[Packages and Crates 2](#_Toc876635)

[Defining Modules to Control Scope and Privacy 4](#_Toc876636)

[Paths for Referring to an Item in the Module Tree 6](#_Toc876637)

[Exposing Paths with the pub Keyword 9](#_Toc876638)

[Starting Relative Paths with super 11](#_Toc876639)

[Making Structs and Enums Public 12](#_Toc876640)

[Bringing Paths into Scope with the use Keyword 14](#_Toc876641)

[Creating Idiomatic use Paths 16](#_Toc876642)

[Providing New Names with the as Keyword 18](#_Toc876643)

[Re-exporting Names with pub use 18](#_Toc876644)

[Using External Packages 19](#_Toc876645)

[Using Nested Paths to Clean Up Large use Lists 20](#_Toc876646)

[The Glob Operator 21](#_Toc876647)

[Separating Modules into Different Files 22](#_Toc876648)

[Summary 23](#_Toc876649)

7

Managing Growing Projects with Packages, Crates, and Modules

As you write large programs, organizing your code is important because it’ll become impossible to keep track of your entire program in your head. By grouping related functionality and separating code with distinct features, you’ll clarify where to find code that implements a particular feature and where to go to change how a feature works.

The programs we’ve written so far have been in one module in one file. As a project grows, you can organize code by splitting it into multiple modules and then multiple files. A package can contain multiple binary crates and optionally one library crate. As a package grows, you can extract parts into separate crates that become external dependencies. This chapter covers all these techniques. For very large projects of a set of interrelated packages that evolve together, Cargo provides workspaces, which we’ll cover in the section “Cargo Workspaces” in Chapter 14.

prod: check xref

In addition to grouping functionality, encapsulating implementation details lets you reuse code at a higher level: once you’ve implemented an operation, other code can call that code without knowing how the implementation works via the code’s public interface. The way you write code defines which parts are public for other code to use and which parts are private implementation details that you reserve the right to change and that other code shouldn’t have to worry about. This is another way to limit the amount of detail you have to keep in your head.

A related aspect to organization and encapsulation is scope: the nested context code is written using a set of names that are defined as “in scope.” When reading, writing, and compiling code, programmers and compilers need to know whether a particular name at a particular spot refers to a variable, function, struct, enum, module, constant, or other item, and what that item means. You can create scopes and change which names are in or out of scope. You can’t have two items with the same name in the same scope; tools are available to resolve name conflicts.

Rust has a number of features that allow you to manage your code’s organization, including which details are exposed and which details are private, and what names are in each scope in your programs. These features are sometimes collectively referred to as the module system and include:

Packages: A Cargo feature that lets you build, test, and share crates

Crates: A tree of modules that produces a library or executable

Modules and use: Let you control the organization, scope, and privacy of paths

Paths: A way of naming an item, such as a struct, function, or module

In this chapter, we’ll cover all these features, discuss how they interact, and explain how to use them to manage scope. By the end, you should have a solid understanding of the module system and be able to work with scopes like a pro!

Packages and Crates

The first parts of the module system we’ll cover are packages and crates. A crate is a binary or library. The crate root is a source file that the Rust compiler starts from and makes up the root module of your crate (we’ll explain modules in depth in the section “Defining Modules to Control Scope and Privacy”). A package is one or more crates that provide a set of functionality. A package contains a Cargo.toml file that describes how to build those crates.

Several rules determine what a package can contain. A package must contain zero or one library crate, and no more. It can contain as many binary crates as you’d like, but it must contain at least one crate (either library or binary).

Let’s walk through what happens when we create a package. First, we enter the command cargo new:

$ cargo new my-project

Created binary (application) `my-project` package

$ ls my-project

Cargo.toml

src

$ ls my-project/src

main.rs

When we entered the command, Cargo created a Cargo.toml file, giving us a package. Looking at the contents of Cargo.toml, there’s no mention of src/main.rs because Cargo follows a convention that src/main.rs is the crate root of a binary crate with the same name as the package. Likewise, Cargo knows that if the package directory contains src/lib.rs, the package contains a library crate with the same name as the package, and src/lib.rs is its crate root. Cargo passes the crate root files to rustc to build the library or binary.

Here, we have a package that only contains src/main.rs, meaning it only contains a binary crate named my-project. If a package contains src/main.rs and src/lib.rs, it has two crates: a library and a binary, both with the same name as the package. A package can have multiple binary crates by placing files in the src/bin directory: each file will be a separate binary crate.

A crate will group related functionality together in a scope so the functionality is easy to share between multiple projects. For example, the rand crate we used in Chapter 2 provides functionality that generates random numbers. We can use that functionality in our own projects by bringing the rand crate into our project’s scope. All the functionality provided by the rand crate is accessible through the crate’s name, rand.

prod: check xref

Keeping a crate’s functionality in its own scope clarifies whether particular functionality is defined in our crate or the rand crate and prevents potential conflicts. For example, the rand crate provides a trait named Rng. We can also define a struct named Rng in our own crate. Because a crate’s functionality is namespaced in its own scope, when we add rand as a dependency, the compiler isn’t confused about what the name Rng refers to. In our crate, it refers to the struct Rng that we defined. We would access the Rng trait from the rand crate as rand::Rng.

Let’s move on and talk about the module system!

Defining Modules to Control Scope and Privacy

In this section, we’ll talk about modules and other parts of the module system, namely paths that allow you to name items; the use keyword that brings a path into scope; and the pub keyword to make items public. We’ll also discuss using the as keyword, external packages, and the glob operator. For now, let’s focus on modules!

Modules let us organize code within a crate into groups for readability and easy reuse. Modules also control the privacy of items, which is whether an item can be used by outside code (public) or whether it’s an internal implementation detail and not available for outside use (private).

As an example, let’s write a library crate that provides the functionality of a restaurant. We’ll define the signatures of functions but leave their bodies empty to concentrate on the organization of the code rather than actually implementing a restaurant in code.

In the restaurant industry, parts of a restaurant are referred to as front of house and others as back of house. Front of house is where customers are and includes hosts seating customers, servers taking orders and payment, and bartenders making drinks. Back of house includes the chefs and cooks in the kitchen, dishwashers cleaning up, and managers doing administrative work.

To structure our crate in the same way that a real restaurant works, we can organize the functions into nested modules. Create a new library named restaurant by running cargo new --lib restaurant; then put the code in Listing 7-1 into src/lib.rs to define some modules and function signatures.

Filename: src/lib.rs

mod front\_of\_house {

mod hosting {

fn add\_to\_waitlist() {}

fn seat\_at\_table() {}

}

mod serving {

fn take\_order() {}

fn serve\_order() {}

fn take\_payment() {}

}

}

Listing 7-1: A front\_of\_house module containing other modules that then contain functions

We define a module by starting with the mod keyword, and then specify the name of the module (in this case, front\_of\_house) and place curly brackets around the body of the module. Inside modules, we can have other modules, as in this case with the modules hosting and serving. Modules can also hold definitions for other items, such as structs, enums, constants, traits, or as in Listing 7-1, functions.

By using modules, we can group related definitions together and name why they’re related. Programmers using this code would have an easier time finding the definitions they want to use because they could navigate the code based on the groups rather than having to read through all the definitions. Programmers adding new functionality to this code would know where to place the code to keep the program organized.

Earlier, we mentioned that src/main.rs and src/lib.rs are called crate roots. The reason for their name is that the contents of either of these two files form a module named crate at the root of the crate’s module structure, known as the module tree.

Listing 7-2 shows the module tree for the structure in Listing 7-1.

crate

└── front\_of\_house

├── hosting

│ ├── add\_to\_waitlist

│ └── seat\_at\_table

└── serving

├── take\_order

├── serve\_order

└── take\_payment

Listing 7-2: The module tree for the code in Listing 7-1

This tree shows how some of the modules nest inside one another (such as hosting nests inside front\_of\_house). The tree also shows how some modules are siblings to each other, meaning they’re defined in the same module (hosting and serving are defined within front\_of\_house). To continue the family metaphor, if module A is contained inside module B, we say that module A is the child of module B, and that module B is the parent of module A. Notice that the entire module tree is rooted under the implicit module named crate.

The module tree might remind you of the filesystem’s directory tree on your computer; this is a very apt comparison! Just like directories in a filesystem, you use modules to organize your code. And just like files in a directory, we need a way to find our modules.

Paths for Referring to an Item in the Module Tree

To show Rust where to find an item in a module tree, we use a path in the same way we use a path when navigating a filesystem. If we want to call a function, we need to know its path.

A path can take two forms:

An absolute path starts from a crate root by using a crate name or a literal crate.

A relative path starts from the current module and uses self, super, or an identifier in the current module.

Both absolute and relative paths are followed by one or more identifiers separated by double colons (::).

Let’s return to the example in Listing 7-1. How do we call the add\_to\_waitlist function? This is the same as asking, what’s the path of the add\_to\_waitlist function? In Listing 7-3, we simplified our code a bit by removing some of the modules and functions. We’ll show two ways to call the add\_to\_waitlist function from a new function eat\_at\_restaurant defined in the crate root. Note that this example won’t compile just yet; we’ll explain why in a bit.

Filename: src/lib.rs

mod front\_of\_house {

mod hosting {

fn add\_to\_waitlist() {}

}

}

pub fn eat\_at\_restaurant() {

// Absolute path

crate::front\_of\_house::hosting::add\_to\_waitlist();

// Relative path

front\_of\_house::hosting::add\_to\_waitlist();

}

Listing 7-3: Calling the add\_to\_waitlist function using absolute and relative paths

The first time we call the add\_to\_waitlist function in eat\_at\_restaurant, we use an absolute path. The add\_to\_waitlist function is defined in the same crate as eat\_at\_restaurant, which means we can use the crate keyword to start an absolute path.

After crate, we include each of the successive modules until we make our way to add\_to\_waitlist. You can imagine a filesystem with the same structure, and we’d specify the path /front\_of\_house/hosting/add\_to\_waitlist to run the add\_to\_waitlist program; using the crate name to start from the crate root is like using / to start from the filesystem root in your shell.

The second time we call add\_to\_waitlist in eat\_at\_restaurant, we use a relative path. The path starts with front\_of\_house, the name of the module defined at the same level of the module tree as eat\_at\_restaurant. Here the filesystem equivalent would be using the path front\_of\_house/hosting/add\_to\_waitlist. Starting with a name means that the path is relative.

Choosing whether to use a relative or absolute path is a decision you’ll make based on your project. The decision should depend on whether you’re more likely to move item definition code separately from or together with the code that uses the item. For example, if we move the front\_of\_house module and the eat\_at\_restaurant function into a module named customer\_experience, we’d need to update the absolute path to add\_to\_waitlist, but the relative path would still be valid. However, if we moved the eat\_at\_restaurant function separately into a module named dining, the absolute path to the add\_to\_waitlist call would stay the same, but the relative path would need to be updated. We tend to specify absolute paths because it’s more likely to move code definitions and item calls independently of each other.

Let’s try to compile Listing 7-3 and find out why it won’t compile yet! The error we get is shown in Listing 7-4.

$ cargo build

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

error[E0603]: module `hosting` is private

--> src/lib.rs:9:28

|

9 | crate::front\_of\_house::hosting::add\_to\_waitlist();

| ^^^^^^^

error[E0603]: module `hosting` is private

--> src/lib.rs:12:21

|

12 | front\_of\_house::hosting::add\_to\_waitlist();

| ^^^^^^^

Listing 7-4: Compiler errors from building the code in Listing 7-3

The error messages say that module hosting is private. In other words, we have the correct paths for the hosting module and the add\_to\_waitlist function, but Rust won’t let us use them because it doesn’t have access to the private sections.

Modules aren’t only useful for organizing your code, they also define Rust’s privacy boundary: the line that separates the implementation details external code isn’t allowed to know about, call, or rely on. So, if you want to make an item like a function or struct private, you put it in a module.

The way privacy works in Rust is that all items (functions, methods, structs, enums, modules, and constants) are private by default. Items in a parent module can’t use the private items inside child modules, but items in child modules can use the items in their ancestor modules. The reason is that child modules wrap and hide their implementation details, but the child modules can see the context in which they’re defined. To continue with the restaurant metaphor, think of the privacy rules like the back office of a restaurant: what goes on in there is private to restaurant customers, but office managers can see and do everything in the restaurant in which they operate.

Rust chose to have the module system function this way so that hiding inner implementation details is the default. That way, you know which parts of the inner code you can change without breaking outer code. But you can expose inner parts of child modules code to outer ancestor modules by making an item public using the pub keyword.

Exposing Paths with the pub Keyword

Let’s return to the error in Listing 7-4 that told us the hosting module is private. We want the eat\_at\_restaurant function in the parent module to have access to the add\_to\_waitlist function in the child module, so we mark the hosting module with the pub keyword, as shown in Listing 7-5.

Filename: src/lib.rs

mod front\_of\_house {

pub mod hosting {

fn add\_to\_waitlist() {}

}

}

pub fn eat\_at\_restaurant() {

// Absolute path

crate::front\_of\_house::hosting::add\_to\_waitlist();

// Relative path

front\_of\_house::hosting::add\_to\_waitlist();

}

Listing 7-5: Declaring the hosting module as pub to use it from eat\_at\_restaurant

Unfortunately, the code in Listing 7-5 still results in an error, as shown in Listing 7-6.

$ cargo build

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

error[E0603]: function `add\_to\_waitlist` is private

--> src/lib.rs:9:37

|

9 | crate::front\_of\_house::hosting::add\_to\_waitlist();

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

error[E0603]: function `add\_to\_waitlist` is private

--> src/lib.rs:12:30

|

12 | front\_of\_house::hosting::add\_to\_waitlist();

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

Listing 7-6: Compiler errors from building the code in Listing 7-5

What happened? Adding the pub keyword in front of mod hosting makes the module public. With this change, if we can access front\_of\_house, we can access hosting. But the contents of hosting are still private; making the module public doesn’t make its contents public. The pub keyword on a module only lets code in its ancestor modules refer to it.

The errors in Listing 7-6 say that the add\_to\_waitlist function is private. The privacy rules apply to structs, enums, functions, and methods as well as modules.

Let’s also make the add\_to\_waitlist function public by adding the pub keyword before its definition, as in Listing 7-7.

Filename: src/lib.rs

mod front\_of\_house {

pub mod hosting {

pub fn add\_to\_waitlist() {}

}

}

pub fn eat\_at\_restaurant() {

// Absolute path

crate::front\_of\_house::hosting::add\_to\_waitlist();

// Relative path

front\_of\_house::hosting::add\_to\_waitlist();

}

Listing 7-7: Adding the pub keyword to mod hosting and fn add\_to\_waitlist lets us call the function from eat\_at\_restaurant

Now the code will compile! Let’s look at the absolute and the relative path, and double-check why adding the pub keyword lets us use these paths in add\_to\_waitlist with respect to the privacy rules.

In the absolute path, we start with crate, the root of our crate’s module tree. Then the front\_of\_house module is defined in the crate root. The front\_of\_house module isn’t public, but because the eat\_at\_restaurant function is defined in the same module as front\_of\_house (that is, eat\_at\_restaurant and front\_of\_house are siblings), we can refer to front\_of\_house from eat\_at\_restaurant. Next is the hosting module marked with pub. We can access the parent module of hosting, so we can access hosting. Finally, the add\_to\_waitlist function is marked with pub and we can access its parent module, so this function call works!

In the relative path, the logic is the same as the absolute path except for the first step: rather than starting from the crate root, the path starts from front\_of\_house. The front\_of\_house module is defined within the same module as eat\_at\_restaurant, so the relative path starting from the module in which eat\_at\_restaurant is defined works. Then, because hosting and add\_to\_waitlist are marked with pub, the rest of the path works and this function call is valid!

Starting Relative Paths with super

We can also construct relative paths that begin in the parent module by using super at the start of the path. This is like starting a filesystem path with the .. syntax. Why would we want to do this?

Consider the code in Listing 7-8 that models the situation in which a chef fixes an incorrect order and personally brings it out to the customer. The function fix\_incorrect\_order calls the function serve\_order by specifying the path to serve\_order starting with super:

Filename: src/lib.rs

fn serve\_order() {}

mod back\_of\_house {

fn fix\_incorrect\_order() {

cook\_order();

super::serve\_order();

}

fn cook\_order() {}

}

Listing 7-8: Calling a function using a relative path starting with super

The fix\_incorrect\_order function is in the back\_of\_house module, so we can use super to go to the parent module of back\_of\_house, which in this case is crate, the root. From there, we look for serve\_order and find it. Success! We think the back\_of\_house module and the serve\_order function are likely to stay in the same relationship to each other and get moved together should we decide to reorganize the crate’s module tree. Therefore, we used super so we’ll have fewer places to update code in the future if this code gets moved to a different module.

Making Structs and Enums Public

We can also use pub to designate structs and enums as public, but there are a few extra details. If we use pub before a struct definition, we make the struct public, but the struct’s fields will still be private. We can make each field public or not on a case-by-case basis. In Listing 7-9, we’ve defined a public back\_of\_house::Breakfast struct with a public toast field but a private seasonal\_fruit field. This models the case in a restaurant where the customer can pick the type of bread that comes with a meal, but the chef decides which fruit accompanies the meal based on what’s in season and in stock. The available fruit changes quickly, so customers can’t choose the fruit or even see which fruit they’ll get.

Filename: src/lib.rs

mod back\_of\_house {

pub struct Breakfast {

pub toast: String,

seasonal\_fruit: String,

}

impl Breakfast {

pub fn summer(toast: &str) -> Breakfast {

Breakfast {

toast: String::from(toast),

seasonal\_fruit: String::from("peaches"),

}

}

}

}

pub fn eat\_at\_restaurant() {

// Order a breakfast in the summer with Rye toast

let mut meal = back\_of\_house::Breakfast::summer("Rye");

// Change our mind about what bread we'd like

meal.toast = String::from("Wheat");

println!("I'd like {} toast please", meal.toast);

// The next line won't compile if we uncomment it; we're not allowed

// to see or modify the seasonal fruit that comes with the meal

// meal.seasonal\_fruit = String::from("blueberries");

}

Listing 7-9: A struct with some public fields and some private fields

Because the toast field in the back\_of\_house::Breakfast struct is public, in eat\_at\_restaurant we can write and read to the toast field using dot notation. Notice that we can’t use the seasonal\_fruit field in eat\_at\_restaurant because seasonal\_fruit is private. Try uncommenting the line modifying the seasonal\_fruit field value to see what error you get!

Also, note that because back\_of\_house::Breakfast has a private field, the struct needs to provide a public associated function that constructs an instance of Breakfast (we’ve named it summer here). If Breakfast didn’t have such a function, we couldn’t create an instance of Breakfast in eat\_at\_restaurant because we can’t set the value of the private seasonal\_fruit field in eat\_at\_restaurant.

In contrast, if we make an enum public, all of its variants are then public. We only need the pub before the enum keyword, as shown in Listing 7-10.

Filename: src/lib.rs

mod back\_of\_house {

pub enum Appetizer {

Soup,

Salad,

}

}

pub fn eat\_at\_restaurant() {

let order1 = back\_of\_house::Appetizer::Soup;

let order2 = back\_of\_house::Appetizer::Salad;

}

Listing 7-10: Designating an enum as public makes all its variants public

Because we made the Appetizer enum public, we can use the Soup and Salad variants in eat\_at\_restaurant. Enums aren’t very useful unless their variants are public; it would be annoying to have to annotate all enum variants with pub in every case, so the default for enum variants is to be public. Structs are often useful without their fields being public, so struct fields follow the general rule of everything being private by default unless annotated with pub.

There’s one more situation involving pub that we haven’t covered, and that is our last module system feature: the use keyword. We’ll cover use by itself first, and then we’ll show how to combine pub and use.

Bringing Paths into Scope with the use Keyword

It might seem like the paths we’ve written to call functions so far are inconveniently long and repetitive. For example, in Listing 7-7, whether we chose the absolute or relative path to the add\_to\_waitlist function, every time we wanted to call add\_to\_waitlist we had to specify front\_of\_house and hosting too. Fortunately, there’s a way to simplify this process. We can bring a path into a scope once and then call the items in that path as if they’re local items with the use keyword.

In Listing 7-11, we bring the crate::front\_of\_house::hosting module into the scope of the eat\_at\_restaurant function so we only have to specify hosting::add\_to\_waitlist to call the add\_to\_waitlist function in eat\_at\_restaurant.

Filename: src/lib.rs

mod front\_of\_house {

pub mod hosting {

pub fn add\_to\_waitlist() {}

}

}

use crate::front\_of\_house::hosting;

pub fn eat\_at\_restaurant() {

hosting::add\_to\_waitlist();

hosting::add\_to\_waitlist();

hosting::add\_to\_waitlist();

}

Listing 7-11: Bringing a module into scope with use

Adding use and a path in a scope is similar to creating a symbolic link in the filesystem. By adding use crate::front\_of\_house::hosting in the crate root, hosting is now a valid name in that scope, just as though the hosting module had been defined in the crate root. Paths brought into scope with use also check privacy, like any other paths.

Specifying a relative path with use is slightly different. Instead of starting from a name in the current scope, we must start the path given to use with the keyword self. Listing 7-12 shows how to specify a relative path to get the same behavior as Listing 7-11.

Filename: src/lib.rs

mod front\_of\_house {

pub mod hosting {

pub fn add\_to\_waitlist() {}

}

}

use self::front\_of\_house::hosting;

pub fn eat\_at\_restaurant() {

hosting::add\_to\_waitlist();

hosting::add\_to\_waitlist();

hosting::add\_to\_waitlist();

}

Listing 7-12: Bringing a module into scope with use and a relative path starting with self

Note that using self in this way might not be necessary in the future; it’s an inconsistency in the language that Rust developers are working on to eliminate.

Creating Idiomatic use Paths

In Listing 7-11, you might have wondered why we specified use crate::front\_of\_house::hosting and then called hosting::add\_to\_waitlist in eat\_at\_restaurant rather than specifying the use path all the way out to the add\_to\_waitlist function to achieve the same result, as in Listing 7-13.

Filename: src/lib.rs

mod front\_of\_house {

pub mod hosting {

pub fn add\_to\_waitlist() {}

}

}

use crate::front\_of\_house::hosting::add\_to\_waitlist;

pub fn eat\_at\_restaurant() {

add\_to\_waitlist();

add\_to\_waitlist();

add\_to\_waitlist();

}

Listing 7-13: Bringing the add\_to\_waitlist function into scope with use, which is unidiomatic

Although both Listing 7-11 and 7-13 accomplish the same task, Listing 7-11 is the idiomatic way to bring a function into scope with use. Bringing the function’s parent module into scope with use so we have to specify the parent module when calling the function makes it clear that the function isn’t locally defined while still minimizing repetition of the full path. The code in Listing 7-13 is unclear as to where add\_to\_waitlist is defined.

On the other hand, when bringing in structs, enums, and other items with use, it’s idiomatic to specify the full path. Listing 7-14 shows the idiomatic way to bring the standard library’s HashMap struct into the scope of a binary crate.

Filename: src/main.rs

use std::collections::HashMap;

fn main() {

let mut map = HashMap::new();

map.insert(1, 2);

}

Listing 7-14: Bringing HashMap into scope in an idiomatic way

There’s no strong reason behind this idiom: it’s just the convention that has emerged, and folks have gotten used to reading and writing Rust code this way.

The exception to this idiom is if we’re bringing two items with the same name into scope with use statements, because Rust doesn’t allow that. Listing 7-15 shows how to bring two Result types into scope that have the same name but different parent modules and how to refer to them.

Filename: src/lib.rs

use std::fmt;

use std::io;

fn function1() -> fmt::Result {

}

fn function2() -> io::Result<()> {

}

Listing 7-15: Bringing two types with the same name into the same scope requires using their parent modules.

As you can see, using the parent modules distinguishes the two Result types. If instead we specified use std::fmt::Result and use std::io::Result, we’d have two Result types in the same scope and Rust wouldn’t know which one we meant when we used Result. Try it and see what compiler error you get!

Providing New Names with the as Keyword

There’s another solution to the problem of bringing two types of the same name into the same scope with use: after the path, we can specify as and a new local name, or alias, for the type. Listing 7-16 shows another way to write the code in Listing 7-15 by renaming one of the two Result types using as.

Filename: src/lib.rs

use std::fmt::Result;

use std::io::Result as IoResult;

fn function1() -> Result {

}

fn function2() -> IoResult<()> {

}

Listing 7-16: Renaming a type when it’s brought into scope with the as keyword

In the second use statement, we chose the new name IoResult for the std::io::Result type, which won’t conflict with the Result from std::fmt that we’ve also brought into scope. Listing 7-15 and Listing 7-16 are considered idiomatic, so the choice is up to you!

Reexporting Names with pub use

When we bring a name into scope with the use keyword, the name available in the new scope is private. To enable the code that calls our code to refer to that name as if it had been defined in that code’s scope, we can combine pub and use. This technique is called reexporting because we’re bringing an item into scope but also making that item available for others to bring into their scope.

Listing 7-17 shows the code in Listing 7-11 with use in the root module changed to pub use.

Filename: src/lib.rs

mod front\_of\_house {

pub mod hosting {

pub fn add\_to\_waitlist() {}

}

}

pub use crate::front\_of\_house::hosting;

pub fn eat\_at\_restaurant() {

hosting::add\_to\_waitlist();

hosting::add\_to\_waitlist();

hosting::add\_to\_waitlist();

}

Listing 7-17: Making a name available for any code to use from a new scope with pub use

By using pub use, external code can now call the add\_to\_waitlist function using hosting::add\_to\_waitlist. If we hadn’t specified pub use, the eat\_at\_restaurant function could call hosting::add\_to\_waitlist in its scope but external code couldn’t take advantage of this new path.

This is useful when the internal structure of your code is different than the way programmers calling your code would think about the domain. For example, in this restaurant metaphor, the people running the restaurant think about “front of house” and “back of house.” But customers visiting a restaurant probably won’t think about the parts of the restaurant in those terms. With pub use, we can write our code with one structure but expose a different structure. Doing so makes our library well organized for programmers working on the library and programmers calling the library.

Using External Packages

In Chapter 2, we programmed a guessing game project that used an external package called rand to get random numbers. To use rand in our project, we added this line to Cargo.toml:

prod: check xref

Filename: Cargo.toml

[dependencies]

rand = "0.5.5"

Adding rand as a dependency in Cargo.toml tells Cargo to download the rand package and any dependencies from <https://crates.io> and make rand available to our project.

Then, to bring rand definitions into the scope of our package, we added a use line starting with the name of the package, rand, and listing the items we wanted to bring into scope. Recall that in the section “Generating a Random Number” in Chapter 2, we brought the Rng trait into scope and called the rand::thread\_rng function:

prod: check xref

use rand::Rng;

fn main() {

let secret\_number = rand::thread\_rng().gen\_range(1, 101);

}

Members of the Rust community have made many packages available at [*https://crates.io*](https://crates.io), and pulling any of them into your package involves these same steps: listing them in your package’s Cargo.toml file and using use to bring items into scope.

Note that the standard library (std) is also a crate that’s external to our package. Because the standard library is shipped with the Rust language, we don’t need to change Cargo.toml to include std. But we do need to refer to it with use to bring items from there into our package’s scope. For example, with HashMap we would use this line:

use std::collections::HashMap;

This is an absolute path starting with std, the name of the standard library crate.

Using Nested Paths to Clean Up Large use Lists

If we’re using multiple items defined in the same package or same module, listing each item on its own line can take up a lot of vertical space in our files. For example, these two use statements we had in Listing 2-4 in the Guessing Game bring items from std into scope:

Filename: src/main.rs

use std::cmp::Ordering;

use std::io;

// ---snip---

Instead, we can use nested paths to bring the same items into scope in one line. We do this by specifying the common part of the path, followed by two colons, and then curly brackets around a list of the parts of the paths that differ, as shown in Listing 7-18.

Filename: src/main.rs

use std::{cmp::Ordering, io};

// ---snip---

Listing 7-18: Specifying a nested path to bring multiple items with the same prefix into scope

In bigger programs, bringing many items into scope from the same package or module using nested paths can reduce the number of separate use statements needed by a lot!

We can use a nested path at any level in a path, which is useful when combining two use statements that share a subpath. For example, Listing 7-19 shows two use statements: one that brings std::io into scope and one that brings std::io::Write into scope:

Filename: src/lib.rs

use std::io;

use std::io::Write;

Listing 7-19: Two use statements where one is a subpath of the other

The common part of these two paths is std::io, and that’s the complete first path. To merge these two paths into one use statement, we can use self in the nested path, as shown in Listing 7-20.

Filename: src/lib.rs

use std::io::{self, Write};

Listing 7-20: Combining the paths in Listing 7-19 into one use statement

This line brings std::io and std::io::Write into scope.

The Glob Operator

If we want to bring all public items defined in a path into scope, we can specify that path followed by \*, the glob operator:

use std::collections::\*;

This use statement brings all public items defined in std::collections into the current scope. Be careful when using the glob operator! Glob can make it harder to tell what names are in scope and where a name used in your program was defined.

The glob operator is often used when testing to bring everything under test into the tests module; we’ll talk about that in the section “How to Write Tests” in Chapter 11. The glob operator is also sometimes used as part of the prelude pattern: see the standard library documentation at <https://doc.rust-lang.org/stable/std/prelude/index.html#other-preludes>/ for more information on that pattern.

prod: check xref

Separating Modules into Different Files

So far, all the examples in this chapter defined multiple modules in one file. When modules get large, you might want to move their definitions to a separate file to make the code easier to navigate.

For example, let’s use the code in Listing 7-17 and move the front\_of\_house module to its own file src/front\_of\_house.rs by changing the crate root file so it contains the code shown in Listing 7-21. In this case, the crate root file is src/lib.rs, but this procedure also works with binary crates whose crate root file is src/main.rs.

Filename: src/lib.rs

mod front\_of\_house;

pub use crate::front\_of\_house::hosting;

pub fn eat\_at\_restaurant() {

hosting::add\_to\_waitlist();

hosting::add\_to\_waitlist();

hosting::add\_to\_waitlist();

}

Listing 7-21: Declaring the front\_of\_house module whose body will be in src/front\_of\_house.rs

And src/front\_of\_house.rs gets the definitions from the body of the front\_of\_house module, as shown in Listing 7-22.

Filename: src/front\_of\_house.rs

pub mod hosting {

pub fn add\_to\_waitlist() {}

}

Listing 7-22: Definitions inside the front\_of\_house module in src/front\_of\_house.rs

Using a semicolon after mod front\_of\_house rather than using a block tells Rust to load the contents of the module from another file with the same name as the module. To continue with our example and extract the hosting module to its own file as well, we change src/front\_of\_house.rs to contain only the declaration of the hosting module:

Filename: src/front\_of\_house.rs

pub mod hosting;

Then we create a src/front\_of\_house directory and a file src/front\_of\_house/hosting.rs to contain the definitions made in the hosting module:

Filename: src/front\_of\_house/hosting.rs

pub fn add\_to\_waitlist() {}

The module tree remains the same, and the function calls in eat\_at\_restaurant will work without any modification, even though the definitions live in different files. This lets you move modules to new files as they grow in size.

Note that the pub use crate::front\_of\_house::hosting statement in src/lib.rs also hasn’t changed, nor does use have any impact on what files are compiled as part of the crate. The mod keyword declares modules, and Rust looks in a file with the same name as the module for the code that goes into that module.

Summary

Rust lets you organize your packages into crates and your crates into modules so you can refer to items defined in one module from another module. You can do this by specifying absolute or relative paths. These paths can be brought into scope with a use statement so you can use a shorter path for multiple uses of the item in that scope. Module code is private by default, but you can make definitions public by adding the pub keyword.

In the next chapter, we’ll look at some collection data structures in the standard library that you can use in your neatly organized code.