# Cargo and Crates



## Organizing Codes: Module System

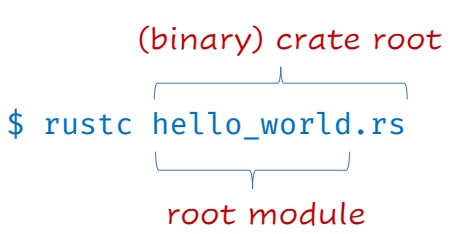
- **module**
  - A unit of composition for crates
  - Consists of coherent data types and/or functions
  - Encapsulation and Scope
- **crate**
  - *Unit of compilation*
  - Consists of a tree of one *root module* + zero or more *sub-modules*
  - Either ***binary*** *crate* or ***library*** *crate*
- **package**
  - Unit of deployment
  - Contains a **Cargo.toml** file that describes how to build contained crates.
  - [binary crate* && library crate?]
- **workspace**
  - A collection of related packages

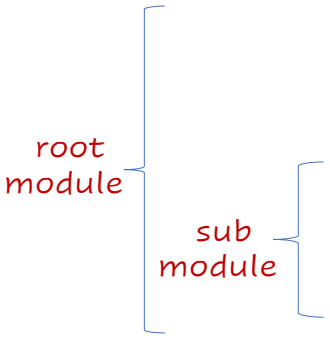
A **crate root** is a file containing the root module

Usually called a "library"

# Crates

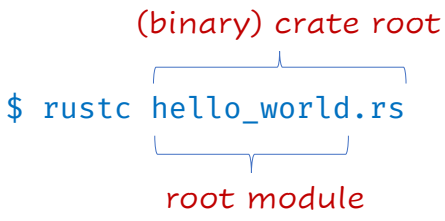
- A crate contains *modules*, some of which may be defined in other files.
- The crate root makes up the ***root module*** of a crate, and corresponds to a source file that the Rust compiler starts from.
- A crate is a ***compilation unit*** in Rust.
  - Modules do *not* get compiled individually, only crates get compiled.

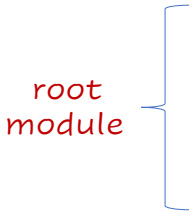


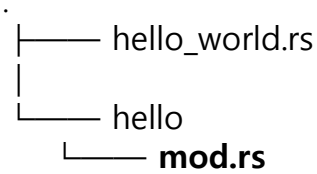

```
// hello_world.rs file
fn main() {
    hello::saly();
}

mod hello {
    pub fn saly() {
        println!("Hello, World");
    }
}
```

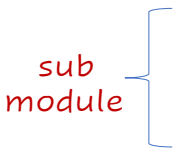
# Crates (Cont'd) – Old style




```
// hello_world.rs file
mod hello;

fn main() {
    hello::saly();
}
```




```
// mod.rs file
pub fn saly() {
  println!("Hello, World!");
}
```

# Crates (Cont'd)

(binary) crate root

```
$ rustc hello_world.rs
```

root module

root module

```
// hello_world.rs file
mod hello;

fn main() {
    hello::saly();
}
```

```
.
├── hello_world.rs
│
└── hello.rs
```

sub module

```
// hello.rs file
pub fn saly() {
  println!("Hello, World!");
}
```



# Binary Crates and Library Crates

A crate can come in one of two forms:

- **Binary crate**
  - programs you can compile to an executable
  - each must have a function called `main()`
- **Library crate**
  - define functionality intended to be shared with multiple projects
  - don't have a `main` function

By "crate", we normally mean library crate, and use "crate" interchangeably with a "library".

# Packages and Crates

- A package is a bundle of one or more crates that provides a set of functionality.
- A package contains a `Cargo.toml` file that describes how to build those crates.
- A package can contain [binaries* && library?] crates.

```
// Sample Cargo.toml file
[package]
name = "tutorial"
version = "0.1.0"
edition = "2021"

[dependencies]
rand = "0.8.5"
futures = { version = "0.3.*" }
tokio = { version = "1.25.*", features = ["full"] }
```



# Cargo

- **cargo** is the official Rust *package management and build tool*.
  - Dependency management and integration with crates.io (the official Rust package registry)
  - Awareness of unit tests
  - Awareness of benchmarks

- Refer to Cargo Book (https://doc.rust-lang.org/cargo/)

# To create, build, and run a new project:

```
# A binary package
cargo new foo --bin

# To build at anywhere!
cargo build

# To execute
cargo run


# A library package
cargo new bar --lib
```

```
.
└── foo
    ├── Cargo.toml   <-- config file
    └── src
        └── main.rs  <-- root crate file
```

```toml
[package]
name = "foo"
version = "0.1.0"
edition = "2021"

[dependencies]
rand = "0.8.5"
futures = { version = "0.3.*" }
tokio = { version = "1.25.*", features = ["full"] }
```

```
.
└── bar
    ├── Cargo.toml
    └── src
        └── lib.rs
```



# Multiple Binary Crates

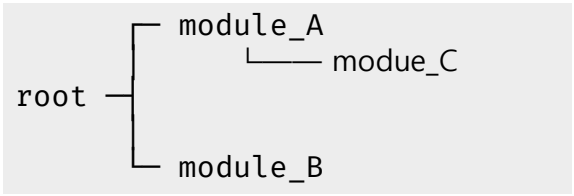
- The default binary name is `main`, but you can add additional binaries by `placing` them in a `bin/` directory:

```
labs
├── Cargo.toml
└── src
    ├── main.rs
    └── bin
        └── multi_file_executable
        |       ├── main.rs
        |       └── sub_module.rs
        └── lifetime.rs
        └── smart_pointer.rs
```

```
# To execute lifetime
cargo run --bin lifetime
```

```toml
[package]
name = "labs"
version = "0.1.0"
edition = "2021"
default-run = "labs"

[[bin]]
name = "lifetime"
path = "src/bin/lifetime.rs"

[[bin]]
name = "smart_pointer"
path = "src/bin/smart_pointer.rs"

[[bin]]
name = "multi_file_executable"
path = "src/bin/multi_file_executable/main.rs"
```

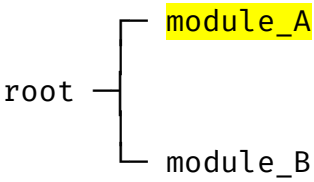
# Modules in a Single File

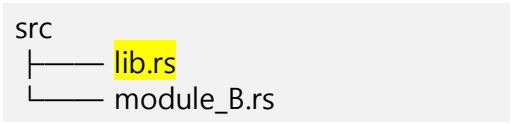



```rust
pub use crate::module_A::module_C;

mod module_A {

  pub mod module_C {

  }
}

// private module
mod module_B {

}
```



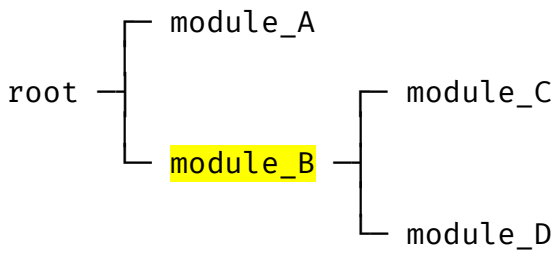
# Modules in Separate Files (1)




```rust
// lib.rs crate root
pub mod module_A {
    pub fn func_a() {
        println!("Hello from module_A::func_a()");
    }
}

pub mod module_B; // Load module_B from external file
```

# Modules in Separate Files (2)

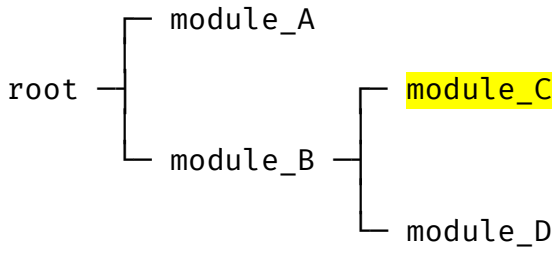



```
src
 ├── lib.rs
 ├── module_B.rs
 ├── module_C.rs
 └── module_D.rs
```

```rust
// file module_B.rs
// Load module_C and D from external files
pub mod module_C;
pub mod module_D;

pub fn func_b() {
    println!("Hello from module_B::func_b()");
}
```



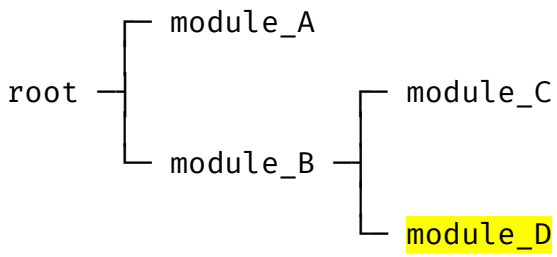
# Modules in Separate Files (3)




```
src
 ├── lib.rs
 ├── module_B.rs
 ├── module_C.rs
 └── module_D.rs
```

```rust
// file module_C.rs
pub fn func_c() {
    println!("Hello from module_C::func_c()");
}
```

# Modules in Separate Files (4)

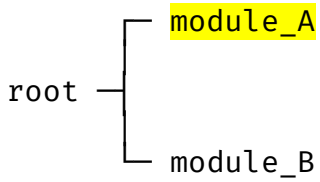



```
src
 ├── lib.rs
 ├── module_B.rs
 ├── module_C.rs
 └── module_D.rs
```

```rust
// file module_D.rs
pub fn func_d() {
    println!("Hello from module_D::func_d()");
}
```



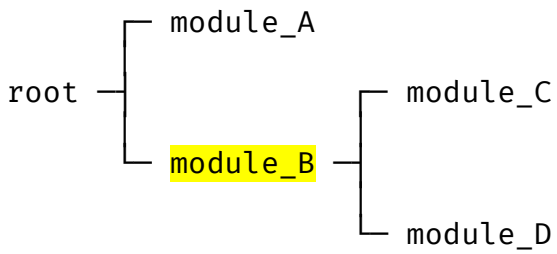
# Modules in Separate Files (1) – Old Style




```
src
 ├── lib.rs
 └── modue_B
       └── mod.rs
```

```rust
// lib.rs crate root
pub mod module_A {
    pub fn func_a() {
        println!("Hello from module_A::func_a()");
    }
}

// Load module_B from external file
pub mod module_B;
```

# Modules in Separate Files (2) – Old Style

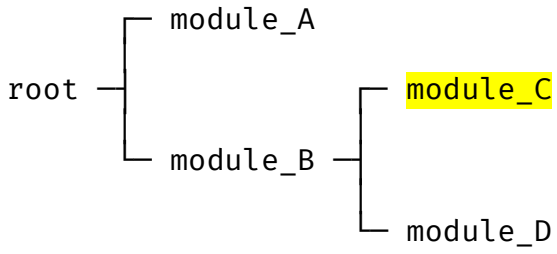



```
src
 ├── lib.rs
 └── modue_B
        └── mod.rs
```

```rust
// file module_B.rs
// Load module_C and D from external files
pub mod module_C;
pub mod module_D;

pub fn func_b() {
    println!("Hello from module_B::func_b()");
}
```



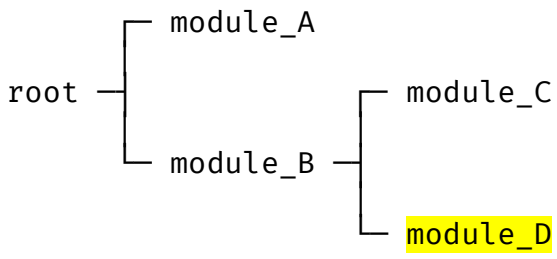
# Modules in Separate Files (3) – Old Style




```
src
 ├── lib.rs
 └── modue_B
        ├── mod.rs
        └── module_C
               └── mod.rs
```

```rust
// file module_C.rs
pub fn func_c() {
    println!("Hello from module_C::func_c()");
}
```

# Modules in Separate Files (4) – Old Style

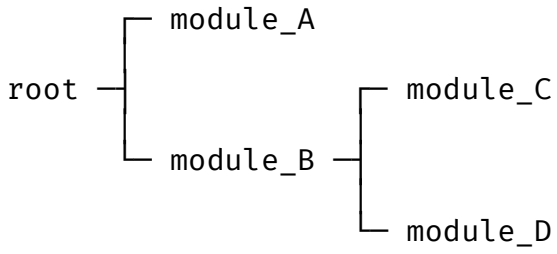



```
src
 ├── lib.rs
 └── modue_B
      ├── mod.rs
      ├── module_C
      │    └── mod.rs
      └── module_D
           └── mod.rs
```

```rust
// file module_D.rs
pub fn func_d() {
    println!("Hello from module_D::func_d()");
}
```



# Modules in Separate Files – Mixed Style




```
src
 ├── lib.rs
 └── modue_B
      ├── mod.rs
      ├── module_C.rs
      └── module_D
           └── mod.rs
```

```rust
// lib.rs crate root
pub mod module_A {
    pub fn func_a() {
        println!("Hello …");
    }
}

pub mod module_B;
```

```rust
// file module_B/mod.rs
pub mod module_C;
pub mod module_D;

pub fn func_b() {
    println!("Hello …");
}
```

```rust
// file module_C.rs
pub fn func_c() {
    println!("Hello …");
}
```

```rust
// file module_D/mod.rs
pub fn func_d() {
    println!("Hello …");
}
```

# Import with `use`

A path can take two forms:

- An *absolute path* starts from a crate root by using a crate name or a literal `crate`.
- A *relative path* starts from the current module and uses `self`, `super`, or an identifier in the current module.

```
restaurant
 ├─── back_of_house
 └─── front_of_house
     └─── hosting
```

```rust
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
    }
}

// `self` refer to current module
use self::front_of_house::hosting;
// use front_of_house::hosting;
// use crate::front_of_house::hosting;
// use restaurant::front_of_house::hosting;
hosting::add_to_waitlist();

fn serve_order() {}

mod back_of_house {
    fn order() {
        // `super` to refer parent module
        super::serve_order();
    }
}
// Brings std::io, std::io::Write into scope
use std::io::{self, Write};
```



# Re-Export with `pub use`

```rust
pub mod restaurant {
    // both current module and external module can refer to `hosting`
    pub use self::front_of_house::hosting;
    …
}

// Some external module that imports `restaurant` module
use restaurant;
hosting::add_to_waitlist();
```

# Using library crate from binary crate

```rust
// Cannot use `crate` here because it refers to binary crate root
use labs::{ module_A, module_B };
use module_B::{ module_C, module_D };

fn main() {
    println!("Hello, world!");

    module_A::func_a();
    module_B::func_b();
    module_C::func_c();
    module_D::func_d();
}
```

```rust
// lib.rs
mod module_A {
   fn func_a() { … }
}

mod module_B {
  fn func_b() { … }

  mod module_C {
    fn func_c() { … }
  }

  mod module_D {
    fn func_d() { … }
  }
}
```



# Testing

Three types of testings

- **Unit test**
- **Integration test**
- **Doc test**

```
# To execute all tests
cargo test
```

```
# To execute filtered unit tests
cargo test pattern
```

```
❯ cargo test
   Compiling testing v0.1.0 (D:\workspace\rust\basics\testing)
    Finished test [unoptimized + debuginfo] target(s) in 1.58s
     Running unittests src\lib.rs (D:\workspace\rust\basics\target\debug\de

running 6 tests
test tests::expensive_test ... ignored
test tests::add_two_using_add_two ... ok
test tests::add_two_with_two_using_add ... ok
test tests::one_hundred ... ok
test tests::should_print_to_screen ... ok
test tests::should_panic - should panic ... ok

test result: ok. 5 passed; 0 failed; 1 ignored; 0 measured; 0 filtered out;

     Running tests\another_test.rs (D:\workspace\rust\basics\target\debug\d

running 1 test
test it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;

     Running tests\integration_test1.rs (D:\workspace\rust\basics\target\de

running 1 test
test it_adds_two ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;

   Doc-tests testing

running 2 tests
test testing\src\lib.rs - add_two (line 14) ... ok
test testing\src\lib.rs - (line 3) ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
```

# Unit Tests

- Define tests inside target modules directly.
- The convention is to create a module named `tests` in each file to contain the test functions and to annotate the module with `cfg(test)`.

```
#[cfg(test)]
mod tests {
    #[test]
    fn it_works() {
        let result = 2 + 2;
        assert_eq!(result, 4);
    }
}
```

```
# To execute unit tests
cargo test --lib
cargo test –bin target
cargo test --bins
```



# Integration Tests

- In Rust, integration tests are entirely external to your library.
  - They use your library in the same way any other code would, which means they can only call functions that are part of your library's public API.
- To create integration tests, you first need a `tests` directory.

```
adder
├── Cargo.lock
├── Cargo.toml
├── src
│   └── lib.rs
└── tests
    └── integration_test.rs
```

```
use adder;

#[test]
fn it_adds_two() {
    assert_eq!(4, adder::add_two(2));
}
```

```
# To execute integration tests
cargo test --test integration_test
```

# Doc Tests

- To guarantee documents are in sync with codes.

```rust
//!
//! # Examples
//! ```
//! let result = testing::add(2, 2);
//! assert_eq!(result, 4);
//! ```

pub fn add(left: usize, right: usize) -> usize { left + right }

///
/// # Examples
/// ```
/// let result = testing::add_two(2);
/// assert_eq!(result, 4);
/// ```

pub fn add_two(a: i32) -> i32 { a + 2 }
```

```
# To execute doc tests
cargo test --doc
```



# Attributes

An attribute is *metadata* applied to some *module*, *crate* or *item*.

- conditional compilation of code
- set crate name, version and type (binary or library)
- disable lints (warnings)
- enable compiler features (macros, glob imports, etc.)
- link to a foreign library
- mark functions as unit tests
- mark functions that will be part of a benchmark
- attribute like macros

# Attributes (cont'd)

- When apply to a whole crate:

```
#![crate_attribute]
```

- When apply to a module or item:

```
#[item_attribute] // notice the missing bang !
```

- Attributes can take arguments with different syntaxes:

```
#[attribute = "value"]
#[attribute(key = "value")]
#[attribute(value)]
```

```
#![allow(dead_code)]
#![allow(unused)]
```



# cfg: Configuration conditional checks

```
// the cfg attribute in attribute position
#[cfg(...)]

// the cfg! macro in boolean expressions
cfg!(...)
```

```
#[cfg(target_os = "linux")]
fn are_you_on_linux() { ... }

#[cfg(not(target_os = "linux"))]
fn are_you_on_linux() { ... }

fn main() {
    are_you_on_linux();

    println!("Are you sure?");
    if cfg!(target_os = "linux") {
        println!("Yes. It's definitely linux!");
    } else {
        println!("Yes. It's definitely *not* linux!");
    }
}
```

# Custom Configuration

```
#[cfg(feature = “some_condition”)]
fn conditional_function() {
    println!("condition met!");
}

fn main() {
    #[cfg(feature = “some_condition”)]
    conditional_function();

    … // other codes
}
```

```
// In Cago.toml
[features]
some_condition = []
```

$ cargo run --features “some_condition”