[mod and the Filesystem 2](#_Toc478551195)

[Module Definitions 2](#_Toc478551196)

[Moving Modules to Other Files 5](#_Toc478551197)

[Rules of Module File Systems 11](#_Toc478551198)

[Controlling Visibility with pub 12](#_Toc478551199)

[Making a Function Public 14](#_Toc478551200)

[Privacy Rules 17](#_Toc478551201)

[Privacy Examples 17](#_Toc478551202)

[Looking at the Errors 17](#_Toc478551203)

[Fixing the Errors 18](#_Toc478551204)

[Importing Names 18](#_Toc478551205)

[Concise Imports with use 19](#_Toc478551206)

[Glob Imports with \* 21](#_Toc478551207)

[Using super to Access a Parent Module 21](#_Toc478551208)

[Summary 24](#_Toc478551209)

Chapter 7

Modules

When you start writing programs in Rust, your code might live solely in the main function. As your code grows, you’ll eventually move functionality into other functions for reuse and better organization. By splitting your code into smaller chunks, each chunk is easier to understand on its own. But what happens if you have too many functions? Rust has a module system that handles reusing code while keeping your code organized.

In the same way that you extract lines of code into a function, you can extract functions (and other code, like structs and enums) into different modules. A module is a namespace that contains definitions of functions or types, and you can choose whether those definitions are visible outside their module (public) or not (private). Here’s an overview of how modules work:

You declare a new module using the keyword mod.

By default, everything is set as private (including modules). You can use the pub keyword to make a module public and therefore visible outside its namespace.

The use keyword allows you to bring modules, or the definitions inside modules, into scope so it’s easier to refer to them.

We’ll look at each of these parts to see how they fit into the whole.

mod and the Filesystem

We’ll start our module example by making a new project with Cargo, but instead of creating a binary crate, we’ll make a library crate: a project that other people can pull into their projects as a dependency. You saw this with the rand crate in Chapter 2.

We’ll create a skeleton of a library that provides some general networking functionality; we’ll concentrate on the organization of the modules and functions but we won’t worry about what code goes in the function bodies. We’ll call our library communicator. By default, Cargo will create a library unless another type of project is specified: if we omit the --bin option that we’ve been using so far, our project will be a library:

$ cargo new communicator

$ cd communicator

Notice that Cargo generated src/lib.rs instead of src/main.rs. Inside src/lib.rs we’ll find the following:

Filename: src/lib.rs

#[cfg(test)]

mod tests {

#[test]

fn it\_works() {

}

}

Cargo creates an empty test to help us get our library started, rather than the “Hello, world!” binary that we get when we use the --bin option. We’ll look at the #[] and mod tests syntax a little later, but for now, make sure you leave it in src/lib.rs.

Because we don’t have a src/main.rs file, there’s nothing for Cargo to execute with the cargo run command. Therefore, we’ll use the cargo build command to compile our library crate’s code.

We’ll look at different options for organizing your library’s code that will be suitable in a variety of situations, depending on the intent of the code.

Module Definitions

For our communicator networking library, we’ll first define a module named network that contains the definition of a function called connect. Every module definition in Rust starts with the mod keyword. Add this code to the beginning of the src/lib.rs file, above the test code:

Filename: src/lib.rs

mod network {

fn connect() {

}

}

After the mod keyword, we put the name of the module, network, and then a block of code in curly braces. Everything inside this block is inside the namespace network. In this case, we have a single function, connect. If we wanted to call this function from a script outside the network module, we would need to specify the module and use the namespace syntax ::, like so: network::connect() rather than just connect().

We can also have multiple modules, side by side, in the same src/lib.rs file. For example, to also have a client module that has a function named connect as well, we can add it as shown in Listing 7-1:

Filename: src/lib.rs

mod network {

fn connect() {

}

}

mod client {

fn connect() {

}

}

Listing 7-1: The network module and the client module are defined side by side in src/lib.rs.

Now we have a network::connect function and a client::connect function. These can have completely different functionality, and the function names do not conflict with each other because they’re in different modules.

In this case, because we’re building a library, there’s nothing special about src/lib.rs. We could also use submodules in src/main.rs. In fact, we can put modules inside of modules, which can be useful as your modules grow to keep related functionality organized together and separate functionality apart. The choice of how you organize your code depends on how you think about the relationship between the parts of your code. For instance, the client code and its connect function might make more sense to users of our library if they were inside the network namespace instead, as in Listing 7-2:

Filename: src/lib.rs

mod network {

fn connect() {

}

mod client {

fn connect() {

}

}

}

Listing 7-2: Moving the client module inside the network module

In your src/lib.rs file, replace the existing mod network and mod client definitions with the ones in Listing 7-2, which have the client module as an inner module of network. Now that we have the functions network::connect and network::client::connect: again, the two functions named connect don’t conflict with each other because they’re in different namespaces.

In this way, modules form a hierarchy. The contents of src/lib.rs are at the topmost level, and the submodules are at lower levels. Here’s what the organization of our example in Listing 7-1 looks like when thought of this way:

communicator

├── network

└── client

And here’s the example in Listing 7-2:

communicator

└── network

└── client

You can see in Listing 7-2 that client is a child of the network module rather than a sibling. More complicated projects can have many modules, and they’ll need to be organized logically in order to keep track of them. What “logically” means in your project is up to you and depends on how you and your library’s users think about your project’s domain. Use the techniques shown here to create side-by-side modules and nested modules in whatever structure you would like.

Moving Modules to Other Files

Modules form a hierarchical structure, much like another structure in computing that you’re used to: filesystems! We can use Rust’s module system along with multiple files to split up Rust projects so not everything lives in src/lib.rs. For this example, let’s start with the code in Listing 7-3:

Filename: src/lib.rs

mod client {

fn connect() {

}

}

mod network {

fn connect() {

}

mod server {

fn connect() {

}

}

}

Listing 7-3: Three modules, client, network, and network::server, all defined in src/lib.rs

The file src/lib.rs has this module hierarchy:

communicator

├── client

└── network

└── server

If these modules had many functions, and those functions were becoming lengthy, it would be difficult to scroll through this file to find the code we wanted to work with. Because the functions are nested inside one or more mod blocks, the lines of code inside the functions will start getting lengthy as well. These would be good reasons to separate the client, network, and server modules from src/lib.rs and place them into their own files.

Let’s start by extracting the client module into another file. First, replace the client module code in src/lib.rs with the following:

Filename: src/lib.rs

mod client;

mod network {

fn connect() {

}

mod server {

fn connect() {

}

}

}

We’re still defining the client module here, but by removing the curly braces and definitions inside the client module and replacing them with a semicolon, we’re telling Rust to look in another location for the code defined inside that module.

Now we need to create the external file with that module name. Create a client.rs file in your src/ directory, and then open it and enter the following, which is the connect function in the client module that we removed in the previous step:

Filename: src/client.rs

fn connect() {

}

Note that we don’t need a mod declaration in this file because we already declared the client module with mod in src/lib.rs. This file just provides the contents of the client module. If we put a mod client here, we’d be giving the client module its own submodule named client!

Rust only knows to look in src/lib.rs by default. If we want to add more files to our project, we need to tell Rust in src/lib.rs to look in other files; this is why mod client needs to be defined in src/lib.rs and can’t be defined in src/client.rs.

Now, everything should compile successfully, although you’ll get a few warnings. Remember to use cargo build instead of cargo run because we have a library crate rather than a binary crate:

$ cargo build

Compiling communicator v0.1.0 (file:///projects/communicator)

warning: function is never used: `connect`, #[warn(dead\_code)] on by default

--> src/client.rs:1:1

|

1 | fn connect() {

| ^

warning: function is never used: `connect`, #[warn(dead\_code)] on by default

--> src/lib.rs:4:5

|

4 | fn connect() {

| ^

warning: function is never used: `connect`, #[warn(dead\_code)] on by default

--> src/lib.rs:8:9

|

8 | fn connect() {

| ^

These warnings tell us that we have functions that are never used. Don’t worry about these warnings for now; we’ll address them later in the chapter. The good news is that they’re just warnings; our project built successfully!

Next, let’s extract the network module into its own file using the same pattern. In src/lib.rs, delete the body of the network module and add a semicolon to the declaration, like so:

Filename: src/lib.rs

mod client;

mod network;

Then create a new src/network.rs file and enter the following:

Filename: src/network.rs

fn connect() {

}

mod server {

fn connect() {

}

}

Notice that we still have a mod declaration within this module file; this is because we still want server to be a submodule of network.

Run cargo build again. Success! We have one more module to extract: server. Because it’s a submodule—that is, a module within a module—our current tactic of extracting a module into a file named after that module won’t work. We’ll try anyway so you can see the error. First, change src/network.rs to have mod server; instead of the server module’s contents:

Filename: src/network.rs

fn connect() {

}

mod server;

Then create a src/server.rs file and enter the contents of the server module that we extracted:

Filename: src/server.rs

fn connect() {

}

When we try to cargo build, we’ll get the error shown in Listing 7-4:

$ cargo build

Compiling communicator v0.1.0 (file:///projects/communicator)

error: cannot declare a new module at this location

--> src/network.rs:4:5

|

4 | mod server;

| ^^^^^^

|

note: maybe move this module `network` to its own directory via `network/mod.rs`

--> src/network.rs:4:5

|

4 | mod server;

| ^^^^^^

note: ... or maybe `use` the module `server` instead of possibly redeclaring it

--> src/network.rs:4:5

|

4 | mod server;

| ^^^^^^

Listing 7-4: Error when trying to extract the server submodule into src/server.rs

The error says we cannot declare a new module at this location and is pointing to the mod server; line in src/network.rs. So src/network.rs is different than src/lib.rs somehow: keep reading to understand why.

The note in the middle of Listing 7-4 is actually very helpful because it points out something we haven’t yet talked about doing:

note: maybe move this module network to its own directory via  
network/mod.rs

Instead of continuing to follow the same file naming pattern we used previously, we can do what the note suggests:

Make a new directory named network, the parent module’s name.

Move the src/network.rs file into the new network directory, and rename  
it to src/network/mod.rs.

Move the submodule file src/server.rs into the network directory.

Here are commands to carry out these steps:

$ mkdir src/network

$ mv src/network.rs src/network/mod.rs

$ mv src/server.rs src/network

Now when we try to run cargo build, compilation will work (we’ll still have warnings though). Our module layout still looks like this, which is exactly the same as it did when we had all the code in src/lib.rs in Listing 7-3:

communicator

├── client

└── network

└── server

The corresponding file layout now looks like this:

├── src

│ ├── client.rs

│ ├── lib.rs

│ └── network

│ ├── mod.rs

│ └── server.rs

So when we wanted to extract the network::server module, why did we have to also change the src/network.rs file to the src/network/mod.rs file and put the code for network::server in the network directory in src/network/server.rs instead of just being able to extract the network::server module into src/server.rs? The reason is that Rust wouldn’t be able to recognize that server was supposed to be a submodule of network if the server.rs file was in the src directory. To clarify Rust’s behavior here, let’s consider a different example with the following module hierarchy, where all the definitions are in src/lib.rs:

communicator

├── client

└── network

└── client

In this example, we have three modules again: client, network, and network::client. Following the same steps we did earlier for extracting modules into files, we would create src/client.rs for the client module. For the network module we would create src/network.rs. But we wouldn’t be able to extract the network::client module into a src/client.rs file because that already exists for the top-level client module! If we put the code in both the client and network::client modules in the src/client.rs file, Rust wouldn’t have any way to know whether the code was for client or for network::client.

Therefore, in order to extract a file for the network::client submodule of the network module, we needed to create a directory for the network module instead of a src/network.rs file. The code that is in the network module then goes into the src/network/mod.rs file, and the submodule network::client can have its own src/network/client.rs file. Now the top-level src/client.rs is unambiguously the code that belongs to the client module.

Rules of Module Filesystems

Let’s summarize the rules of modules with regard to files:

If a module named foo has no submodules, you should put the declarations for foo in a file named foo.rs.

If a module named foo does have submodules, you should put the declarations for foo in a file named foo/mod.rs.

These rules apply recursively, so if a module named foo has a submodule named bar and bar does not have submodules, you should have the following files in your src directory:

├── foo

│ ├── bar.rs (contains the declarations in `foo::bar`)

│ └── mod.rs (contains the declarations in `foo`, including `mod bar`)

The modules should be declared in their parent module’s file using the mod keyword.

Next, we’ll talk about the pub keyword and get rid of those warnings!

Controlling Visibility with pub

We resolved the error messages shown in Listing 7-4 by moving the network and network::server code into the src/network/mod.rs and src/network/server.rs files, respectively. At that point, cargo build was able to build our project, but we still get warning messages about the client::connect, network::connect, and network::server::connect functions not being used:

warning: function is never used: `connect`, #[warn(dead\_code)] on by default

src/client.rs:1:1

|

1 | fn connect() {

| ^

warning: function is never used: `connect`, #[warn(dead\_code)] on by default

--> src/network/mod.rs:1:1

|

1 | fn connect() {

| ^

warning: function is never used: `connect`, #[warn(dead\_code)] on by default

--> src/network/server.rs:1:1

|

1 | fn connect() {

| ^

So why are we receiving these warnings? After all, we’re building a library with functions that are intended to be used by our users, not necessarily by us within our own project, so it shouldn’t matter that these connect functions go unused. The point of creating them is that they will be used by another project, not our own.

To understand why this program invokes these warnings, let’s try using the connect library as if we were another project, calling it externally. To do that, we’ll create a binary crate in the same directory as our library crate by making a src/main.rs file containing this code:

Filename: src/main.rs

extern crate communicator;

fn main() {

communicator::client::connect();

}

We use the extern crate command to bring the communicator library crate into scope, because our package now contains two crates. Cargo treats src/main.rs as the root file of a binary crate, which is separate from the existing library crate whose root file is src/lib.rs. This pattern is quite common for executable projects: most functionality is in a library crate, and the binary crate uses that library crate. As a result, other programs can also use the library crate, and it’s a nice separation of concerns.

From the point of view of a crate outside the communicator library looking in, all the modules we’ve been creating are within a module that has the same name as the crate, communicator. We call the top-level module of a crate the root module.

Also note that even if we’re using an external crate within a submodule of our project, the extern crate should go in our root module (so in src/main.rs or src/lib.rs). Then, in our submodules, we can refer to items from external crates as if the items are top-level modules.

Right now, our binary crate just calls our library’s connect function from the client module. However, invoking cargo build will now give us an error after the warnings:

error: module `client` is private

--> src/main.rs:4:5

|

4 | communicator::client::connect();

| ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

Ah ha! This error tells usthat the client module is private, which is the crux of the warnings. It’s also the first time we’ve run into the concepts of public and private in the context of Rust. The default state of all code in Rust is private: no one else is allowed to use the code. If you don’t use a private function within your program, because your program is the only code allowed to use that function, Rust will warn you that the function has gone unused.

After we specify that a function like client::connect is public, not only will our call to that function from our binary crate be allowed, but the warning that the function is unused will go away. Marking something public lets Rust know that the function will be used by code outside of our program. Rust considers the theoretical external usage that’s now possible as the function “being used.” Thus, when something is marked public, Rust will not require that it be used in our program and will stop warning that the item is unused.

Making a Function Public

To tell Rust to make something public, we add the pub keyword to the start of the declaration of the item we want to make public. We’ll focus on fixing the warning that indicates that client::connect has gone unused for now, as well as the “module client is private” error from our binary crate. Modify src/lib.rs to make the client module public, like so:

Filename: src/lib.rs

pub mod client;

mod network;

The pub keyword is placed right before mod. Let’s try building again:

<warnings>

error: function `connect` is private

--> src/main.rs:4:5

|

4 | communicator::client::connect();

| ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

Hooray! We have a different error! Yes, different error messages are a cause for celebration. The new error shows function connect is private, so let’s edit src/client.rs to make client::connect public too:

Filename: src/client.rs

pub fn connect() {

}

Now run cargo build again:

warning: function is never used: `connect`, #[warn(dead\_code)] on by default

--> src/network/mod.rs:1:1

|

1 | fn connect() {

| ^

warning: function is never used: `connect`, #[warn(dead\_code)] on by default

--> src/network/server.rs:1:1

|

1 | fn connect() {

| ^

The code compiled, and the warning about client::connect not being used is gone!

Unused code warnings don’t always indicate that something needs to be made public: if you didn’t want these functions to be part of your public API, unused code warnings could be alerting you to code you no longer needed and can safely delete. They could also be alerting you to a bug if you had just accidentally removed all places within your library where this function is called.

But in this case, we do want the other two functions to be part of our crate’s public API, so let’s mark them as pub as well to try to get rid of the remaining warnings. Modify src/network/mod.rs to look like the following:

Filename: src/network/mod.rs

pub fn connect() {

}

mod server;

Then compile the code:

warning: function is never used: `connect`, #[warn(dead\_code)] on by default

--> src/network/mod.rs:1:1

|

1 | pub fn connect() {

| ^

warning: function is never used: `connect`, #[warn(dead\_code)] on by default

--> src/network/server.rs:1:1

|

1 | fn connect() {

| ^

Hmmm, we’re still getting an unused function warning, even though network::connect is set to pub. The reason is that the function is public within the module, but the network module that the function resides in is not public. We’re working from the interior of the library out this time, whereas with client::connect we worked from the outside in. We need to change src/lib.rs to make network public too, like so:

Filename: src/lib.rs

pub mod client;

pub mod network;

Now when we compile, that warning is gone:

warning: function is never used: `connect`, #[warn(dead\_code)] on by default

--> src/network/server.rs:1:1

|

1 | fn connect() {

| ^

Only one warning is left! Try to fix this one on your own!

Privacy Rules

Overall, these are the rules for item visibility:

If an item is public, it can be accessed through any of its parent modules.

If an item is private, it can be accessed only by the current module and its child modules.

Privacy Examples

Let’s look at a few more privacy examples to get some practice. Create a new library project and enter the code in Listing 7-5 into your new project’s src/lib.rs:

Filename: src/lib.rs

mod outermost {

pub fn middle\_function() {}

fn middle\_secret\_function() {}

mod inside {

pub fn inner\_function() {}

fn secret\_function() {}

}

}

fn try\_me() {

outermost::middle\_function();

outermost::middle\_secret\_function();

outermost::inside::inner\_function();

outermost::inside::secret\_function();

}

Listing 7-5: Examples of private and public functions, some of which are incorrect

Before you try to compile this code, make a guess about which lines in the try\_me function will have errors. Then, try compiling the code to see whether you were right, and read on for the discussion of the errors!

Looking at the Errors

The try\_me function is in the root module of our project. The module named outermost is private, but the second privacy rule states that the try\_me function is allowed to access the outermost module because outermost is in the current (root) module, as is try\_me.

The call to outermost::middle\_function will work because middle\_function is public, and try\_me is accessing middle\_function through its parent module outermost. We determined in the previous paragraph that this module is accessible.

The call to outermost::middle\_secret\_function will cause a compilation error. middle\_secret\_function is private, so the second rule applies. The root module is neither the current module of middle\_secret\_function (outermost is), nor is it a child module of the current module of middle\_secret\_function.

The module named inside is private and has no child modules, so it can only be accessed by its current module outermost. That means the try\_me function is not allowed to call outermost::inside::inner\_function or outermost::inside::secret\_function.

Fixing the Errors

Here are some suggestions for changing the code in an attempt to fix the errors. Before you try each one, make a guess as to whether it will fix the errors, and then compile the code to see whether or not you’re right, using the privacy rules to understand why.

What if the inside module was public?

What if outermost was public and inside was private?

What if, in the body of inner\_function, you called ::outermost::middle\_secret\_function()? (The two colons at the beginning mean that we want to refer to the modules starting from the root module.)

Feel free to design more experiments and try them out!

Next, let’s talk about bringing items into a scope with the use keyword.

Importing Names

We’ve covered how to call functions defined within a module using the module name as part of the call, as in the call to the nested\_modules function shown here in Listing 7-6:

Filename: src/main.rs

pub mod a {

pub mod series {

pub mod of {

pub fn nested\_modules() {}

}

}

}

fn main() {

a::series::of::nested\_modules();

}

Listing 7-6: Calling a function by fully specifying its enclosing module’s path

As you can see, referring to the fully qualified name can get quite lengthy. Fortunately, Rust has a keyword to make these calls more concise.

Concise Imports with use

Rust’s use keyword shortens lengthy function calls by bringing the modules of the function you want to call into a scope. Here’s an example of bringing the a::series::of module into a binary crate’s root scope:

Filename: src/main.rs

pub mod a {

pub mod series {

pub mod of {

pub fn nested\_modules() {}

}

}

}

use a::series::of;

fn main() {

of::nested\_modules();

}

The line use a::series::of; means that rather than using the full a::series::of path wherever we want to refer to the of module, we can use of.

The use keyword brings only what we’ve specified into scope: it does not bring children of modules into scope. That’s why we still have to use of::nested\_modules when we want to call the nested\_modules function.

We could have chosen to bring the function into scope by instead specifying the function in the use as follows:

pub mod a {

pub mod series {

pub mod of {

pub fn nested\_modules() {}

}

}

}

use a::series::of::nested\_modules;

fn main() {

nested\_modules();

}

Doing so allows us to exclude all the modules and reference the function directly.

Because enums also form a sort of namespace like modules, we can import an enum’s variants with use as well. For any kind of use statement, if you’re importing multiple items from one namespace, you can list them using curly braces and commas in the last position, like so:

enum TrafficLight {

Red,

Yellow,

Green,

}

use TrafficLight::{Red, Yellow};

fn main() {

let red = Red;

let yellow = Yellow;

let green = TrafficLight::Green; // because we didn’t `use` TrafficLight::Green

}

Glob Imports with \*

To import all the items in a namespace at once, we can use the \* syntax. For example:

enum TrafficLight {

Red,

Yellow,

Green,

}

use TrafficLight::\*;

fn main() {

let red = Red;

let yellow = Yellow;

let green = Green;

}

The \* is called a glob, and it will import everything that’s visible inside the namespace. You should use globs sparingly: they are convenient, but you might also pull in more things than you expected and cause naming conflicts.

Using super to Access a Parent Module

As you now know, when you create a library crate, Cargo makes a tests module for you. Let’s go into more detail about that now. In your communicator project, open src/lib.rs:

Filename: src/lib.rs

pub mod client;

pub mod network;

#[cfg(test)]

mod tests {

#[test]

fn it\_works() {

}

}

Chapter 12 explains more about testing, but parts of this example should make sense now: we have a module named tests that lives next to our other modules and contains one function named it\_works. Even though there are special annotations, the tests module is just another module! So our module hierarchy looks like this:

communicator

├── client

├── network

| └── client

└── tests

Tests are for exercising the code within our library, so let’s try to call our client::connect function from this it\_works function, even though we won’t be checking any functionality right now:

Filename: src/lib.rs

#[cfg(test)]

mod tests {

#[test]

fn it\_works() {

client::connect();

}

}

Run the tests by invoking the cargo test command:

$ cargo test

Compiling communicator v0.1.0 (file:///projects/communicator)

error[E0433]: failed to resolve. Use of undeclared type or module `client`

--> src/lib.rs:9:9

|

9 | client::connect();

| ^^^^^^^^^^^^^^^ Use of undeclared type or module `client`

warning: function is never used: `connect`, #[warn(dead\_code)] on by default

--> src/network/server.rs:1:1

|

1 | fn connect() {

| ^

The compilation failed, but why? We don’t need to place communicator:: in front of the function like we did in src/main.rs because we are definitely within the communicator library crate here. The reason is that paths are always relative to the current module, which here is tests. The only exception is in a use statement, where paths are relative to the crate root by default. Our tests module needs the client module in its scope!

So how do we get back up one module in the module hierarchy to call the client::connect function in the tests module? In the tests module, we can either use leading colons to let Rust know that we want to start from the root and list the whole path, like this:

::client::connect();

Or, we can use super to move up one module in the hierarchy from our current module, like this:

super::client::connect();

These two options don’t look that different in this example, but if you’re deeper in a module hierarchy, starting from the root every time would make your code lengthy. In those cases, using super to get from the current module to sibling modules is a good shortcut. Plus, if you’ve specified the path from the root in many places in your code and then you rearrange your modules by moving a subtree to another place, you’d end up needing to update the path in several places, which would be tedious.

It would also be annoying to have to type super:: in each test, but you’ve already seen the tool for that solution: use! The super:: functionality changes the path you give to use so it is relative to the parent module instead of to the root module.

For these reasons, in the tests module especially, use super::something is usually the best solution. So now our test looks like this:

Filename: src/lib.rs

#[cfg(test)]

mod tests {

use super::client;

#[test]

fn it\_works() {

client::connect();

}

}

When we run cargo test again, the test will pass and the first part of the test result output will be the following:

$ cargo test

Compiling communicator v0.1.0 (file:///projects/communicator)

Running target/debug/communicator-92007ddb5330fa5a

running 1 test

test tests::it\_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured

Summary

Now you know some new techniques for organizing your code! Use these techniques to group related functionality together, keep files from becoming too long, and present a tidy public API to your library users.

Next, we’ll look at some collection data structures in the standard library that you can use in your nice, neat code!