Lec 5: Modules

Jack Duvall

Don't Put Everything In One File!

- Easier to read short files
- Allows code reuse
- Every other modern language has modules (C#, Go, Java, Python, Typescript, etc.)

What Is A Module?

- "A bag of things that go together"
- Containing any or all of:
  - Structs + Enums
  - Types + Traits
  - Functions
  - Constants
  - Static members
  - Other modules!

A Module Defines A Namespace

- Function in current module: no prefix
- Functions in a different module: `module_name::function_name`
  - Can also have `long::path::to::module_name::function_name`
  - Can prepend with `::` to have an absolute path
- More examples to come

Crates

What Is A Crate?

- Crate: highest level module
- May have modules inside it
- May contain multiple Rust files, as well as associated data
- Similar to packages in other languages

Types of Crates

- Binary
- Library

Binary Crates

- Results in executable you can run
- Crate root: `src/main.rs`
- Has `main` function in that file
- Cannot have integration tests

Library Crates

- Results in something you can link against
  - Link: "I can use some of this code without recompiling"
- Crate root: `src/lib.rs`
- Does not need a `main` function
- Can have integration tests

When To Use Binary vs Library Crates

- Library: Almost Always
  - Can lead to nicer test structuring
  - Easier to reuse code
- Binary: When You Can't Use A Library
  - Often just a wrapper around a core library

Magic Incantations

- `cargo new <name_of_crate>`
- Or, manually:
  - Create a `Cargo.toml` with the appropriate fields
  - Create a `src` directory
  - Create `src/main.rs` for a binary crate
  - Create `src/lib.rs` for a library crate
  - Exclude the `target/` directory in your `.gitignore`
- Very opinionated, names must match exactly!

A Sample `Cargo.toml`

```toml
[package]

name = "foobar"

version = "0.1.0"

authors = ["Jack Duvall <jrduvall@andrew.cmu.edu>"]

edition = "2021"
```

Modules And Files

Modules inline with text

```
// In `src/lib.rs`:
mod foo { // Now the `foo` module exists, in a
    …     // separate namespace from the rest of the
file
}
```

Directory Structure *Is* Module Structure

- If modules `bar`, `bar::baz`, and `bar::qux` are all modules corresponding to files:

```
src/
├──lib.rs
├──bar/
   ├──mod.rs (bar)
   ├──baz.rs (bar::baz)
   ├──qux.rs (bar::qux)
```

Alternatively...

- We can name directories the same as a file for submodules
  - I don't recommend this since what if you rename just one accidentally??

```
src/
├──lib.rs
├──bar.rs (bar)
├──bar/
   ├──baz.rs (bar::baz)
   ├──quz.rs (bar::qux)
```

Using File Modules

```
// In `src/lib.rs`:
mod bar;
// In `src/bar/mod.rs`:
mod baz;
mod qux;
```

But I Don't Want My Directories To Represent My Module Structure!

- First of all, why??
- But also, you can do that: (https://doc.rust-lang.org/reference/items/modules.html)

```
#[path = "thread_files"]
mod thread {
    // Load the `local_data` module from `thread_files/tls.rs` relative to
    // this source file's directory.
    #[path = "tls.rs"]
    mod local_data;
}
```

Visibility

- By default, everything inside a module is private to that module!
- Can make things visible to other modules using the `pub` keyword:

```
// In `src/lib.rs`
mod foo {
    pub fn foo() -> usize { 42 }
}
// Calling `foo::foo()` works now! Wouldn't work without
`pub`
```

Visibility On Other Things

- Structs: fields private by default, need to selectively make them `pub` too
- Enums: all variants public if the enum is `pub`
- Functions: if the function is `pub`, all arguments type and return type must also be `pub`
- Traits: all members public if the trait is `pub`
- Modules: only `pub` things inside the module are public

Updating File Module Example

```rust
// In `src/lib.rs`:
pub mod bar; // Now `bar` is accessible in this file
// In `src/bar/mod.rs`:
pub mod baz; // Now `bar::baz` is accessible in `src/lib.rs`
mod qux;     // Maybe we want `qux` to stay private to `bar`! We can do that
```

`use`ing Modules

The `use` Keyword: Basic Usage

- Typing out full module name every time is hard to read
- Better way:

```
// In `src/lib.rs`:

use bar::baz::bar_function;

// Now we can just type `bar_function` and it'll use
`bar::baz::bar_function`!
```

The `use` Keyword: Multiple Things

```
// In `src/lib.rs`

use bar::{bar_function, baz::baz_function};

// Now we can call `bar_function` and `baz_function`
without the module names!

use foo::*;

// Now we can all any public function from `foo`! Or
`foo`'s modules, types, etc. as if they were in our own
namespace
```

The `use` Keyword: `self`, `as`

```
// In `src/lib.rs`:

use std::io::{self, Result as IoResult};

// Now we can call `std::io::method_name` as just `io::method_name`, and refer to an `std::io::Result` as an `IoResult`!
```

Module Path Syntax

In a path that looks like `mod1::mod2::mod3::thing`:

- First, we'll see if there's a module called `mod1` that's a submodule of the current module
- If so, we'll try to see if it has a submodule called `mod2` that has a submodule called `mod3` which has something called `thing` inside it
- If not, we'll try to see if there's a *crate* called `mod1` that (blah blah)
    - To force the use of crates, prefix the path with `::`

The `crate` Keyword

- `crate`: used in module paths to start from the base of the crate, not the base namespace
- `crate::bar::baz` is the same in `src/lib.rs` and `src/bar/qux.rs`, but just `bar::baz` is not

```
// In `src/bar/qux.rs`:

use crate::bar::baz::*; // Uses things from bar::baz

use bar::baz::*; // Uses things from
`bar::qux::bar::baz`, not what you wanted probably!
```

`pub` and `use` together: Re-Exports

- Often, key types are scattered throughout modules
- Pain to include them all manually, better to have a "prelude" that includes them all for you and re-exports them:

```
// In `src/bar/prelude.rs`:

pub use crate::bar::{bar_function, baz::baz_function};

// Now, doing `use bar::prelude::*;` in `src/lib.rs`
will give us `bar_function` and `baz_function`
```

More Complex Visibility: `pub` With Parens

Visible: "A module or any descendant module can reference this item"

`pub` by default: visible to any external module

`pub(crate)`: visible to any other module in the crate

`pub(super)`: visible to the parent module

`pub(in path)` where path is a module path starting with `crate`, `super`, or `self`: visible to that module

See https://doc.rust-lang.org/reference/visibility-and-privacy.html for more details

Using Crates

Cargo Is Your Friend

- In your `Cargo.toml`:

```
[dependencies]

clap = "2.33" # Remote crates just need version number

test_utils = { path = " ../test_utils/" } # Local crates can have a path specified

regex = { git = "https://github.com/rust-lang/regex", branch = "next" } # Can also specify remote crates from a git repository
```

Same Usage As Before!

```
// In any `*.rs` file in the crate:
type App = clap::App;
fn main() { test_utils::run_tests(App); }
// Or:
use clap::App;
use test_utils::run_tests;
fn main() { run_tests(App); }
```

Aside: SemVer

- SemVer: "Semantic Versioning".
- `<major_version>.<minor_version>.<patch_number>`
- Major Version:
    - Completely different API/functionality, considerable effort to upgrade from previous major version
- Minor Version:
    - Some API/functionality has changed, but probably not enough that most people need to rewrite their code.
- Patch Version:
    - Hardly any API/functionality changes, except for bug fixes

Other Version Number Tricks

- Carat Requirements: "Don't upgrade past the next big version"
  - `"^1.2.3" := ⩾1.2.3, <2.0.0`
  - `"^0.2"   := ⩾0.2.0, <0.3.0`
- Tilde Requirements: "Only allow smaller changes"
  - `"~1.2.3" := ⩾1.2.3, <1.3.0`
  - `"~1.2"   := ⩾1.2.0, <1.3.0`
  - `"~1"     := ⩾1.0.0, <2.0.0`
- Wildcard Requirements: "Any number in that spot is allowed"
  - `"1.*"    := ⩾1.0.0, <2.0.0`
  - `"1.2.*"  := ⩾1.2.0, <1.3.0`

This Doesn't Nearly Cover Everything

- See https://doc.rust-lang.org/cargo/reference/specifying-dependencies.html for the full specification about how you can specify dependencies in Cargo.toml.
- Mostly just use `<major>.<minor>` versions, everything else is more rare

Aside: The Orphan Rule

- "Can't implement foreign traits on foreign types"
- Foreign = not in this *crate*. Types/traits from different modules inside a crate are OK
- Why not? So that there is only ever one trait implementation for a given type: "coherence"

Getting Around The Orphan Rule: Local Type

```
use other_crate::{ForeignTrait, ForeignType};
struct LocalType(ForeignType);
impl ForeignTrait for LocalType {
  // Use self.0 to get at the ForeignType value
}
```

Features

How Can We Split Out Functionality?

Scenario: you want to implement traits exported by a large library

- Don't want to force all users to import the library
- Can't split into two crates, since some types must be private, plus it's not feasible to implement foreign traits on foreign types

Are we just stuck?

Scenario Code

```
use serde::Deserialize;
#[derive(Deserialize)]
struct PrivateState;
#[derive(Deserialize)]
pub struct PublicData {
  state: PrivateState,
}
```

Features To The Rescue!

- Features let you conditionally compile/declare things
  - Structs, enums, constants, traits, functions entire modules!

```
#[cfg(feature = "serde_impl")]
mod serde_impl {
  use serde;
  // trait impls
}
```

Declaring What Features Exist

```
// In Cargo.toml for my-library
[features]
serde_impl = ["serde", "some_other_feature"]
some_other_feature = []
[dependencies]
serde = "1.0"
```

Enabling Features On Dependencies

```
// In Cargo.toml for my-binary
[dependencies]
my-library = {
    version = "0.1",
    features = ["serde_impl"],
}
```

TOML Syntax For Lots Of Features

```
[dependencies.windows]
version = "0.29.0"
features = [
     "alloc",
     "Win32_Foundation",
     "Win32_Security",
     "Win32_System_Threading",
     "Win32_System_Console",
     "Win32_System_Pipes",
     "Win32_System_SystemServices",
     "Win32_System_WindowsProgramming",
     "Win32_System_IO",
     "Win32_Storage_FileSystem",
]
```

Example Time!

A Sample Rust Project

https://github.com/duvallj/tungstenite_testings/blob/master/Cargo.toml

Documentation

Documentation Comments

- Regular comments: `//` for single-line or `/* … */` for multi-line
- Doc comments: `///`, `/** … */`, `//!`, and `/*! … */`
  - Multiple consecutive single-line comments considered as an entire block
- `///` and `/** … */`: document the following item
- `//!` and `/*! … */`: document the "enclosing" item
  - Often used for preface documentation for an entire module, in addition to documentation about each item

Rustdoc Is Your Friend

- All doc comments support Markdown
- Building documentation: `cargo doc` (that's it!)
- Generated documentation is very fancy

Documentation Tests

- Rust code in doc comments are considered tests
- Run the same way as other tests: `cargo test`
- Combining example code with tests is a super neat idea!

````
/// This function doubles a number
/// ```rust
/// assert_eq!(mycrate::double(42), 84);
/// ```
pub fn double(x: usize) { 2 * x }
````

Example Time 2!

https://docs.rs/rand/latest/rand/

- When you publish something to crates.io, the corresponding docs.rs page is generated for you!

Homework

"Midterm" Assignment!

- Decide what project you want to do for the final
- Short description of project goals, external crates you plan on using
- Turn in on Gradescope by 03/02

Backup: Cargo Workspaces

target/ Is The New node_modules/

- Good practice: split out code into separate crates when possible
- Tightly-dependent crates will have similar dependencies
- Each cargo project will compile and download these to separate target/ directories!

Solution: Workspaces

```
// In a main Cargo.toml:
[workspace]
members = [
    "bin_crate",  // these are names of directories with
    "lib_crate1", // Cargo.toml files, don't have to match
    "lib_crate2", // the name of individual crates
]
```

Notes About Workspaces

- Crates are still logically separate
- They share dependencies for compatibility and compilation speed
- `cargo test` tests all crates in a workspace
- `cargo publish` must still be done on each crate separately
- See https://doc.rust-lang.org/stable/book/ch14-03-cargo-workspaces.html for official docs

Backup: Local Patches

Scenario: I Want To Contribute to FOSS!

But: reproducing a bug requires going through a dependency chain:

```
my_affected_crate v0.1.0
→ dependency v0.5.11
→ buggy_crate v0.12.0
```

Solution: Cargo.toml Patch Section

```
// In my_affected_crate's Cargo.toml:
[dependencies]
dependency = "0.5.11"
[patch.crates-io]
buggy_crate = { path = "../local_buggy_crate" }
```

Can I Override Other Sources?

Yes!

```
[patch.crates-io]

foo = { git = 'https://github.com/example/foo' }

bar = { path = 'my/local/bar' }

[dependencies.baz]

git = 'https://github.com/example/baz'

[patch.'https://github.com/example/baz']

baz = { git = 'https://github.com/example/patched-baz', branch = 'my-branch' }
```

Official Documentation Explains Further

https://doc.rust-lang.org/cargo/reference/overriding-dependencies.html

Backup: Supertraits

What Are Supertraits?

- Supertrait: "Trait bound on implementing a trait"

```
trait BaseTrait {}

trait SuperTrait : BaseTrait {}

impl SuperTrait for () {

  // will fail to compile unless we also impl BaseTrait
for ()

}
```

Built-in Supertraits

- `Copy: Clone`
- `Display: Debug`
- `Eq: PartialEq`
- `Ord: PartialOrd`

Why Have Supertraits?

- Inheritance-like things are good sometimes, and we'd like to support that pattern

An Extension of this: Extension Traits

- Main functionality is in one trait
- Extension trait: "automatically add new functionality for anything implementing the previous trait"
- See: Future, FutureExt in the `future` crate

```
trait FutureExt : Future {
  // implementation uses methods from Future
}
impl<T: Future> FutureExt for T {}
```