CoE 164

Computing Platforms

04b: Cargo Packages

GROWING CODEBASE

As a software project increases, so is its codebase. Code organization is important to facilitate faster work and increase separation of concerns.

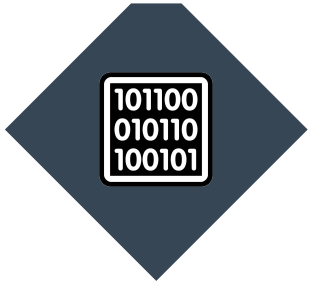
Rust provides a *module system* so that certain codes can be visible to the "public" (i.e. other developers).

CRATES

A **crate** is the basic "unit" of code in Rust. Compiling code using `rustc` is considered to be compiling a single crate.

Crates always have a *crate root*, which is where Rust starts looking for files to compile.

CRATES: TYPES



Binary

- Code with a `main` entry point
- Can be run by itself
- Intended to be used as a standalone program
- Starts usually at `main.rs`

Library

- Collection of codes
- Cannot be run by itself
- Intended to be used as a reference program for other programs
- Starts usually at `lib.rs`

PACKAGES

A **package** is a collection of related crates that provide some sort of functionality.

A package always has a metadata file named `Cargo.toml` describing how to compile and build the crates in it.

CARGO

The Cargo package is the build system of Rust. It enables programmers to manage their Rust projects and automates usual tasks.

A new package can be generated using `cargo new <package_name>`.

```
PS ~ cargo new complex_nums
     Created binary (application) `complex_nums` package
PS ~ cd complex_nums
```

Example

CARGO: FOLDER STRUCTURE

Cargo will generate a `Cargo.toml` file. It will also generate a `src/` folder where the source code should be stored. A "Hello, world!" `main.rs` file is also inside of it for editing.

You can freely add folders and files in the `src/` folder.

Example

```
complex_nums
    | src
    |   | main.rs
    | .gitignore
    | Cargo.toml
```

* .gitignore is a file used to exclude files from being included in some version control system named Git.

CARGO: MANIFEST FILE

The `Cargo.toml` file, or the **manifest** contains metadata to inform Cargo how to compile the package. It is written in TOML (Tom's Obvious Minimal Language), a file format similar to Windows configuration files (.ini).

Example

```
[package]
name = "complex_nums"
version = "0.1.0"
edition = "2021"
```

CRATE ROOTS

Binary crates run themselves starting from the `src/main.rs` file. When compiled, it produces an executable of the same name as the package.

Alternatively, the root of a library crate is the `src/lib.rs` file.

By default, the `src/main.rs` and `src/lib.rs` files are called *crate roots*.

Example

```
complex_nums
    | src
    |   | main.rs
    |   | lib.rs
    | .gitignore
    | Cargo.toml
```

MULTIPLE CRATES

Additional binary crates are added in the `src/bin` folder.

It is possible for a package to be both a binary and a library crate. It can contain any number of binary crates but only one library crate.

Example

```
complex_nums
    | src
    |   | main.rs
    |   | lib.rs
    | bin
    |   | anyhow
    |   |  | ...
    | .gitignore
    | Cargo.toml
```

MODULES

A **module** is some grouping of files within the source tree of a package. Modules enable scoping and privacy of files for cases when we wanted to disseminate the package publicly but block usage of code not intended for use by others.

MODULES

Modules are declared using a `mod` block. This can contain any variables, functions, or declared data types grouped sensibly.

Modules can also contain other modules.

Example

```
mod complex {
    mod dtypes {
        struct ComplexRect {
            real: f64,
            imag: f64
        }
    }

    fn gcd(a: i64, b:i64) -> i64 {
    }
}
```

MODULES: MODULE TREE

In addition to the file tree, Cargo looks into the *module tree* to resolve **paths**, alternative names for functions and data types within a project.

The module tree is created using the information in the crate roots.

Example

```
crate
    | complex
    |     | dtypes
    |     |    | ComplexRect
    |     | gcd
```

MODULES: PATHS

Code inside modules or folders can be invoked using paths. Each "layer" is separated by double colons (::).

Absolute paths are stated relative to the crate root denoted by the `crate` keyword. It is recommended to use absolute paths to refer to different parts of a module.

```
// Absolute
crate::complex::gcd(7, 77);

let z = crate::complex::dtypes::ComplexRect {
    real: 12.00,
    imag: -10.00,
};
```

Example

MODULES: PATHS

Relative paths are stated relative to the current module.

The `super` keyword denotes the parent of the current module.

Example

```
// Relative
let z = dtypes::ComplexRect {
    real: 12.00,
    imag: -10.00,
};

// Super
let z =
super::dtypes::ComplexRect {
    real: 12.00,
    imag: -10.00,
};
```

MODULES: VISIBILITY

By default, modules are *private* and inaccessible from users that would import them. The `pub` keyword should be placed beside the module, function, or struct to selectively make them public from users.

Blocks with parents should have their parents also set to public so that they become visible.

Example

```
pub mod complex {
    pub mod dtypes {
        struct ComplexRect {
            real: f64,
            imag: f64,
        }
    }

    pub fn gcd(a: i64, b:i64) ->
i64 {
        // ...
    }
}
```

MODULES: VISIBILITY

Fields of structs should be separately marked as public even though the struct itself is. In contrast, all variants of an enum become public if the enum itself is.

Example

```
pub struct ComplexRect {
    pub real: f64,
    pub imag: f64,
}

pub enum Option <E> {
    Some(E),
    None,
}
```

MODULES: SHORTCUTS

The `use` keyword is used to create a "shortcut" to a path in the module. The last name in the path declaration is the default name of the shortcut.

Optionally, we can rename the shortcut by writing the `as` keyword followed by the new name.

```
use crate::complex::{gcd};
use crate::complex::dtypes  as dtyp2;
use dtypes::{ComplexRect};
```

Example

MODULES: IDIOMATIC SHORTCUTS

When exposing functions, the `use` shortcut should be referenced up to its parent module. This provides a *namespace* that tags the function under a certain module.

On the other hand, structs, enums, and other items can be referenced up to the said item itself *unless* at least two modules have the same item name.

Example

```
use crate::complex::{gcd};
use crate::complex::dtypes as dtyp2;
use dtypes::{ComplexRect};

gcd(7, 77);

let z = ComplexRect {
    real: 12.00,
    imag: -10.00,
};

dtyp2::ComplexRect {
    real: 0.00,
    imag: 0.30,
};
```

MODULES: NESTED PATHS

Items inside the same path can be separately declared by listing them inside curly braces instead of having a separate `use` declaration.

```
use dtypes::{ComplexRect};

let z = ComplexRect {
    real: 12.00,
    imag: -10.00,
};
```

Example

MODULES: RE-EXPORT

Modules sometimes use their own modules but still want to expose those for use by outside packages.

The `pub use` keyword can be used to both use the module in the code *and* make it available publicly *from* that code.

```
pub use crate::complex::gcd;

// Outside packages can reference complex_nums::gcd()
// instead of complex_nums::complex::gcd()
gcd(7, 5);
```

Example

EXTERNAL PACKAGES

Publicly-available crates can be added to a package by adding the name and version of the crate under the `dependencies` key in its `Cargo.toml` file. When building, these crates are automatically downloaded off the internet.

crates.io is the official crate registry of the Rust community.

```
# For most program dependencies
[dependencies]
rand = "0.8.5"

# For example code and tests
[dev-dependencies]
pretty_assertions = "1"
```

Example

RESOURCES

- The Rust Book

CoE 164

Computing Platforms

04b: Cargo Packages