CS 241: Systems Programming
Lecture 17. Modules

Fall 2025

Prof. Stephen Checkoway

Packages and crates and modules, oh my!

Rust code is organized into packages, crates, and modules

Packages are the largest* unit of organizing code

- Created with cargo new or cargo init
- Composed of crates
- At most one library crate (src/lib.rs)
- Zero or more binary crates (src/main.rs or src/bin/*.rs)

* Technically, there are also workspaces composed of multiple packages

Crates

The smallest unit of code compiled by rustc

- Each crate has a “root”
  - src/lib.rs for library crates in packages
  - src/main.rs or src/bin/*.rs for binary crates in packages
  - the file you pass to rustc when compiling by hand rather than cargo
- Crates form a tree of modules starting at the root
- Crates may depend on library crates
  - E.g., rand, clap, and colored

Rust developers usually mean a library crate when they say crate

https://crates.io is the central repository of crates

Modules

Crates are composed of modules

Two types of modules

- Inline modules
- File modules (I don’t know if these have a real name)

Modules form a tree whereby modules may contain submodules

Inline modules

A module defined inside another module (or the crate root) using mod { }

```
mod foo {
    struct SomeType {
        // ...
    }

    fn bar() {
        println!("Function bar() in module foo")
    }
}
```

Test modules

The most common use for an inline module is unit tests

```
#[cfg(test)]
mod test {
    #[test]
    fn test_something() { }
}
```

The `#[cfg(test)]` annotation tells the compiler that the test module should not be compiled unless it is compiling unit tests (e.g., via cargo test)

The `#[test]` annotation on a function indicates it is a unit test that will be run by cargo test

Most modules are files

Declare a module foo with

```
mod foo;
```

Where the code for foo lives depends on if this module is declared in the crate root (lib.rs/main.rs) or in a module (as a submodule)

Modules declared in the crate root

```
// main.rs
mod process;

fn main() {
    let proc = process::Process::new();
    println!("{proc:?}");
}
```

Modules declared in the crate root

```rust
// main.rs
mod process;

fn main() {
    let proc = process::Process::new();
    println!("{proc:?}");
}
```

```rust
// process.rs
#[derive(Debug)]
pub struct Process {
    // ...
}

impl Process {
    pub fn new() -> Self {
        Self {
            // ...
        }
    }
}
```

Modules declared in the crate root

```rust
// main.rs
mod process;

fn main() {
    let proc = process::Process::new();
    println!("{proc:?}");
}
```

```
src
├── main.rs
└── process.rs
```

Same directory as main.rs

```rust
// process.rs
#[derive(Debug)]
pub struct Process {
    // ...
}

impl Process {
    pub fn new() -> Self {
        Self {
            // ...
        }
    }
}
```

Submodules

```
// main.rs
mod process;

fn main() {
    let proc = process::Process::new();
    println!("{proc:?}");
}

// process.rs


#[derive(Debug)]
pub struct Process {
    // ...
}

// ...
```

Submodules

```
// main.rs
mod process;

fn main() {
    let proc = process::Process::new();
    println!("{proc:?}");
}

// process.rs
mod state;

#[derive(Debug)]
pub struct Process {
    // ...
}

// ...
```

Submodules

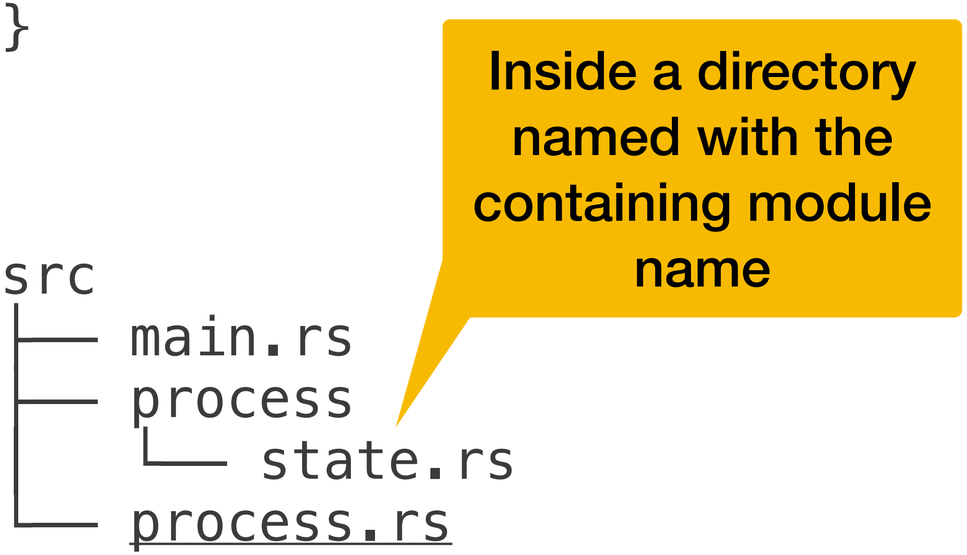
```rust
// main.rs
mod process;

fn main() {
    let proc = process::Process::new();
    println!("{proc:?}");
}

// process.rs
mod state;

#[derive(Debug)]
pub struct Process {
    // ...
}

// ...
```

```rust
// state.rs
pub enum ProcessState {
    Running,
    Runnable,
    // ...
}
```

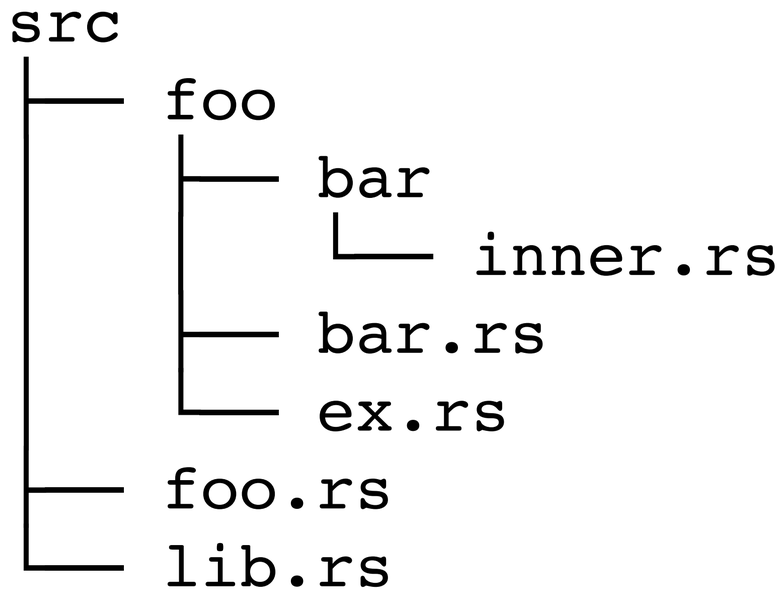
Submodules

```rust
// main.rs
mod process;

fn main() {
    let proc = process::Process::new();
    println!("{proc:?}");
}

// process.rs
mod state;

#[derive(Debug)]
pub struct Process {
    // ...
}

// ...
```

```rust
// state.rs
pub enum ProcessState {
    Running,
    Runnable,
    // ...
}
```



Imagine we have a Rust library crate structured as multiple modules/ submodules. Here's the src directory.



How many "file modules" are there (including submodules)?

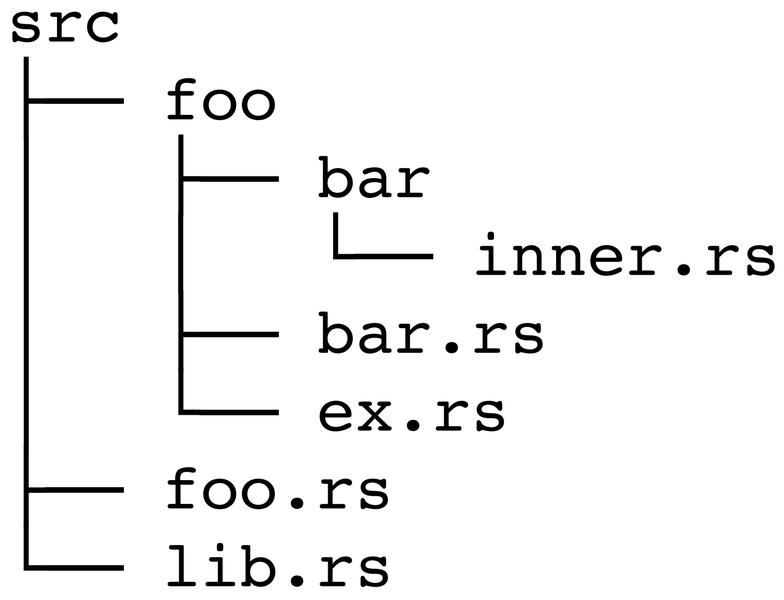
A. 2

B. 3

C. 4

D. 5

Imagine we have a Rust library crate structured as multiple modules/ submodules. Here's the src directory.



Which file contains the line: mod bar;

A. bar.rs

B. ex.rs

C. foo.rs

D. inner.rs

E. lib.rs

Modules form a tree

Imagine each of main.rs, process.rs, and state.rs contained inline test modules for their unit tests

Module tree

```
crate
├── process
│   ├── state
│   │   └── test
│   └── test
└── test
```

File system

```
src
├── main.rs
├── process
│   └── state.rs
└── process.rs
```

Paths to items in modules

We name items (types, functions, etc.) in a module by giving a path to the item

- `crate::process::Process`
- `crate::process::state::ProcessState`

The `crate` keyword refers to the current crate

To name an item in a different crate, we start with the name of the crate

- `std::collections::HashMap`
- `std::io::BufRead`
- `rand::random()`

Absolute vs. relative paths

Inside the same crate, we can refer to items via relative paths

Example: inside the `process` module, we can use `state::ProcessState` to mean the same item as `crate::process::state::ProcessState`

Example: inside the `state` module, we can use `super::Process` to mean the same item as `crate::process::Process`

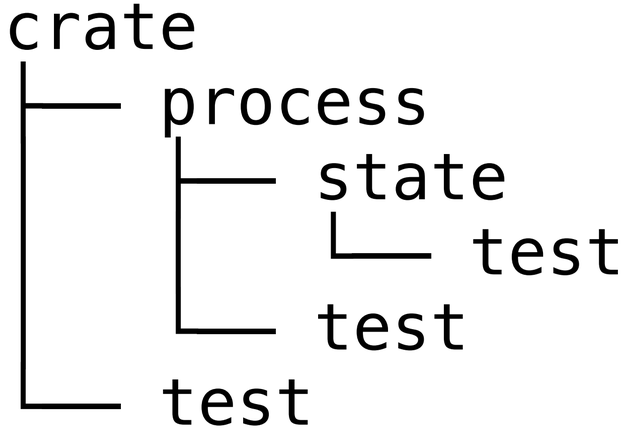
`super` acts like `..` in file system paths: it refers to the parent module

Paths are module paths, not file system paths!

A module’s path need not directly reflect the file system (inline modules)

All paths in Rust are paths within crates and modules, not file system paths

Given the module tree below, how can code in the test submodule of state refer to the ProcessState enum defined in the state module?



A. `ProcessState`

B. `state::ProcessState`

C. `super::ProcessState`

D. `crate::process::state::ProcessState`

E. More than one of the above (which ones?)

Using use

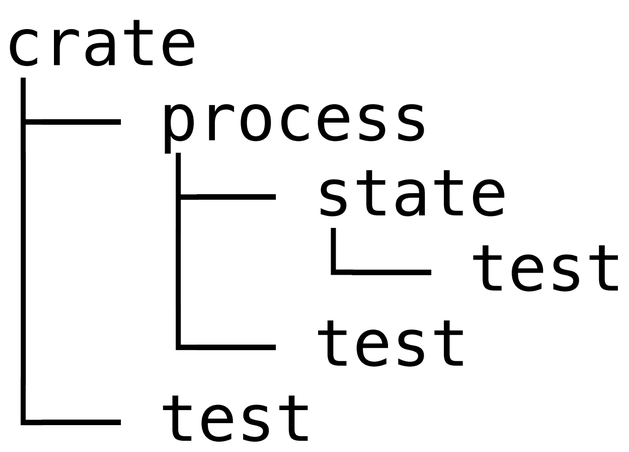
We use the `use` keyword to bring items or modules into scope

```
mod state;

use state::ProcessState;

#[derive(Debug)]
pub struct Process {
    state: ProcessState,
}

impl Process {
    pub fn new() -> Self {
        Self {
            state: ProcessState::Runnable,
        }
    }
}
```

Relative path

Naming the enum without the path

Given the module tree below, how can code in the test submodule of state refer to the ProcessState enum defined in the state module **assuming the test submodule contains the line: use super::*;**



A. `ProcessState`

B. `state::ProcessState`

C. `super::ProcessState`

D. `crate::process::state::ProcessState`

E. More than one of the above (which ones?)

Public vs. private

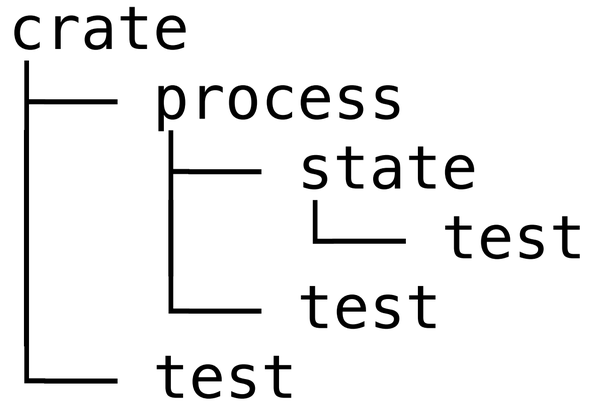
Modules, functions, types, methods, struct fields are all **private by default**

Modules may not access the private items in their descendent modules

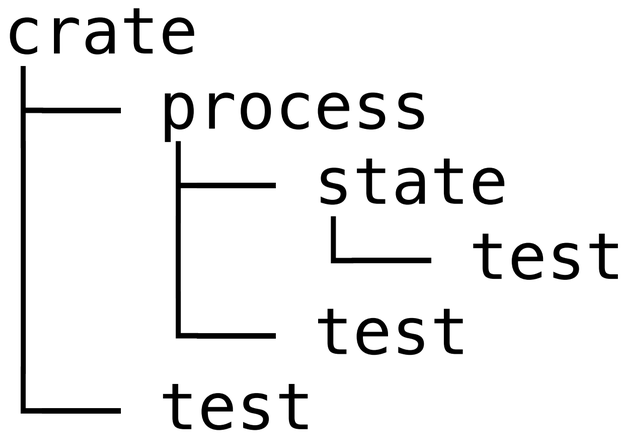
Modules may access private items in their ancestor modules

Examples

- Every module **may** access private items in the crate root
- process **may not** access private items in state
- state **may** access private items in process
- all test modules **may** access private items in their parent module



Which of the following statements are true?



A. `state` may access private items in `process`

B. `process` may access private items in `state`

C. Every module may access private items in the crate root

D. A and C

E. A, B, and C

Accessing an item by path

To quote the Rust Reference

With the notion of an item being either public or private, Rust allows item accesses in two cases:

1. If an item is public, then it can be accessed externally from some module m if you can access all the item's ancestor modules from m.

2. If an item is private, it may be accessed by the current module and its descendants.

https://doc.rust-lang.org/reference/visibility-and-privacy.html

The pub keyword

To make an item publicly visible outside the module, use pub

```
pub mod foo;
```

Declares a module foo and makes it public

- The contents of public modules are still private by default!

```
pub struct Process {
    state: ProcessState,
}
```

Process is public, but its state field is still private

Public structs/enums

By default a public struct's fields are private

Use `pub` before the field name to make that field public

```
pub struct Foo {
    pub x: i32,
    y: i32,
}
```

x is public, y is private

The variants of public enums are always public

```rust
// process.rs
mod state;

use state::ProcessState;

#[derive(Debug)]
pub struct Process {
    state: ProcessState,
}

impl Process {
    pub fn new() -> Self {
        Self {
            state: ProcessState::Runnable,
        }
    }
    pub fn state(&self) -> ProcessState {
        self.state
    }
}
```

```rust
// state.rs
#[derive(Debug, Clone, Copy)]
pub enum ProcessState {
    Running,
    Runnable,
    // ...
}
```

	process::Process.state	state::ProcessState::Runnable
A	public	public
B	public	private
C	private	public
D	private	private

Accessing items in private modules

```rust
// process.rs
mod state;

use state::ProcessState;

#[derive(Debug)]
pub struct Process {
    state: ProcessState,
}

impl Process {
    pub fn new() -> Self {
        Self {
            state: ProcessState::Runnable,
        }
    }

    pub fn state(&self) -> ProcessState {
        self.state
    }
}
```

```rust
// state.rs
#[derive(Debug, Clone, Copy)]
pub enum ProcessState {
    Running,
    Runnable,
    // ...
}
```

Accessing items in private modules

```rust
// process.rs
mod state;

use state::ProcessState;

#[derive(Debug)]
pub struct Process {
    state: ProcessState,
}

impl Process {
    pub fn new() -> Self {
        Self {
            state: ProcessState::Runnable,
        }
    }

    pub fn state(&self) -> ProcessState {
        self.state
    }
}
```

```rust
// state.rs
#[derive(Debug, Clone, Copy)]
pub enum ProcessState {
    Running,
    Runnable,
    // ...
}
```

```rust
// main.rs
mod process;

use process::Process;

fn main() {
    let proc = Process::new();
    match proc.state() {
        process::state::ProcessState::Runnable => {
            println!("Runnable")
        }
        _ => println!("Something else"),
    }
}
```

Accessing items in private modules

```rust
// process.rs
mod state;

use state::ProcessState;

#[derive(Debug)]
pub struct Process {
    state: ProcessState,
}

impl Process {
    pub fn new() -> Self {
        Self {
            state: ProcessState::Runnable,
        }
    }

    pub fn state(&self) -> ProcessState {
        self.state
    }
}
```

```rust
// state.rs
#[derive(Debug, Clone, Copy)]
pub enum ProcessState {
    Running,
    Runnable,
    // ...
}
```

```rust
// main.rs
mod process;

use process::Process;

fn main() {
    let proc = Process::new();
    match proc.state() {
        process::state::ProcessState::Runnable => {
            println!("Runnable")
        }
        _ => println!("Something else"),
    }
}
```

Error!

The error: state module is private

```
error[E0603]: module `state` is private
 --> src/main.rs:9:18
  |
9 |             process::state::ProcessState::Runnable => {
  |                    ^^^^^ private module -------- unit
variant `Runnable` is not publicly re-exported
  |
note: the module `state` is defined here
 --> src/process.rs:2:1
  |
2 | mod state;
  | ^^^^^^^^^^
  |
```

Reexporting items from private modules

Brings ProcessState into scope which gives a new path by which we can refer to it

```
// process.rs
mod state;

use state::ProcessState;

#[derive(Debug)]
pub struct Process {
    state: ProcessState,
}

impl Process {
    pub fn new() -> Self {
        Self {
            state: ProcessState::Runnable,
        }
    }

    pub fn state(&self) -> ProcessState {
        self.state
    }
}
```

```
// state.rs
#[derive(Debug, Clone, Copy)]
pub enum ProcessState {
    Running,
    Runnable,
    // ...
}
```

```
// main.rs
mod process;

use process::Process;

fn main() {
    let proc = Process::new();
    match proc.state() {
        process::ProcessState::Runnable => {
            println!(“Runnable”)
        }
        _ => println!("Something else"),
    }
}
```

Removed state; still an error!

The error: process::ProcessState is private

```
error[E0603]: enum import `ProcessState` is private
 --> src/main.rs:9:18
  |
9 |         process::ProcessState::Runnable => {
  |                  ^^^^^^^^^^^^  -------- unit variant `Runnable` is not publicly re-exported
  |                  |
  |                  private enum import
  |
note: the enum import `ProcessState` is defined here...
 --> src/process.rs:4:5
  |
4 | use state::ProcessState;
  |     ^^^^^^^^^^^^^^^^^^^
note: ...and refers to the enum `ProcessState` which is defined here
 --> src/process/state.rs:3:1
  |
3 | pub enum ProcessState {
  | ^^^^^^^^^^^^^^^^^^^^^ consider importing it directly
```

Wrong advice???

Re-export ProcessState from process

Re-exporting state::ProcessState

```rust
// process.rs
mod state;

pub use state::ProcessState;

#[derive(Debug)]
pub struct Process {
    state: ProcessState,
}

impl Process {
    pub fn new() -> Self {
        Self {
            state: ProcessState::Runnable,
        }
    }

    pub fn state(&self) -> ProcessState {
        self.state
    }
}
```

```rust
// state.rs
#[derive(Debug, Clone, Copy)]
pub enum ProcessState {
    Running,
    Runnable,
    // ...
}
```

```rust
// main.rs
mod process;

use process::Proce

fn main() {
    let proc = Process :new();
    match proc.state() {
        process::ProcessState::Runnable => {
            println!(“Runnable”)
        }
        _ => println!("Something else"),
    }
}
```

Can now use process::ProcessState

Advice

Leave things private by default until rustc complains

If you get an error accessing a field of a struct

- Should code outside the module be able to directly manipulate the fields? If yes, make the fields public. If no, add accessor methods

If you get an error that a module is private while accessing an item in it

- Should external code know about/be able to access the module? If yes, make the module public, if no, re-export the item from the child module

If you get an error that an item in a public module is private

- Should external code be able to access it? If yes, make it public