Rust : Compiling and Running

David Bouchet

david.bouchet.epita@gmail.com

Your First Program

hello.rs

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

```
$ ls
hello.rs
$ rustc hello.rs
$ ls
hello hello.rs
$ ./hello
Hello, world!
```

Rust’s Compiler

Rust’s compiler is *rustc*.

To see all of its options:

```
rustc --help
```

or

https://doc.rust-lang.org/rustc/command-line-arguments.html

But we don’t usually use *rustc* directly!

Cargo

Cargo is Rust’s build system and package manager.

It **checks**, **compiles**, **executes** and **tests** your code.

It handles **packages** and **dependencies**.

Creating a New Package

The cargo new command creates a new package.

```
$ ls
$ cargo new hello
$ ls
hello
$ tree hello/
hello/
├── Cargo.toml
└── src
    └── main.rs

1 directory, 2 files
```

Cargo.toml

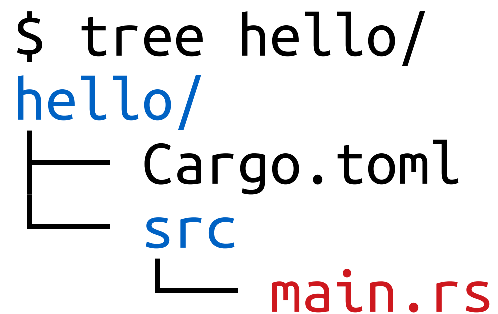
This **configuration** file contains information about the package and its dependencies.

```
$ tree hello/
hello/
├── Cargo.toml
└── src
    └── main.rs

1 directory, 2 files
```

```
[package]
name = "hello"
version = "0.1.0"
authors = ["Your Name <you@example.com>"]
edition = "2018"

[dependencies]
```

The *src* Directory

The ***src*** directory must contain all the source files.
A default main file is generated.



```
$ tree hello/
hello/
├── Cargo.toml
└── src
    └── main.rs

1 directory, 2 files
```

Default main file:

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

Version Control System (VCS)

The cargo new command initializes a Git repository.

```
$ cd hello/
$ ls
Cargo.toml  src
$ ls -a
.  ..  Cargo.toml  .git  .gitignore  src
```

Use cargo new --vcs none to disable this option:

```
$ cargo new --vcs none hello
```

Or, create a “~/.cargo/config” file:

```
[cargo-new]
vcs = "none"
```

Compiling

The cargo build command generates an executable file.

```
$ cargo build
   Compiling hello v0.1.0 (/home/david/hello)
    Finished dev [unoptimized + debuginfo] target(s) in 0.23s
$ ls
Cargo.lock  Cargo.toml  src  target
```

Two new items:

- The Cargo.lock file.
- The target directory.

Cargo.lock

“This file keeps track of the exact versions of dependencies in your project.”(1)

“You won’t ever need to change this file manually; Cargo manages its contents for you.”(1)

(1) https://doc.rust-lang.org/book/ch01-03-hello-cargo.html#building-and-running-a-cargo-project

The *target* directory

This directory contains the binary and intermediate files generated by the compiler.

By default, the *debug* mode is used.

- The compilation is faster.
- The execution is slower.
- The executable file contains some debug information.

```
$ ls target/
debug
$ ls target/debug/
build  deps  examples  hello  hello.d  incremental  native
$ target/debug/hello # We can execute the program this way
Hello, world!        # but it is not so common.
```

Running

The cargo run command *compiles* and *runs* your code.

```
$ cargo run
   Compiling hello v0.1.0 (/home/david/hello)
    Finished dev [unoptimized + debuginfo] target(s) in 0.44s
     Running `target/debug/hello`
Hello, world!
```

You can pass some arguments to the executable file.
Use the '--' separator: cargo run -- <list of arguments>

Example with three arguments:

```
$ cargo run -- arg1 arg2 arg3
```

Checking

The cargo check command checks your code.

```
$ cargo check
    Checking hello v0.1.0 (/home/david/hello)
    Finished dev [unoptimized + debuginfo] target(s) in 0.23s
$ ls target/debug/
build  deps  examples  incremental  native
```

cargo check and cargo build are similar:

- **cargo build** generates an executable file.
- **cargo check** does not generate any object or executable files.

→ cargo check is faster.

Cleaning

The cargo clean command cleans your package directory.

It deletes your target directory.

```
$ ls
Cargo.lock  Cargo.toml  src  target
$ cargo clean
$ ls
Cargo.lock  Cargo.toml  src
```

Building for Release

"When your project is finally ready for release, you can use cargo build --release to compile it with optimizations.

This command will create an executable in target/release instead of target/debug.

The optimizations make your Rust code run faster, but turning them on lengthens the time it takes for your program to compile."(1)

(1) https://doc.rust-lang.org/book/ch01-03-hello-cargo.html#building-for-release

Unchanged and Updated Files

Cargo is smart.

It compiles files that have been updated only.

It does not compile unchanged file.

```
$ cargo clean
$ cargo build
   Compiling hello v0.1.0 (/home/david/hello)
    Finished dev [unoptimized + debuginfo] target(s) in 0.44s
$ cargo build
    Finished dev [unoptimized + debuginfo] target(s) in 0.01s
```

Cargo’s Help: *cargo -h*

```
$ cargo -h
Rust's package manager

USAGE:
    cargo [OPTIONS] [SUBCOMMAND]

OPTIONS:
    -V, --version           Print version info and exit
        --list              List installed commands
        --explain <CODE>    Run `rustc --explain CODE`
    -v, --verbose           Use verbose output (-vv very verbose/build.rs output)

... snip ...

Some common cargo commands are (see all commands with --list):
    build       Compile the current package
    check       Analyze the current package and report errors, but don't build object files
    clean       Remove the target directory
    doc         Build this package's and its dependencies' documentation
    new         Create a new cargo package
    init        Create a new cargo package in an existing directory

... snip ...

See 'cargo help <command>' for more information on a specific command.
```

Cargo’s Help: *cargo help <command>*

```
$ cargo help new
cargo-new
Create a new cargo package at <path>

USAGE:
    cargo new [OPTIONS] <path>

OPTIONS:
        --registry <REGISTRY>    Registry to use
        --vcs <VCS>              Initialize a new repository for the given version ...
        --bin                    Use a binary (application) template [default] ...
        --lib                    Use a library template
        --edition <YEAR>         Edition to set for the crate generated
        --name <NAME>            Set the resulting package name, defaults to the ...
    -v, --verbose                Use verbose output (-vv very verbose/build.rs output)
    -q, --quiet                  No output printed to stdout
        --color <WHEN>           Coloring: auto, always, never
        --frozen                 Require Cargo.lock and cache are up to date
        --locked                 Require Cargo.lock is up to date
    -Z <FLAG>...                 Unstable (nightly-only) flags to Cargo, see ...
    -h, --help                   Prints help information

ARGS:
    <path>
```

Crates

A package contains one or more crates.

A crate is either a binary crate or a library crate:

- A ***binary crate*** generates an executable file.
- A ***library crate*** generates a library.

A package can contain:

- Any number of ***binary crates***.
- And zero ***library crates*** or just one.

Binary Crates

A package contains a binary crate when the src directory contains a main.rs file.

→ The cargo new instruction generates a binary crate by default.

```
cargo new hello
```

=

```
cargo new --bin hello
```

Binary Crates

```
cargo new hello
```

The name of the **package** is hello.

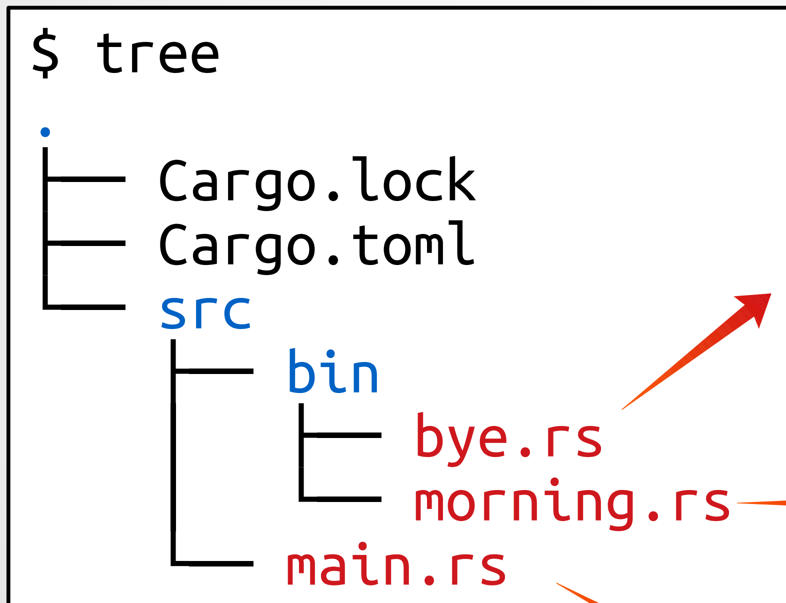
The name of the **binary crate** is hello.

src/main.rs is the crate root.

Multiple Binary Crates

A package can contain multiple binary crates.

Use the src/bin/ directory.



bye.rs

```
fn main() {
    println!("Good bye, world!");
}
```

morning.rs

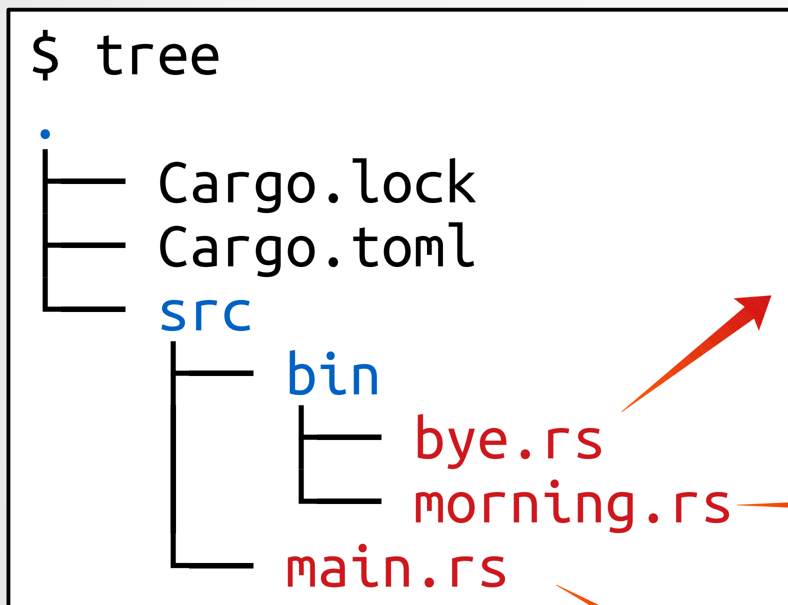
```
fn main() {
    println!("Good morning, world!");
}
```

main.rs

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

Multiple Binary Crates

This package has three crates.



The **"bye"** crate.

The **"morning"** crate.

The **"hello"** crate (the **crate root**).

Multiple Binary Crates

```
cargo build --bin hello
```

Builds hello *(main.rs)*

```
cargo build --bin bye
```

Builds bye *(bye.rs)*

```
cargo build --bin morning
```

Builds morning *(morning.rs)*

```
cargo build --bins
```

```
cargo build
```

Builds all crates

The same goes for cargo check.

```
$ tree
.
├── Cargo.lock
├── Cargo.toml
└── src
    ├── bin
    │   ├── bye.rs
    │   └── morning.rs
    └── main.rs
```

Multiple Binary Crates

```
$ cargo run
error: `cargo run` requires that a package only have one
executable; use the `--bin` option to specify which one to run
available binaries: hello, morning, bye
$ cargo run -q --bin hello
Hello, world!
$ cargo run -q --bin bye
Good bye, world!
$ cargo run -q --bin morning
Good morning, world!
```

```
$ tree
.
├── Cargo.lock
├── Cargo.toml
└── src
    ├── bin
    │   ├── bye.rs
    │   └── morning.rs
    └── main.rs
```

-q is the “quiet” option.
It prints only the output of the executable file.

Library Crates

A package contains a binary crate when the src directory contains a lib.rs file.

```
cargo new --lib mylib
```

The name of the **package** is mylib.

The name of the **library crate** is mylib.

src/lib.rs is the crate root.

Library Crates

```
$ cargo new --lib mylib
     Created library `mylib` package
$ ls
mylib
$ tree mylib/
mylib/
├── Cargo.toml
└── src
    └── lib.rs

1 directory, 2 files
```

Default lib file:

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

Library Crates

A library crate can be built, checked and tested but not run.

A package can contain a library crate and multiple binary crates.

A library crate can be used by the binary crates of the same package and also by any external library or binary crates.