
RUST PROGRAMMING
Modules and Crates
Kamal kumar mukiri, Apt Computing Labs

MODULES AND CRATES

Modules

- Introduction
- Defining and accessing
- Visibility and Privacy
- Organizing

Crates

- Importing Third Party Crates
- Examples

Introduction

Crates: In Rust, a "crate" is the fundamental unit of compilation, packaging, and distribution.

- Self-contained
- Focused-Responsibility
- Encapsulation
- Reusable
- Development Management
- Cohesion

A sample example:

1. How to create a crate (library) ?
2. How to include library as dependency in my application ?
3. How to invoke the APIs from Library ?

Types of Modules

- **Inline Modules**: Define the module content directly in the same file.
- **File-based Modules**: Define the module in a separate file to keep the code more organized.

In-line Modules: Example


```rust
// Defining module
mod math {
    pub fn add(a:i32, b:i32) -> i32 {
        return a+b;
    }
    // Defining sub-module
    pub mod trignometric {
        pub fn trig_fun1() -> Option<String> {
            return Some(String::from("returning from trig_fun1"));
        }
    }
}

fn main() {
    // Calling function from module
    println!("Result = {}", math::add(1,2));

    //Calling function from sub module of "math"
    println!("trig_fun1 output = {:?}", math::trignometric::trig_fun1());
}
```

```
> ./inline_module
Result = 3
trig_fun1 output = Some("returning from trig_fun1")
```

1. Module should be defined with start “mod” and module name.
2. Invoking API ”add” from module “math”.
3. Defining sub-module
4. Invoking API from sub-module

Keyword “mod”:

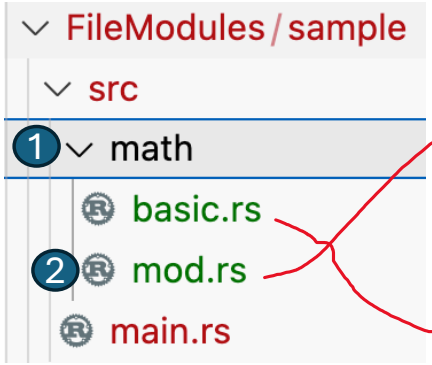
What is mod Keyword ?

- The mod keyword is used to **declare a module**.
- It defines the module structure in your codebase, making it possible to include files or internal modules in your project.
- The mod declaration tells the compiler where to look for the module's code.

When to use mod?

- You use mod when you want to **define** a module inside your project.
- You can create a new module inside the same file or in a separate file.

File based Modules: Example




```rust
pub mod basic;
```

```rust
// math.rs

pub fn add(a: i32, b: i32) -> i32 {
    a + b
}

pub fn subtract(a: i32, b: i32) -> i32 {
    a - b
}
```

```rust
mod math;
fn main() {
    println!("Hello, world!");
    //calling funciton "add" from module "math"
    println!("Result = {}", math::basic::add(1,2));
}
```

1. ”math” is not a simple module, it has many functionalities
2. The mod.rs file serves as the entry point for the “math” module, allowing you to group multiple modules inside “math”.

Prelude:

- In Rust, the **prelude** is a set of commonly used items (functions, traits, macros, etc.) that are automatically imported into every Rust program without needing to explicitly specify them. It's designed to make common tasks easier by reducing the need to import frequently used types and functions manually.

Prelude:

Steps for Creating a Custom Prelude:

1. **Create a prelude module**: Define a separate module in your crate (often named prelude) where you re-export the items (traits, structs, functions) you want to make easily available to the users.
2. **Re-export items in the prelude module**: In the prelude module, re-export the most commonly used items from your crate.
3. **Users can use the prelude**: Users of your crate can then simply import your prelude, making all of its contents accessible without needing to explicitly list them out.

Prelude: Example

1
```rust
// src/lib.rs
pub mod my_module;
pub mod prelude;  // Define a `prelude` module
```

2
```rust
// src/my_module.rs
pub struct MyStruct;

pub fn my_function() {
    println!("Hello from my_function!");
}

pub trait MyTrait {
    fn do_something(&self);
}

impl MyTrait for MyStruct {
    fn do_something(&self) {
        println!("MyStruct does something!");
    }
}
```

3
```rust
// src/prelude.rs
// Re-export commonly used items here
pub use crate::my_module::{MyStruct, MyTrait, my_function};
```

4
```rust
// Main file or another crate
use my_crate::prelude::*;

fn main() {
    let instance = MyStruct;
    instance.do_something();  // Using `MyTrait`

    my_function();  // Using the re-exported function
}
```

Creating lib and application in same project:

- Create a New Cargo Project

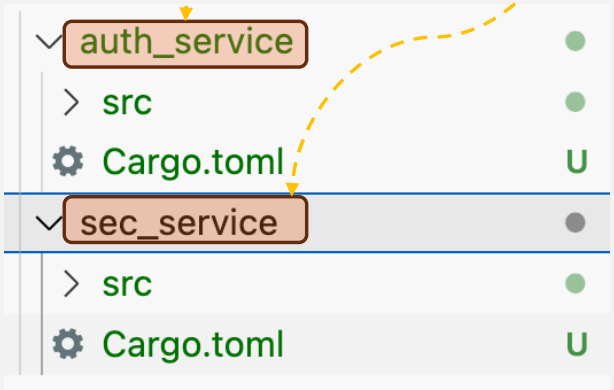
```
cargo new my_project --lib
```

- Add the Binary Entry Point (main.rs)
- Write the Code for the Library (lib.rs)
- Write the Code for the Binary (main.rs)

Lib, app as separate projects:

Step 1: Create the **Library** and **Application**

```
➢cargo new --lib auth_service
➢cargo new --bin sec_service
```



Cont...

Step 2: Include the Library in Application

- Add details about library and its path in **Cargo.toml (Application environment)**

```
1   [package]
2   name = "sec_service"
3   version = "0.1.0"
4   edition = "2021"
5
6   [dependencies]
7   auth_service = {path= "../auth_service"}
8
```

Cont...

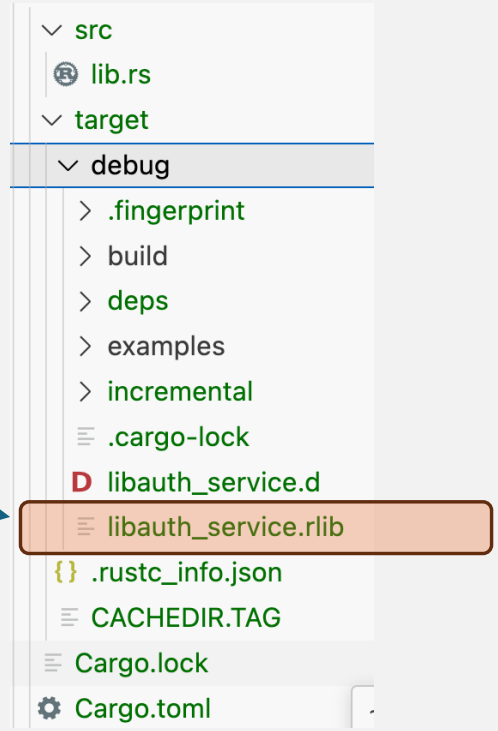
Step 3: Add functionality to library, compile and find .rlib

```rust
#[derive(Debug)]
0 implementations
pub enum AuthError {
    AuthSuccess,
    AuthError
}

pub fn verify_authentication(user:&str, pass:&str) -> Result<AuthError, String> {
    println!("Going to to authenticate user: {} and pass: {}", user, pass);
    return Ok(AuthError::AuthSuccess);
}

```

```
➤ cargo build
```



Cont...

Step 4: Include library and invoke API

```
1   use auth_service;
2   fn main() {
3       println!("Hello, world!");
4       let result = auth_service::verify_authentication("Kamal", "@123");
5       println!("Result = {:?}", result);
6   }
```

➢ cargo build

```
> cargo build --verbose
   Compiling auth_service v0.1.0 (/Users/kamalmukiri/Documents/GitHub/Courses/APTRUSTSESS01/Classes/Kamal/Sep30_Modules_IntegerOverflow/Modules/auth_service)
     Running `rustc --crate-name auth_service --edition=2021 /Users/kamalmukiri/Documents/GitHub/Courses/APTRUSTSESS01/Classes/Kamal/Sep30_Modules_IntegerOverflow/Modules/auth_service/src/lib.rs --error-format=json --json=diagnostic-rendered-ansi,artifacts,future-incompat --diagnostic-width=248 --crate-type lib --emit=dep-info,metadata,link -C embed-bitcode=no -C debuginfo=2 -C split-debuginfo=unpacked -C metadata=e04437df63bd6c9b -C extra-filename=-e04437df63bd6c9b --out-dir /Users/kamalmukiri/Documents/GitHub/Courses/APTRUSTSESS01/Classes/Kamal/Sep30_Modules_IntegerOverflow/Modules/sec_service/target/debug/deps -C incremental=/Users/kamalmukiri/Documents/GitHub/Courses/APTRUSTSESS01/Classes/Kamal/Sep30_Modules_IntegerOverflow/Modules/sec_service/target/debug/incremental -L dependency=/Users/kamalmukiri/Documents/GitHub/Courses/APTRUSTSESS01/Classes/Kamal/Sep30_Modules_IntegerOverflow/Modules/sec_service/target/debug/deps`
   Compiling sec_service v0.1.0 (/Users/kamalmukiri/Documents/GitHub/Courses/APTRUSTSESS01/Classes/Kamal/Sep30_Modules_IntegerOverflow/Modules/sec_service)
     Running `rustc --crate-name sec_service --edition=2021 src/main.rs --error-format=json --json=diagnostic-rendered-ansi,artifacts,future-incompat --diagnostic-width=248 --crate-type bin --emit=dep-info,link -C embed-bitcode=no -C debuginfo=2 -C split-debuginfo=unpacked -C metadata=2ad8da3b13476cdb -C extra-filename=-2ad8da3b13476cdb --out-dir /Users/kamalmukiri/Documents/GitHub/Courses/APTRUSTSESS01/Classes/Kamal/Sep30_Modules_IntegerOverflow/Modules/sec_service/target/debug/deps -C incremental=/Users/kamalmukiri/Documents/GitHub/Courses/APTRUSTSESS01/Classes/Kamal/Sep30_Modules_IntegerOverflow/Modules/sec_service/target/debug/incremental -L dependency=/Users/kamalmukiri/Documents/GitHub/Courses/APTRUSTSESS01/Classes/Kamal/Sep30_Modules_IntegerOverflow/Modules/sec_service/target/debug/deps --extern auth_service=/Users/kamalmukiri/Documents/GitHub/Courses/APTRUSTSESS01/Classes/Kamal/Sep30_Modules_IntegerOverflow/Modules/sec_service/target/debug/deps/libauth_service-e04437df63bd6c9b.rlib`
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.95s
> 
```

Binary and Libraries are to be part of same project:


```
$ cargo new veh-stack --bin
$ cd veh-stack
$ cargo new sensors --lib
$ cargo new control --lib

// Cargo.toml (root workspace)
[workspace]
members = ["sensors", "control", "veh-stack"]
```

Cont...

Step 5: Run the application

```
> cargo run
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.06s
     Running `target/debug/sec_service`
Hello, world!
Going to to authenticate user: Kamal and pass: @123
Result = Ok(AuthSuccess)
>
```

Introduction:

In Rust, crates are used to share reusable code across projects. Here are some examples of using third-party crates in Rust.

”use” keyword:

What is the use of keyword “use” ?

- The use keyword is used to **bring a module, function, type, or constant into scope**.
- It makes it easier to reference items defined in other modules without needing to use the fully qualified path each time.

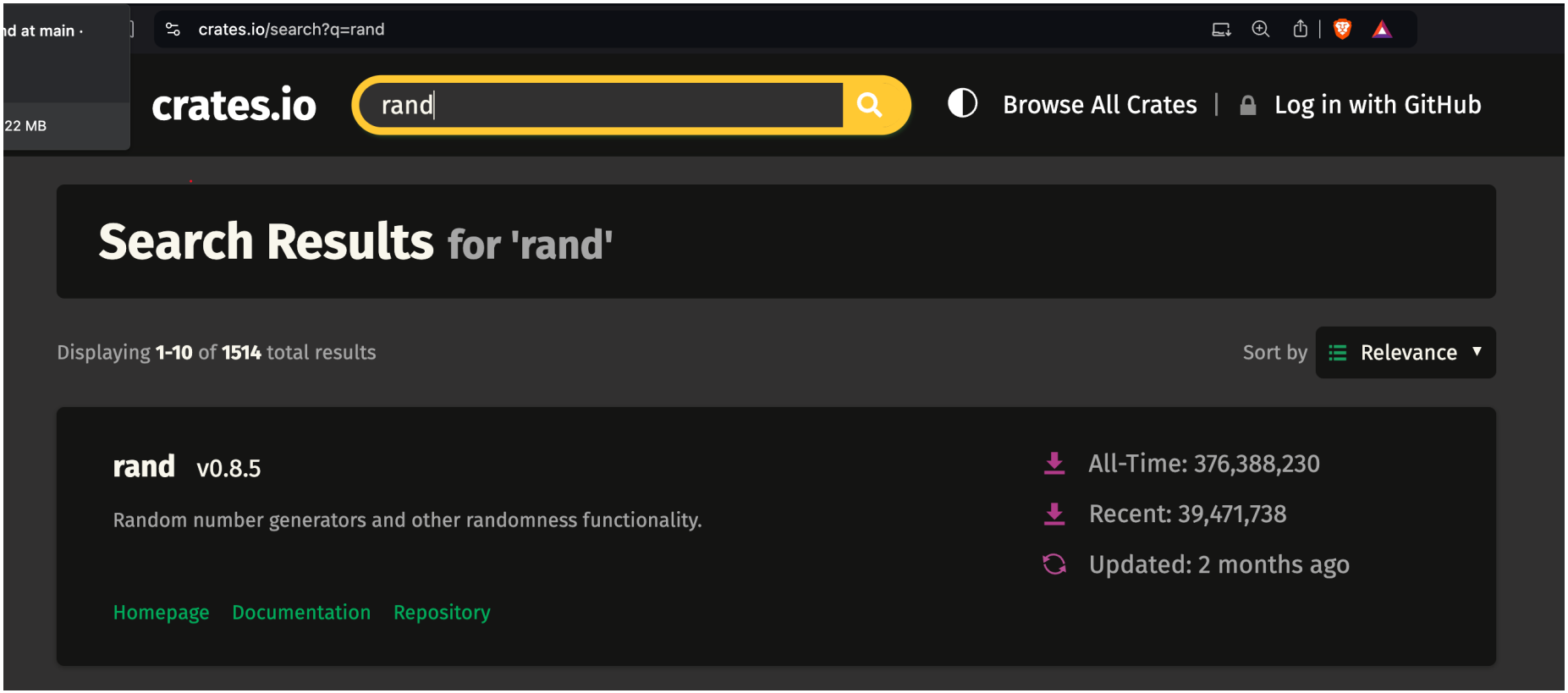
When to use “use”?

- Use use when you want to **access** functions, structs, or other items from a module.
- This allows you to use shorter paths to refer to those items in your code.

Steps to use third party crates:

1) Find crate in crates.io
2) Include crate name and version in Cargo.toml
3) Import crate in source code
4) Call API/Interface/etc...

Finding crate in website:



Include create name in project file: Method 1

Oct2_Crates > rand > rand_example > Cargo.toml

```
1   [package]
2   name = "rand_example"
3   version = "0.1.0"
4   edition = "2021"
5
6   [dependencies]
7   rand = "*"
```

“*” means a recent version

Include create name in project file: Method 2


```
> cargo add rand
```

Note: The above command adds the crate name in Cargo.toml as dependencies along with recent version.

Import crate in source code and invoke API:


```rust
use rand::Rng;

fn main() {
    let mut rng = rand::thread_rng();
    let random_number: u32 = rng.gen_range(1..=100);
    println!("Random number: {}", random_number);
}
```

Thank you