Managing Growing Projects with Packages, Crates, and Modules

As you write large programs, organizing your code will become increasingly important. 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 should 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 comprising a set of interrelated packages that evolve together, Cargo provides *workspaces*, which we'll cover in the "Cargo Workspaces" section in Chapter 14.

We'll also discuss encapsulating implementation details, which lets you reuse code at a higher level: once you've implemented an operation, other code can call your code via its public interface without having to know how the implementation works. 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. This is another way to limit the amount of detail you have to keep in your head.

A related concept is scope: the nested context in which code is written has 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, which details are private, and what names are in each scope in your programs. These features, sometimes collectively referred to as the *module system*, 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 the smallest amount of code that the Rust compiler considers at a time. Even if you run `rustc` rather than `cargo` and pass a single source code file (as we did all the way back in the "Writing and Running a Rust Program" section of Chapter 1), the compiler considers that file to be a crate. Crates can contain modules, and the modules may be defined in other files that get compiled with the crate, as we'll see in the coming sections.

A crate can come in one of two forms: a binary crate or a library crate. *Binary crates* are programs you can compile to an executable that you can run, such as a command-line program or a server. Each must have a function called `main` that defines what happens when the executable runs. All the crates we've created so far have been binary crates.

Library crates don't have a `main` function, and they don't compile to an executable. Instead, they define functionality intended to be shared with multiple projects. For example, the `rand` crate we used in Chapter 2 provides functionality that generates random numbers. Most of the time when Rustaceans say "crate", they mean library crate, and they use "crate" interchangeably with the general programming concept of a "library". IMP

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 "Defining Modules to Control Scope and Privacy" section).

A *package* is a bundle of one or more crates that provides a set of functionality. A package contains a *Cargo.toml* file that describes how to build those crates. Cargo is actually a package that contains the binary crate for the command-line tool you've been using to build your code. The Cargo package also contains a library crate that the binary crate depends on. Other projects can depend on the Cargo library crate to use the same logic the Cargo command-line tool uses.

A crate can come in one of two forms: a binary crate or a library crate. A package can contain as many binary crates as you like, but at most only one library crate. A package must contain at least one crate, whether that's a library or binary crate. V IMP

Let's walk through what happens when we create a package. First we enter the command `cargo new my-project`:

```
$ cargo new my-project
     Created binary (application) `my-project` package
$ ls my-project
Cargo.toml
src
$ ls my-project/src
main.rs
```

After we run `cargo new my-project`, we use `ls` to see what Cargo creates. In the project directory, there's a *Cargo.toml* file, giving us a package. There's also a *src* directory that contains *main.rs*. Open *Cargo.toml* in your text editor, and note there's no mention of *src/main.rs*. 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 binary and a library, 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.

Defining Modules to Control Scope and Privacy

In this section, we'll talk about modules and other parts of the module system, namely *paths*, which 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 the `as` keyword, external packages, and the glob operator.

Modules Cheat Sheet

Before we get to the details of modules and paths, here we provide a quick reference on how modules, paths, the `use` keyword, and the `pub` keyword work in the compiler, and how most developers organize their code. We'll be going through examples of each of these rules throughout this chapter, but this is a great place to refer to as a reminder of how modules work.

- **Start from the crate root**: When compiling a crate, the compiler first looks in the crate root file (usually *src/lib.rs* for a library crate or *src/main.rs* for a binary crate) for code to compile.
- **Declaring modules**: In the crate root file, you can declare new modules; say you declare a "garden" module with `mod garden;`. The compiler will look for the module's code in these places:
  - Inline, within curly brackets that replace the semicolon following `mod garden`
  - In the file *src/garden.rs*
  - In the file *src/garden/mod.rs*
- **Declaring submodules**: In any file other than the crate root, you can declare submodules. For example, you might declare `mod vegetables;` in *src/garden.rs*. The compiler will look for the submodule's code within the directory named for the parent module in these places:
  - Inline, directly following `mod vegetables`, within curly brackets instead of the semicolon
  - In the file *src/garden/vegetables.rs*
  - In the file *src/garden/vegetables/mod.rs*
- **Paths to code in modules**: Once a module is part of your crate, you can refer to code in that module from anywhere else in that same crate, as long as the privacy rules allow, using the path to the code. For example, an `Asparagus` type in the garden vegetables module would be found at `crate::garden::vegetables::Asparagus`.
- **Private vs. public**: Code within a module is private from its parent modules by default. To make a module public, declare it with `pub mod` instead of `mod`. To make items within a

public module public as well, use `pub` before their declarations.

- **The `use` keyword**: Within a scope, the `use` keyword creates shortcuts to items to reduce repetition of long paths. In any scope that can refer to `crate::garden::vegetables::Asparagus`, you can create a shortcut with `use crate::garden::vegetables::Asparagus;` and from then on you only need to write `Asparagus` to make use of that type in the scope.

Here, we create a binary crate named `backyard` that illustrates these rules. The crate's directory, also named `backyard`, contains these files and directories:

```
backyard
├── Cargo.lock
├── Cargo.toml
└── src
    ├── garden
    │   └── vegetables.rs
    ├── garden.rs
    └── main.rs
```

The crate root file in this case is *src/main.rs*, and it contains:

Filename: src/main.rs

```
use crate::garden::vegetables::Asparagus;

pub mod garden;

fn main() {
    let plant = Asparagus {};
    println!("I'm growing {plant:?}!");
}
```

The `pub mod garden;` line tells the compiler to include the code it finds in *src/garden.rs*, which is:

Filename: src/garden.rs

```
pub mod vegetables;
```

Here, `pub mod vegetables;` means the code in *src/garden/vegetables.rs* is included too. That code is:

```
#[derive(Debug)]
pub struct Asparagus {}
```

Now let's get into the details of these rules and demonstrate them in action!

Grouping Related Code in Modules

Modules let us organize code within a crate for readability and easy reuse. Modules also allow us to control the *privacy* of items because code within a module is private by default. Private items are internal implementation details not available for outside use. We can choose to make modules and the items within them public, which exposes them to allow external code to use and depend on them.

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 the implementation of a restaurant.

In the restaurant industry, some parts of a restaurant are referred to as *front of house* and others as *back of house*. Front of house is where customers are; this encompasses where the hosts seat customers, servers take orders and payment, and bartenders make drinks. Back of house is where the chefs and cooks work in the kitchen, dishwashers clean up, and managers do administrative work.

To structure our crate in this way, we can organize its functions into nested modules. Create a new library named `restaurant` by running `cargo new restaurant --lib`. Then enter the code in Listing 7-1 into *src/lib.rs* to define some modules and function signatures; this code is the front of house section.

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 with the `mod` keyword followed by the name of the module (in this case, `front_of_house`). The body of the module then goes inside curly brackets. Inside modules, we can place 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, and—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 can navigate the code based on the groups rather than having to read through all the definitions, making it easier to find the definitions relevant to them. 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 other modules; for example, `hosting` nests inside `front_of_house`. The tree also shows that some modules are *siblings*, meaning they're defined in the same module; `hosting` and `serving` are siblings defined within `front_of_house`. 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. To call a function, we need to know its path.

A path can take two forms:

- An *absolute path* is the full path starting from a crate root; for code from an external crate, the absolute path begins with the crate name, and for code from the current crate, it starts with the 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 (`::`).

Returning to Listing 7-1, say we want to call the `add_to_waitlist` function. This is the same as asking: what's the path of the `add_to_waitlist` function? Listing 7-3 contains Listing 7-1 with some of the modules and functions removed.

We'll show two ways to call the `add_to_waitlist` function from a new function, `eat_at_restaurant`, defined in the crate root. These paths are correct, but there's another problem remaining that will prevent this example from compiling as is. We'll explain why in a bit.

The `eat_at_restaurant` function is part of our library crate's public API, so we mark it with the `pub` keyword. In the "Exposing Paths with the `pub` Keyword" section, we'll go into more detail about `pub`.

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. We then include each of the successive modules until we make our way to `add_to_waitlist`. You can imagine a filesystem with the same structure: 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 module 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, and it depends 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 moved 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. Our preference in general is to specify absolute paths because it's more likely we'll want 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 errors we get are 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();
  |                            ^^^^^^^  --------------- function `add_to_waitlist`
is not publicly re-exported
  |                            |
  |                            private module
  |
note: the module `hosting` is defined here
 --> src/lib.rs:2:5
  |
2 |     mod hosting {
  |     ^^^^^^^^^^^

error[E0603]: module `hosting` is private
  --> src/lib.rs:12:21
   |
12 |     front_of_house::hosting::add_to_waitlist();
   |                     ^^^^^^^  --------------- function `add_to_waitlist` is
not publicly re-exported
   |                     |
   |                     private module
   |
note: the module `hosting` is defined here
  --> src/lib.rs:2:5
   |
2  |     mod hosting {
   |     ^^^^^^^^^^^

For more information about this error, try `rustc --explain E0603`.
error: could not compile `restaurant` (lib) due to 2 previous errors
```

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. In Rust, all items (functions, methods, structs, enums, modules, and constants) are private to parent modules by default. If you want to make an item like a function or struct private, you put it in a module.

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. This is because child modules wrap and hide their implementation details, but the child modules can see the context in which they're defined. To continue with our metaphor, think of the privacy rules as being 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 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. However, Rust does give you the option to expose inner parts of child modules' code to outer ancestor modules by using the `pub` keyword to make an item public.

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 compiler errors, 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();
  |                                     ^^^^^^^^^^^^^^^ private function
  |
note: the function `add_to_waitlist` is defined here
 --> src/lib.rs:3:9
  |
3 |         fn add_to_waitlist() {}
  |         ^^^^^^^^^^^^^^^^^^^^

error[E0603]: function `add_to_waitlist` is private
  --> src/lib.rs:12:30
   |
12 |     front_of_house::hosting::add_to_waitlist();
   |                              ^^^^^^^^^^^^^^^ private function
   |
note: the function `add_to_waitlist` is defined here
  --> src/lib.rs:3:9
   |
3  |         fn add_to_waitlist() {}
   |         ^^^^^^^^^^^^^^^^^^^^

For more information about this error, try `rustc --explain E0603`.
error: could not compile `restaurant` (lib) due to 2 previous errors
```

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, not access its inner code. Because modules are containers, there's not much we can do by only making the module public; we need to go further and choose to make one or more of the items within the module public as well.

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! To see why adding the `pub` keyword lets us use these paths in `eat_at_restaurant` with respect to the privacy rules, let's look at the absolute and the relative paths.

In the absolute path, we start with `crate`, the root of our crate's module tree. The `front_of_house` module is defined in the crate root. While `front_of_house` isn't public, 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!

If you plan on sharing your library crate so other projects can use your code, your public API is your contract with users of your crate that determines how they can interact with your code. There are many considerations around managing changes to your public API to make it easier for people to depend on your crate. These considerations are out of the scope of this book; if you're interested in this topic, see The Rust API Guidelines.

Best Practices for Packages with a Binary and a Library

We mentioned that a package can contain both a *src/main.rs* binary crate root as well as a *src/lib.rs* library crate root, and both crates will have the package name by default. Typically, packages with this pattern of containing both a library and a binary crate will have just enough code in the binary crate to start an executable that calls code within the library crate. This lets other projects benefit from most of the functionality that the package provides because the library crate's code can be shared.

The module tree should be defined in *src/lib.rs*. Then, any public items can be used in the binary crate by starting paths with the name of the package. The binary crate becomes a user of the library crate just like a completely external crate would use the library crate: it can only use the public API. This helps you design a good API; not only are you the author, you're also a client!

In Chapter 12, we'll demonstrate this organizational practice with a command-line program that will contain both a binary crate and a library crate.

Starting Relative Paths with `super`

We can construct relative paths that begin in the parent module, rather than the current module or the crate root, by using `super` at the start of the path. This is like starting a filesystem path with the `..` syntax. Using `super` allows us to reference an item that we know is in the parent module, which can make rearranging the module tree easier when the module is closely related to the parent but the parent might be moved elsewhere in the module tree someday.

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` defined in the `back_of_house` module calls the function `deliver_order` defined in the parent module by specifying the path to `deliver_order`, starting with `super`.

Filename: src/lib.rs

```
fn deliver_order() {}

mod back_of_house {
    fn fix_incorrect_order() {
        cook_order();
        super::deliver_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 `deliver_order` and find it. Success! We think the `back_of_house` module and the `deliver_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 to the usage of `pub` with structs and enums. 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 couldn'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

Having to write out the paths to call functions can feel inconvenient and repetitive. 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 create a shortcut to a path with the `use` keyword once, and then use the shorter name everywhere else in the scope.

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();
}
```

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.

Note that `use` only creates the shortcut for the particular scope in which the `use` occurs. Listing 7-12 moves the `eat_at_restaurant` function into a new child module named `customer`, which is then a different scope than the `use` statement, so the function body won't compile.

Filename: src/lib.rs

```
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
    }
}

use crate::front_of_house::hosting;

mod customer {
    pub fn eat_at_restaurant() {
        hosting::add_to_waitlist();
    }
}
```

Listing 7-12: A `use` statement only applies in the scope it's in

The compiler error shows that the shortcut no longer applies within the `customer` module:

```
$ cargo build
   Compiling restaurant v0.1.0 (file:///projects/restaurant)
error[E0433]: failed to resolve: use of undeclared crate or module `hosting`
  --> src/lib.rs:11:9
   |
11 |         hosting::add_to_waitlist();
   |         ^^^^^^^ use of undeclared crate or module `hosting`
   |
help: consider importing this module through its public re-export
   |
10 +     use crate::hosting;
   |

warning: unused import: `crate::front_of_house::hosting`
 --> src/lib.rs:7:5
  |
7 | use crate::front_of_house::hosting;
  |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  |
  = note: `#[warn(unused_imports)]` on by default

For more information about this error, try `rustc --explain E0433`.
warning: `restaurant` (lib) generated 1 warning
error: could not compile `restaurant` (lib) due to 1 previous error; 1 warning
emitted
```

Notice there's also a warning that the `use` is no longer used in its scope! To fix this problem, move the `use` within the `customer` module too, or reference the shortcut in the parent module with `super::hosting` within the child `customer` module.

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();
}
```

Listing 7-13: Bringing the `add_to_waitlist` function into scope with `use`, which is unidiomatic

Although both Listing 7-11 and Listing 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` means we have to specify the parent module when calling the function. Specifying 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 {
    // --snip--
}

fn function2() -> io::Result<()> {
    // --snip--
}
```

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`.

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 {
    // --snip--
}

fn function2() -> IoResult<()> {
    // --snip--
}
```

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!

Re-exporting 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 *re-exporting* 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();
}
```

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

Before this change, external code would have to call the `add_to_waitlist` function by using the path `restaurant::front_of_house::hosting::add_to_waitlist()`, which also would have required the `front_of_house` module to be marked as `pub`. Now that this `pub use` has re-exported the `hosting` module from the root module, external code can use the path `restaurant::hosting::add_to_waitlist()` instead.

Re-exporting is useful when the internal structure of your code is different from how 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. We'll look at another example of `pub use` and how it affects your crate's documentation in the "Exporting a Convenient Public API with `pub use`" section of Chapter 14.

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*:

Filename: Cargo.toml

```
rand = "0.8.5"
```

Adding `rand` as a dependency in *Cargo.toml* tells Cargo to download the `rand` package and any dependencies from 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 crate, `rand`, and listed the items we wanted to bring into scope. Recall that in the "Generating a Random Number" section in Chapter 2, we brought the `Rng` trait into scope and called the `rand::thread_rng` function:

```
use rand::Rng;

fn main() {
    let secret_number = rand::thread_rng().gen_range(1..=100);
}
```

Members of the Rust community have made many packages available at 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 from their crates into scope.

Note that the standard `std` library 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 crate 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 the guessing game in Listing 2-4 bring items from `std` into scope:

Filename: src/main.rs

```
// --snip--
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

```
// --snip--
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 crate 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 "How to Write Tests" section in Chapter 11. The glob operator is also sometimes used as part of the prelude pattern: see the standard library documentation for more information on that pattern.

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 start from the code in Listing 7-17 that had multiple restaurant modules. We'll extract modules into files instead of having all the modules defined in the crate root file. 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*.

First we'll extract the `front_of_house` module to its own file. Remove the code inside the curly brackets for the `front_of_house` module, leaving only the `mod front_of_house;` declaration, so that *src/lib.rs* contains the code shown in Listing 7-21. Note that this won't compile until we create the *src/front_of_house.rs* file in Listing 7-22.

Filename: src/lib.rs

```
mod front_of_house;

pub use crate::front_of_house::hosting;

pub fn eat_at_restaurant() {
    hosting::add_to_waitlist();
}
```

Listing 7-21: Declaring the `front_of_house` module whose body will be in *src/front_of_house.rs*

Next, place the code that was in the curly brackets into a new file named *src/front_of_house.rs*, as shown in Listing 7-22. The compiler knows to look in this file because it came across the module declaration in the crate root with the name `front_of_house`.

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*

Note that you only need to load a file using a `mod` declaration *once* in your module tree. Once the compiler knows the file is part of the project (and knows where in the module tree the code

resides because of where you've put the `mod` statement), other files in your project should refer to the loaded file's code using a path to where it was declared, as covered in the "Paths for Referring to an Item in the Module Tree" section. In other words, `mod` is *not* an "include" operation that you may have seen in other programming languages.

Next, we'll extract the `hosting` module to its own file. The process is a bit different because `hosting` is a child module of `front_of_house`, not of the root module. We'll place the file for `hosting` in a new directory that will be named for its ancestors in the module tree, in this case *src/front_of_house*.

To start moving `hosting`, 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 *hosting.rs* file to contain the definitions made in the `hosting` module:

Filename: src/front_of_house/hosting.rs

```
pub fn add_to_waitlist() {}
```

If we instead put *hosting.rs* in the *src* directory, the compiler would expect the *hosting.rs* code to be in a `hosting` module declared in the crate root, and not declared as a child of the `front_of_house` module. The compiler's rules for which files to check for which modules' code mean the directories and files more closely match the module tree.

Alternate File Paths

So far we've covered the most idiomatic file paths the Rust compiler uses, but Rust also supports an older style of file path. For a module named `front_of_house` declared in the crate root, the compiler will look for the module's code in:

- *src/front_of_house.rs* (what we covered)
- *src/front_of_house/mod.rs* (older style, still supported path)

For a module named `hosting` that is a submodule of `front_of_house`, the compiler will look for the module's code in:

- *src/front_of_house/hosting.rs* (what we covered)

- *src/front_of_house/hosting/mod.rs* (older style, still supported path)

If you use both styles for the same module, you'll get a compiler error. Using a mix of both styles for different modules in the same project is allowed, but might be confusing for people navigating your project.

The main downside to the style that uses files named *mod.rs* is that your project can end up with many files named *mod.rs*, which can get confusing when you have them open in your editor at the same time.

We've moved each module's code to a separate file, and the module tree remains the same. The function calls in `eat_at_restaurant` will work without any modification, even though the definitions live in different files. This technique 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 split a package into multiple crates and a crate 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.