-- Remove the 'pub' modifier
pub use another::Another; <-- Add a use'ing to map Another directly
                                into 'foo' and make it public
pub struct MyStruct {}
```

Here we no longer want the module another to be public, so we remove the `pub` keyword. Then, the `use` statement will map the fully qualified type of `Another` into the *foo* namespace (because we are in the foo module).

Last, let's update our main.rs —

```
mod foo;
use crate::foo::{MyStruct,Another};

fn main() {
  let _ms = MyStruct {};
  let _a = Another {};
}
```

Much better! Note that since we've mapped the type name of `Another` into the *foo* module, we can make use of the extended `use` syntax to import multiple names at once.

The key takeaway here is that you should really think of the *mod.rs* file as defining the interface to your module. Although it may seem a little daunting at first, it gives you a lot of control over exactly what is exposed publicly, while still allowing full access within the module (for things like testing).

Ok, that's great ... so what the heck is a *prelude* I hear you ask! Well, a prelude is just a pattern for making available all types you want to be public, in an idiomatic way. Not all crates define a prelude (although many do) and you don't always need one, but let's go ahead and define one for our little *hello_rust* project anyway.

Back to our *main.rs* we go —

```
mod foo;

mod prelude {                                  <-- Create module inline
  pub use crate::foo::{MyStruct,Another};      <-- Note the 'pub' here!
}

use crate::prelude::*;                         <-- Make the types exposed
                                                   in the prelude
                                                   available

fn main() {
  let _ms = MyStruct {};
  let _a = Another {};
}
```

We define the prelude as just another module (using `mod`), only this time we are specifying the module directly, instead of letting Rust look for the corresponding file or folder.

Now we can also use the `prelude` module just like any other, for example in the *mod.rs* file—

```
mod another;
pub use another::Another;

use crate::prelude::*;

pub struct MyStruct {}
```

In this contrived case, the prelude isn't necessary at all. But you can see that if you had declared multiple crates, standard library types, constants and other modules within the prelude, then you can access them immediately, with just the single `use` statement.

It also highlights a couple of other interesting parts of module use —

- You can import all public names from a module with a wildcard `::*`
- You can access the root of the module tree (ie, the main module in this case) using `crate::` and you can do this from anywhere in your application.

Summary

The module system in Rust was definitely one of the more puzzling aspects of the language. Coming from a C++/C# background, combined with the module visibility rules (and preludes), it was downright confusing! But once you wrap your head around what a module is (file, folder) and how you import them (`mod`) and then map names into different modules (`use`) it begins to make sense.

It's also important to keep in mind that Rust project structure is very specific (application vs library = *main.rs* vs *lib.rs*), requiring certain files to exist in different contexts (*mod.rs*).

Hope this was helpful (it was for me writing it!).

Next up, structs, associated functions and methods.

Rust | Software Development | Software Engineering | Programming Languages | Programming

Written by Gian Lorenzetto, PhD

105 Followers · Writer for CodeX

Senior Engineering Manger @ Octopus Deploy

Follow