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.