Crates & Testing

Rust, In Theory and in Practice

CAS CS 392 M1

Overview

We'll look at Rust's **module system,** which is used to organize projects and encapsulate implementation details

We'll talk briefly about **testing**

(We'll talk about what's next in the course)

Organizing Crates

Crates

What is a crate? What is a package?

There are two kinds of crates: **binary** and **library**

Binary crate must have **main** function

A **package** is a bundle of crates, zero or one library create and zero or more binary crates (the file is for the package)

Crate Conventions

Where do you put binary/library crates?

Cargo assumes src/main.rs is a binary crate and src/lib.rs is the library crate

each file in src/bin corresponds to the root of a binary crate

Modules

What is a module?

A **module** is a collection of functions/types in a user-defined namespace

A library crate is composed of modules which encapsulate parts of the code base

(we haven't used modules all that much so far)

Namespaces

What is the point of a module?

A module creates a namespace for a collection of shared functionalities

Modules have mechanisms for making only parts of the functionality public

This is generally good for code hygiene

The Compiler's View

How does the compiler look through modules?

1. Start at the root of a crate

2. find code for declared modules (**mod mod_name**)
   1. inline
   2. src/mod_name.rs
   3. src/mod_name/mod.rs (less idiomatic)

3. Recurse, do the same for submodules

Paths

How do I refer to a function in a module?

By it's path:

module1::submodule2::fun_name

Public vs. Private

How do I hide implementation details from the user of my crate?

Modules and code within modules is considered private by default

pub mod mod_name makes (sub)module public

pub fn fun_name(...) ... makes a function public

We can even use **pub** on particular parts of a structure/enumeration

Making something public essentially tells you which paths are valid

The "use" Keyword

Paths can get verbose, we can use the **use** keyword to bring functions/types into scope so that we don't need to use the whole path

(We've been doing this)

A couple tricks:

- **use mod1::submod2::{fun1, fun2}** is used to bring multiple functions into scope
- **use mod1::submod2::fun1 as fun2** is used to bring into scope with different name

Separating into Different Files

How do I organize my code across multiple files?

It's just about knowing where to put things

» **mod_name** goes in **src/mod_name.rs**

» **submod_name** goes in **src/mod_name/submod_name.rs**

(My personal approach: start with one file, break into multiple files as necessary)

demo

Testing

Overview

Rust has a pretty impressive easy-to-use built-in testing framework

1. Create a module and add the attribute **#[cfg(test)]**

2. Add functions to the module with the **#[test]** attribute, using the various assert macros

Assert Macros

assert!(e) checks if **e** is **true**

assert_eq!(e1, e2) checks **e1** and **e2** are equal (type must implement **PartialEq**)

assert_ne!(e1, e2) you can guess...

Organizing Tests

Where should I put tests?

1. Anywhere! You can put a testing module anywhere you can put a module (with the **#[cfg(test)]** attribute)

2. In the **tests** directory (only has access to public functions)

Testing Cheatsheet

`cargo test`	cargo command to run tests
`cargo test partial_name`	run a subset of tests
`cargo test -- --ignored`	run only ignored tests
`#[cfg(test)]`	attribute to label a module for testing
`#[test]`	attribute to label a function for testing
`#[ignore]`	attribute to ignore a test
`#[should_panic]`	attribute to say that a test will panic
`assert!(e)`	macro that checks if e is true
`assert_eq!(e1, e2)`	macro that checks e1 and e2 are equal (type must implement PartialEq)
`assert_ne!(e1, e2)`	macro that checks if e1 and e2 are not equal (type must implement PartialEq)

demo