[Customizing Builds with Release Profiles 1](#_Toc499037503)

[Publishing a Crate to Crates.io 3](#_Toc499037504)

[Making Useful Documentation Comments 3](#_Toc499037505)

[Commonly Used Sections 4](#_Toc499037506)

[Documentation Comments as Tests 5](#_Toc499037507)

[Commenting Contained Items 5](#_Toc499037508)

[Exporting a Convenient Public API with pub use 6](#_Toc499037509)

[Setting up a Crates.io Account 11](#_Toc499037510)

[Before Publishing a New Crate 12](#_Toc499037511)

[Publishing to Crates.io 13](#_Toc499037512)

[Publishing a New Version of an Existing Crate 14](#_Toc499037513)

[Removing Versions from Crates.io with cargo yank 14](#_Toc499037514)

[Cargo Workspaces 15](#_Toc499037515)

[Specifying Workspace Dependencies 15](#_Toc499037516)

[Creating the Second Crate in the Workspace 16](#_Toc499037517)

[Depending on an External Crate in a Workspace 18](#_Toc499037518)

[Adding a Test to a Workspace 19](#_Toc499037519)

[Installing Binaries from Crates.io with cargo install 21](#_Toc499037520)

[Extending Cargo with Custom Commands 22](#_Toc499037521)

[Summary 22](#_Toc499037522)

14

More About Cargo and Crates.io

So far we’ve used only the most basic features of Cargo to build, run, and test our code, but it can do a lot more. In this chapter, we’ll discuss some of its other, more advanced features to show you how to:

Customize your build through release profiles

Publish libraries on https://crates.io/

Organize large projects with workspaces

Install binaries from https://crates.io/

Extend Cargo using custom commands

Cargo can do even more than what we cover in this chapter, so for a full explanation of all its features, see its documentation at <https://doc.rust-lang.org/cargo/>.

Customizing Builds with Release Profiles

In Rust, release profiles are predefined and customizable profiles with different configurations that allow a programmer to have more control over various options for compiling code. Each profile is configured independently of the others.

Cargo has two main profiles: the dev profile Cargo uses when you run cargo build and the release profile Cargo uses when you run cargo build --release. The dev profile is defined with good defaults for developing, and the release profile has good defaults for release builds.

These profile names might be familiar from the output of your builds, which shows the profile used in the build:

$ cargo build

Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs

$ cargo build --release

Finished release [optimized] target(s) in 0.0 secs

The dev and release shown in this build output indicate that the compiler is using different profiles.

Cargo has default settings for each of the profiles that apply when there aren’t any [profile.\*] sections in the project’s Cargo.toml file. By adding [profile.\*] sections for any profile we want to customize, we can override any subset of the default settings. For example, here are the default values for the opt-level setting for the dev and release profiles:

Cargo.toml

[profile.dev]

opt-level = 0

[profile.release]

opt-level = 3

The opt-level setting controls the number of optimizations Rust will apply to your code with a range of zero to three. Applying more optimizations extends compiling time, so if you’re in development and compiling your code often, you want faster compiling even at the expense of the resulting code running slower. That is the reason the default opt-level for dev is 0. When you’re ready to release your code, it’s best to spend more time compiling. You’ll only compile in release mode once and run the compiled program many times, so release mode trades longer compile time for code that runs faster. That is the reason the default opt-level for the release profile is 3.

We can override any default setting by adding a different value for it in Cargo.toml. For example, if we want to use optimization level 1 in the development profile, we can add these two lines to our project’s Cargo.toml file:

Cargo.toml

[profile.dev]

opt-level = 1

This code overrides the default setting of 0. Now when we run cargo build, Cargo will use the defaults for the dev profile plus our customization to opt-level. Because we set opt-level to 1, Cargo will apply more optimizations than the default, but not as many as a release build.

For the full list of configuration options and defaults for each profile, see Cargo’s documentation at <https://doc.rust-lang.org/cargo/>.

Publishing a Crate to Crates.io

We’ve used packages from https://crates.io/ as dependencies of our project, but you can also share your code for other people to use by publishing your own packages. The crate registry at https://crates.io/ distributes the source code of your packages, so it primarily hosts code that is open source.

Rust and Cargo have features that help make your published package easier for people to use and to find in the first place. We’ll talk about some of these features next, and then explain how to publish a package.

Making Useful Documentation Comments

Accurately documenting your packages will help other users know how and when to use them, so it’s worth spending time writing documentation. In Chapter 3, we discussed how to comment Rust code using //. Rust also has a particular kind of comment for documentation, which is known conveniently as documentation comments, that will generate HTML documentation. The HTML displays the contents of documentation comments for public API items intended for programmers interested in knowing how to use your crate as opposed to how your crate is implemented.

prod: xref ok

Documentation comments use /// instead of // and support Markdown notation for formatting the text if you want to use it. You place documentation comments just before the item they’re documenting. Listing 14-1 shows documentation comments for an add\_one function in a crate named my\_crate:

src/lib.rs

/// Adds one to the number given.

///

/// # Examples

///

/// ```

/// let five = 5;

///

/// assert\_eq!(6, my\_crate::add\_one(5));

/// ```

pub fn add\_one(x: i32) -> i32 {

x + 1

}

Listing 14-1: A documentation comment for a function

Here, we give a description of what the add\_one function does, start a section with the heading Examples, and then provide code that demonstrates how to use the add\_one function. We can generate the HTML documentation from this documentation comment by running cargo doc. This command runs the rustdoc tool distributed with Rust and puts the generated HTML documentation in the target/doc directory.

For convenience, running cargo doc --open will build the HTML for your current crate’s documentation (as well as the documentation for all of your crate’s dependencies) and open the result in a web browser. Navigate to the add\_one function and you’ll see how the text in the documentation comments is rendered, as shown in Figure 14-1:



Figure 14-1: HTML documentation for the add\_one function

au to add image

Commonly Used Sections

We used the # Examples Markdown heading in Listing 14-1 to create a section in the HTML with the title “Examples.” Some other sections that crate authors commonly use in their documentation include:

Panics: The scenarios in which the function being documented could panic!. Callers of the function who don’t want their programs to panic should make sure they don’t call the function in these situations.

Errors: If the function returns a Result, describing the kinds of errors that might occur and what conditions might cause those errors to be returned can be helpful to callers so they can write code to handle the different kinds of errors in different ways.

Safety: If the function is unsafe to call (we discuss unsafety in Chapter 19), there should be a section explaining why the function is unsafe and covering the invariants that the function expects callers to uphold.

prod: confirm xref

Most documentation comment sections don’t need all of these sections, but it’s a good list to check to remind you of the aspects of your code that people calling your code will be interested in knowing about.

Documentation Comments as Tests

Adding examples in code blocks in your documentation comments can clearly demonstrate how to use your library, and doing so has an additional bonus: running cargo test will run the code examples in your documentation as tests! Nothing is better than documentation with examples. But nothing is worse than examples that don’t work because the code has changed since the documentation was written. Run cargo test with the documentation for the add\_one function from Listing 14-1; you should see a section in the test results like this:

Doc-tests my\_crate

running 1 test

test src/lib.rs - add\_one (line 5) ... ok

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

Now change either the function or the example so the assert\_eq! in the example panics. Run cargo test again; you’ll see that the doc tests catch that the example and the code are out of sync from one another!

Commenting Contained Items

Another style of doc comment, //!, adds documentation to the item that contains the comments rather than adding documentation to the items following the comments. We typically use these doc comments inside the crate root file (src/lib.rs by convention) or inside a module to document the crate or the module as a whole.

For example, if we want to add documentation that describes the purpose of the my\_crate crate that contains the add\_one function, we can add documentation comments that start with //! to the beginning of the src/lib.rs file, as shown in Listing 14-2:

src/lib.rs

//! # My Crate

//!

//! `my\_crate` is a collection of utilities to make performing certain

//! calculations more convenient.

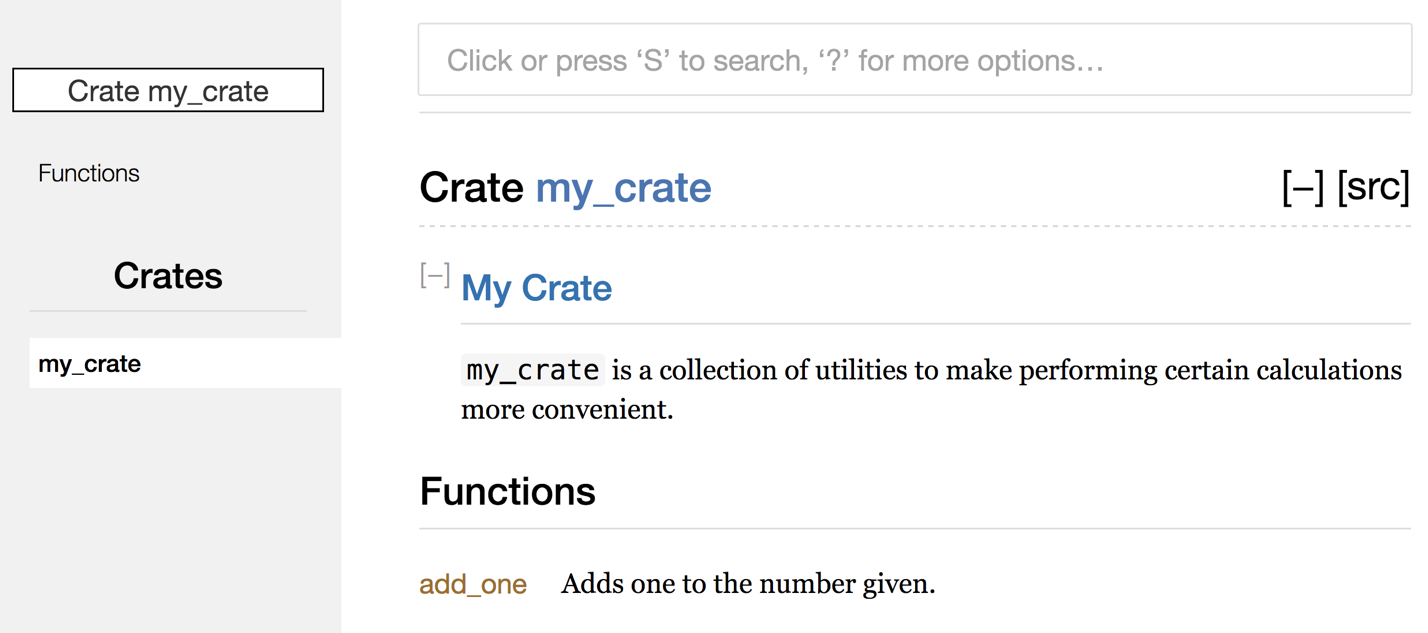
/// Adds one to the number given.

// --snip--

Listing 14-2: Documentation for the my\_crate crate as a whole

Notice there isn’t any code after the last line that begins with //!. Because we started the comments with //! instead of ///, we’re documenting the item that contains this comment rather than an item that follows this comment. In this case, the item that contains this comment is the src/lib.rs file, which is the crate root. These comments describe the entire crate.

When we run cargo doc --open, these comments will display on the front page of the documentation for my\_crate above the list of public items in the crate, as shown in Figure 14-2:



AU: add image

Figure 14-2: Rendered documentation for my\_crate including the comment describing the crate as a whole

Documentation comments within items are useful for describing crates and modules especially. Use them to explain the purpose of the container overall to help your crate users understand your organization.

Exporting a Convenient Public API with pub use

In Chapter 7, we covered how to organize our code into modules using the mod keyword, how to make items public using the pub keyword, and how to bring items into a scope with the use keyword. However, the structure that makes sense to you while you’re developing a crate might not be very convenient for your users. You might want to organize your structs in a hierarchy containing multiple levels, but people who want to use a type you’ve defined deep in the hierarchy might have trouble finding out that those types exist. They might also be annoyed at having to enter use my\_crate::some\_module::another\_module::UsefulType; rather than use my\_crate::UsefulType;.

prod: xref OK

The structure of your public API is a major consideration when publishing a crate. People who use your crate are less familiar with the structure than you are and might have difficulty finding the pieces they want to use if your crate has a large module hierarchy.

The good news is that if the structure isn’t convenient for others to use from another library, you don’t have to rearrange your internal organization: instead, you can re-export items to make a public structure that’s different than your private structure by using pub use. Re-exporting takes a public item in one location and makes it public in another location, as if it was defined in the other location instead.

For example, say we made a library named art for modeling artistic concepts. Within this library are two modules: a kinds module containing two enums named PrimaryColor and SecondaryColor, and a utils module containing a function named mix, as shown in Listing 14-3:

src/lib.rs

//! # Art

//!

//! A library for modeling artistic concepts.

pub mod kinds {

/// The primary colors according to the RYB color model.

pub enum PrimaryColor {

Red,

Yellow,

Blue,

}

/// The secondary colors according to the RYB color model.

pub enum SecondaryColor {

Orange,

Green,

Purple,

}

}

pub mod utils {

use kinds::\*;

/// Combines two primary colors in equal amounts to create

/// a secondary color.

pub fn mix(c1: PrimaryColor, c2: PrimaryColor) -> SecondaryColor {

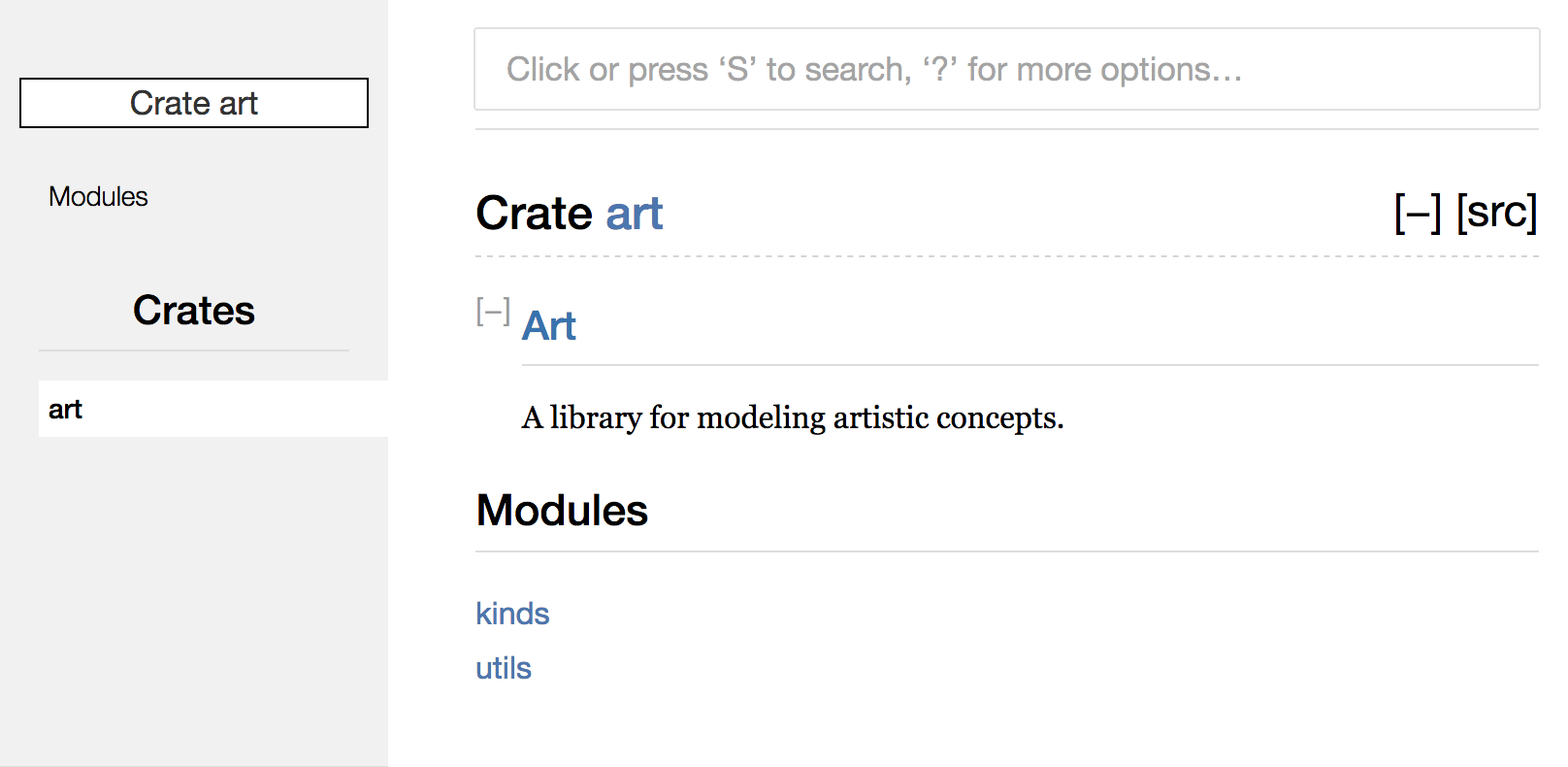
// --snip--

}

}

Listing 14-3: An art library with items organized into kinds and utils modules

Figure 14-3 shows what the front page of the documentation for this crate generated by cargo doc would look like:



au to add picture

Figure 14-3: Front page of the documentation for art that lists the kinds and utils modules

Note that the PrimaryColor and SecondaryColor types aren’t listed on the front page, nor is the mix function. We have to click kinds and utils to see them.

Another crate that depends on this library would need use statements that import the items from art, including specifying the module structure that’s currently defined. Listing 14-4 shows an example of a crate that uses the PrimaryColor and mix items from the art crate:

src/main.rs

extern crate art;

use art::kinds::PrimaryColor;

use art::utils::mix;

fn main() {

let red = PrimaryColor::Red;

let yellow = PrimaryColor::Yellow;

mix(red, yellow);

}

Listing 14-4: A crate using the art crate’s items with its internal structure exported

The author of the code in Listing 14-4, which uses the art crate, had to figure out that PrimaryColor is in the kinds module and mix is in the utils module. The module structure of the art crate is more relevant to developers working on the art crate than developers using the art crate. The internal structure that organizes parts of the crate into the kinds module and the utils module doesn’t contain any useful information for someone trying to understand how to use the art crate. Instead, the art crate’s module structure causes confusion because developers have to figure out where to look, and the structure is inconvenient because developers must specify the module names in the use statements.

To remove the internal organization from the public API, we can modify the art crate code in Listing 14-3 to add pub use statements to re-export the items at the top level, as shown in Listing 14-5:

src/lib.rs

//! # Art

//!

//! A library for modeling artistic concepts.

pub use kinds::PrimaryColor;

pub use kinds::SecondaryColor;

pub use utils::mix;

pub mod kinds {

// --snip--

}

pub mod utils {

// --snip--

}

Listing 14-5: Adding pub use statements to re-export items

The API documentation that cargo doc generates for this crate will now list and link re-exports on the front page, as shown in Figure 14-4, which makes the PrimaryColor and SecondaryColor types and the mix function easier to find:

Au to add figure

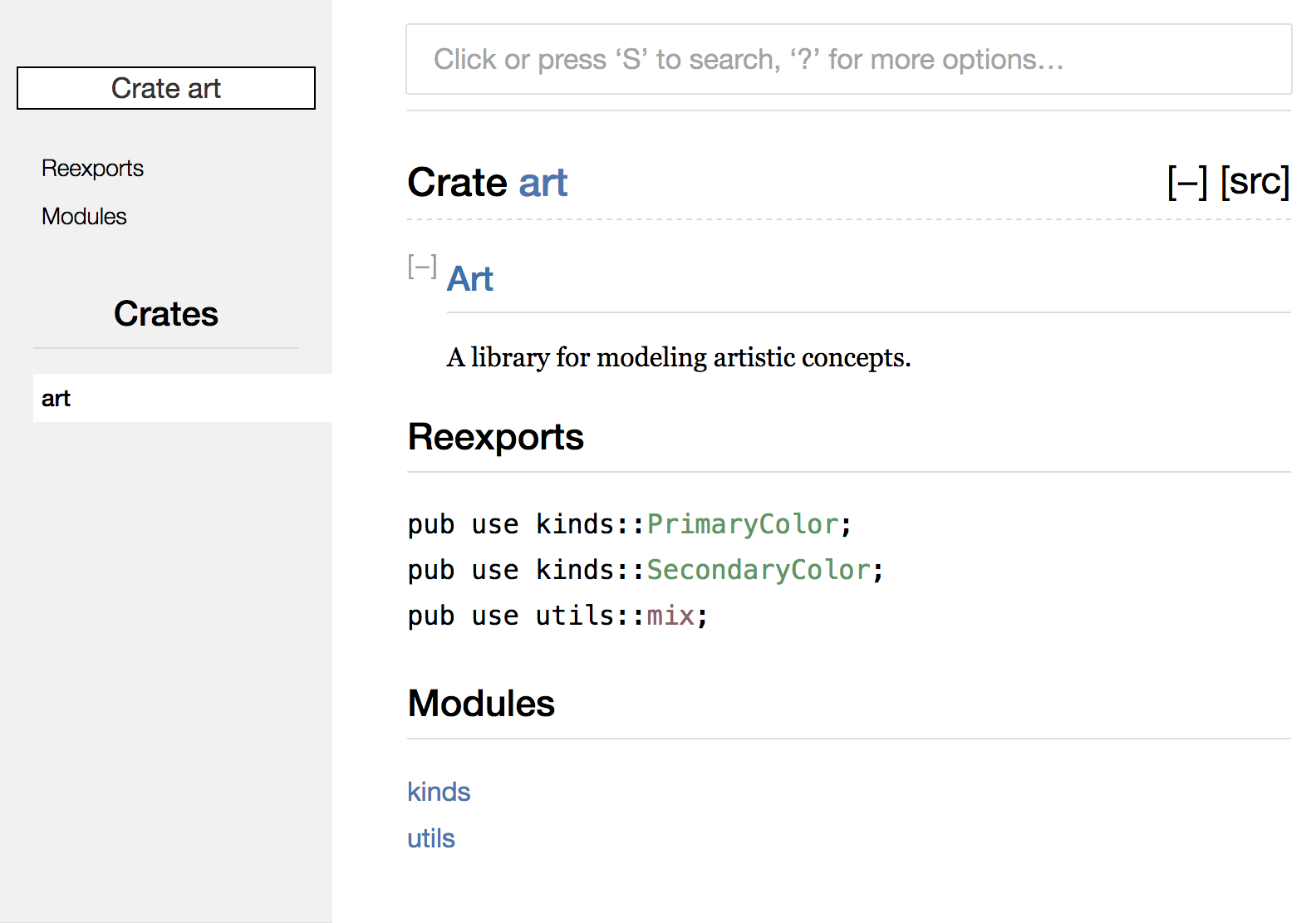


Figure 14-4: The front page of the documentation for art that lists the re-exports

The art crate users can still see and use the internal structure from Listing 14-3 as demonstrated in Listing 14-4, or they can use the more convenient structure in Listing 14-5, as shown in Listing 14-6:

src/main.rs

extern crate art;

use art::PrimaryColor;

use art::mix;

fn main() {

// --snip--

}

Listing 14-6: A program using the re-exported items from the art crate

In cases where there are many nested modules, re-exporting the types at the top level with pub use can make a significant difference in the experience of people who use the crate.

Creating a useful public API structure is more of an art than a science, and you can iterate to find the API that works best for your users. Choosing pub use gives you flexibility in how you structure your crate internally and decouples that internal structure with what you present to your users. Look at some of the code of crates you’ve installed to see if their internal structure differs from their public API.

Setting Up a Crates.io Account

Before you can publish any crates, you need to create an account on https://crates.io/ and get an API token. To do so, visit the home page at <https://crates.io>/ and log in via a GitHub account: the GitHub account is currently a requirement, but the site might support other ways of creating an account in the future. Once you’re logged in, visit your account settings at <https://crates.io/me>/ and retrieve your API key. Then run the cargo login command with your API key, like this:

$ cargo login abcdefghijklmnopqrstuvwxyz012345

This command will inform Cargo of your API token and store it locally in ~/.cargo/credentials. Note that this token is a secret: do not share it with anyone else. If you do share it with anyone for any reason, you should revoke it and generate a new token on https://crates.io/.

Before Publishing a New Crate

Now that you have an account, let’s say you have a crate you want to publish. Before publishing, you’ll need to add some metadata to your crate by adding it to the [package] section of the crate’s Cargo.toml file.

Your crate will need a unique name. While you’re working on a crate locally, you can name a crate whatever you’d like. However, crate names on https://crates.io/ are allocated on a first-come, first-served basis. Once a crate name is taken, no one else can publish a crate with that name. Search for the name you want to use on the site to find out if it has been used. If it hasn’t, edit the name in the Cargo.toml file under [package] to use the name for publishing, like so:

Cargo.toml

[package]

name = "guessing\_game"

Even if you’ve chosen a unique name, when you run cargo publish to publish the crate at this point, you’ll get a warning and then an error:

$ cargo publish

Updating registry `https://github.com/rust-lang/crates.io-index`

warning: manifest has no description, license, license-file, documentation,

homepage or repository.

--snip--

error: api errors: missing or empty metadata fields: description, license.

The reason is that you’re missing some crucial information: a description and license are required so people will know what your crate does and under what terms they can use it. To rectify this error, you need to include this information in the Cargo.toml file.

Add a description that is just a sentence or two, because it will appear with your crate in search results. For the license field, you need to give a license identifier value. The Linux Foundation’s Software Package Data Exchange (SPDX) at <http://spdx.org/licenses/> lists the identifiers you can use for this value. For example, to specify that you’ve licensed your crate using the MIT License, add the MIT identifier:

Cargo.toml

[package]

name = "guessing\_game"

license = "MIT"

If you want to use a license that doesn’t appear in the SPDX, you need to place the text of that license in a file, include the file in your project, and then use license-file to specify the name of that file instead of using the license key.

Guidance on which license is appropriate for your project is beyond the scope of this book. Many people in the Rust community license their projects in the same way as Rust by using a dual license of MIT OR Apache-2.0, which demonstrates that you can also specify multiple license identifiers separated by OR to have multiple licenses for your project.

With a unique name, the version, the author details that cargo new added when you created the crate, your description, and a license added, the Cargo.toml file for a project that is ready to publish might look like this:

Cargo.toml

[package]

name = "guessing\_game"

version = "0.1.0"

authors = ["Your Name <you@example.com>"]

description = "A fun game where you guess what number the computer has chosen."

license = "MIT OR Apache-2.0"

[dependencies]

Cargo’s documentation at <https://doc.rust-lang.org/cargo>/ describes other metadata you can specify to ensure others can discover and use your crate more easily!

Publishing to Crates.io

Now that you’ve created an account, saved your API token, chosen a name for your crate, and specified the required metadata, you’re ready to publish! Publishing a crate uploads a specific version to https://crates.io/ for others to use.

Be careful when publishing a crate because a publish is permanent. The version can never be overwritten, and the code cannot be deleted. One major goal of https://crates.io/ is to act as a permanent archive of code so that builds of all projects that depend on crates from https://crates.io/ will continue to work. Allowing version deletions would make fulfilling that goal impossible. However, there is no limit to the number of crate versions you can publish.

Run the cargo publish command again. It should succeed now:

$ cargo publish

Updating registry `https://github.com/rust-lang/crates.io-index`

Packaging guessing\_game v0.1.0 (file:///projects/guessing\_game)

Verifying guessing\_game v0.1.0 (file:///projects/guessing\_game)

Compiling guessing\_game v0.1.0

(file:///projects/guessing\_game/target/package/guessing\_game-0.1.0)

Finished dev [unoptimized + debuginfo] target(s) in 0.19 secs

Uploading guessing\_game v0.1.0 (file:///projects/guessing\_game)

Congratulations! You’ve now shared your code with the Rust community, and anyone can easily add your crate as a dependency of their project.

Publishing a New Version of an Existing Crate

When you’ve made changes to your crate and are ready to release a new version, you change the version value specified in your Cargo.toml file and republish. Use the Semantic Versioning rules at http://semver.org/ to decide what an appropriate next version number is based on the kinds of changes you’ve made. Then run cargo publish to upload the new version.

Removing Versions from Crates.io with cargo yank

Although you can’t remove previous versions of a crate, you can prevent any future projects from adding them as a new dependency. This is useful when a crate version is broken for one reason or another. In such situations, Cargo supports yanking a crate version.

Yanking a version prevents new projects from starting to depend on that version while allowing all existing projects that depend on it to continue to download and depend on that version. Essentially, a yank means that all projects with a Cargo.lock will not break, and any future Cargo.lock files generated will not use the yanked version.

To yank a version of a crate, run cargo yank and specify which version you want to yank:

$ cargo yank --vers 1.0.1

By adding --undo to the command, you can also undo a yank and allow projects to start depending on a version again:

$ cargo yank --vers 1.0.1 --undo

A yank does not delete any code. For example, the yank feature is not intended for deleting accidentally uploaded secrets. If that happens, you must reset those secrets immediately.

Cargo Workspaces

In Chapter 12, we built a package that included a binary crate and a library crate. As your project develops, you might find that the library crate continues to get bigger and you want to split up your package further into multiple library crates. In this situation, Cargo offers a feature called workspaces that can help manage multiple related packages that are developed in tandem.

prod: confirm xref

A workspace is a set of packages that share the same Cargo.lock and output directory. Let’s make a project using a workspace and use trivial code so we can concentrate on the structure of the workspace. There are multiple ways to structure a workspace; we’re going to show a common way. We’ll have a workspace containing a binary and two libraries. The binary will provide the main functionality to be used as a command line tool, and it will depend on the two libraries. One library will provide an add\_one function, and a second library will provide an add\_two function. These three crates will be part of the same workspace. We’ll start by creating a new directory for the workspace:

$ mkdir add

$ cd add

In the add directory, create a Cargo.toml file. This is the Cargo.toml file that configures the entire workspace. It won’t have a [package] section or metadata we’ve seen in other Cargo.toml files. Instead, we’ll start with a [workspace] section and add a member to the workspace by specifying the path adder, which is where we’ll put our binary crate:

Cargo.toml

[workspace]

members = [

"adder",

]

Next, we’ll create the adder binary crate by running cargo new within the add directory:

$ cargo new --bin adder

Created binary (application) `adder` project

At this point, we can build the workspace by running cargo build. The files in your add directory should look like this:

├── Cargo.lock

├── Cargo.toml

├── adder

│ ├── Cargo.toml

│ └── src

│ └── main.rs

└── target

The workspace has one target directory at the top level; the adder crate doesn’t have its own target directory. Even if we go into the adder directory and run cargo build, the compiled artifacts end up in add/target rather than add/adder/target. The crates in a workspace are meant to depend on each other. If each crate had its own target directory, each crate in the workspace would have to recompile each of the other crates in the workspace to have the artifacts in its own target directory. By sharing one target directory, the crates in the workspace can avoid rebuilding the other crates in the workspace more than necessary.

Creating the Second Crate in the Workspace

Next, let’s specify another member crate in the workspace. This crate will be in the add-one directory, so change the top-level Cargo.toml to have the add-one path as well:

Cargo.toml

[workspace]

members = [

"adder",

"add-one",

]

Then generate a new library crate named add-one:

$ cargo new add-one

Created library `add-one` project

Your add directory should now have these directories and files:

├── Cargo.lock

├── Cargo.toml

├── add-one

│ ├── Cargo.toml

│ └── src

│ └── lib.rs

├── adder

│ ├── Cargo.toml

│ └── src

│ └── main.rs

└── target

In the add-one/src/lib.rs file, let’s add an add\_one function:

add-one/src/lib.rs

pub fn add\_one(x: i32) -> i32 {

x + 1

}

Now that we have a library crate in the workspace, let’s have the binary crate adder depend on the library crate add-one. First, we’ll need to add a path dependency on add-one to adder/Cargo.toml:

adder/Cargo.toml

[dependencies]

add-one = { path = "../add-one" }

Crates in a workspace don’t have to depend on each other, so we still need to be explicit about the dependency relationships between the crates in a workspace.

Next, let’s use the add\_one function from the add-one crate in the adder crate. Open the adder/src/main.rs file and add an extern crate line at the top to bring the new add-one library crate into scope. Then change the main function to call the add\_one function, as in Listing 14-7:

adder/src/main.rs

extern crate add\_one;

fn main() {

let num = 10;

println!("Hello, world! {} plus one is {}!", num, add\_one::add\_one(num));

}

Listing 14-7: Using the add-one library crate from the adder crate

Let’s build the workspace by running cargo build in the add directory!

$ cargo build

Compiling add-one v0.1.0 (file:///projects/add/add-one)

Compiling adder v0.1.0 (file:///projects/add/adder)

Finished dev [unoptimized + debuginfo] target(s) in 0.68 secs

To run the binary crate from the top-level add directory, we need to specify which package in the workspace we want to use by using the -p argument and the package name with cargo run:

$ cargo run -p adder

Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs

Running `target/debug/adder`

Hello, world! 10 plus one is 11!

This runs the code in adder/src/main.rs, which depends on the add-one crate.

Depending on an External Crate in a Workspace

Notice that the workspace has only one Cargo.lock file at the top level of the workspace rather than having a Cargo.lock in each crate’s directory. This ensures that all crates are using the same version of all dependencies. If we add the rand crate to the adder/Cargo.toml and add-one/Cargo.toml files, Cargo will resolve both of those to one version of rand and record that in the one Cargo.lock. Making all crates in the workspace use the same dependencies means the crates in the workspace will always be compatible with each other. Let’s add the rand crate to the [dependencies] section in the add-one/Cargo.toml file to be able to use the rand crate in the add-one crate:

add-one/Cargo.toml

[dependencies]

rand = "0.3.14"

We can now add extern crate rand; to the add-one/src/lib.rs file, and building the whole workspace by running cargo build in the add directory will bring in and compile the rand crate:

$ cargo build

Updating registry `https://github.com/rust-lang/crates.io-index`

Downloading rand v0.3.14

--snip--

Compiling rand v0.3.14

Compiling add-one v0.1.0 (file:///projects/add/add-one)

Compiling adder v0.1.0 (file:///projects/add/adder)

Finished dev [unoptimized + debuginfo] target(s) in 10.18 secs

The top-level Cargo.lock now contains information about the dependency of add-one on rand. However, even though rand is used somewhere in the workspace, we can’t use it in other crates in the workspace unless we add rand to their Cargo.toml files as well. For example, if we add extern crate rand; to the adder/src/main.rs file for the adder crate, we’ll get an error:

$ cargo build

Compiling adder v0.1.0 (file:///projects/add/adder)

error: use of unstable library feature 'rand': use `rand` from crates.io (see issue #27703)

--> adder/src/main.rs:1:1

|

1 | extern crate rand;

To fix this, edit the Cargo.toml file for the adder crate and indicate that rand is a dependency for that crate as well. Building the adder crate will add rand to the list of dependencies for adder in Cargo.lock, but no additional copies of rand will be downloaded. Cargo has ensured that any crate in the workspace using the rand crate will be using the same version. Using the same version of rand across the workspace saves space because we won’t have multiple copies and ensures that the crates in the workspace will be compatible with each other.

Adding a Test to a Workspace

For another enhancement, let’s add a test of the add\_one::add\_one function within the add\_one crate:

add-one/src/lib.rs

pub fn add\_one(x: i32) -> i32 {

x + 1

}

#[cfg(test)]

mod tests {

use super::\*;

#[test]

fn it\_works() {

assert\_eq!(3, add\_one(2));

}

}

Now run cargo test in the top-level add directory:

$ cargo test

Compiling add-one v0.1.0 (file:///projects/add/add-one)

Compiling adder v0.1.0 (file:///projects/add/adder)

Finished dev [unoptimized + debuginfo] target(s) in 0.27 secs

Running target/debug/deps/add\_one-f0253159197f7841

running 1 test

test tests::it\_works ... ok

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

Running target/debug/deps/adder-f88af9d2cc175a5e

running 0 tests

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

Doc-tests add-one

running 0 tests

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

The first section of the output shows that the it\_works test in the add-one crate passed. The next section shows that 0 tests were found in the adder crate, and then the last section shows 0 documentation tests were found in the add-one crate. Running cargo test in a workspace structured like this one will run the tests for all the crates in the workspace.

We can also run tests for one particular crate in a workspace from the top-level directory by using the -p flag and specifying the name of the crate we want to test:

$ cargo test -p add-one

Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs

Running target/debug/deps/add\_one-b3235fea9a156f74

running 1 test

test tests::it\_works ... ok

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

Doc-tests add-one

running 0 tests

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

This output shows cargo test only ran the tests for the add-one crate and didn’t run the adder crate tests.

If you publish the crates in the workspace to https://crates.io/, each crate in the workspace will need to be published separately. The cargo publish command does not have an --all flag or a -p flag, so you must change to each crate’s directory and run cargo publish on each crate in the workspace to publish them.

For additional practice, add an add-two crate to this workspace in a similar way as the add-one crate!

As your project grows, consider using a workspace: it’s easier to understand smaller, individual components than one big blob of code. Keeping the crates in a workspace can make coordination between them easier if they are often changed at the same time.

Installing Binaries from Crates.io with cargo install

The cargo install command allows you to install and use binary crates locally. This isn’t intended to replace system packages; it’s meant to be a convenient way for Rust developers to install tools that others have shared on https://crates.io/. You can only install packages that have binary targets. A binary target is the runnable program that is created if the crate has a src/main.rs file or another file specified as a binary, as opposed to a library target that isn’t runnable on its own but is suitable for including within other programs. Usually, crates have information in the README file about whether a crate is a library, has a binary target, or both.

All binaries installed with cargo install are stored in the installation root’s bin folder. If you installed Rust using rustup.rs and don’t have any custom configurations, this directory will be $HOME/.cargo/bin. Ensure that directory is in your $PATH to be able to run programs you’ve installed with cargo install.

For example, in Chapter 12 we mentioned that there’s a Rust implementation of the grep tool called ripgrep for searching files. If we want to install ripgrep, we can run the following:

prod: confirm xref

$ cargo install ripgrep

Updating registry `https://github.com/rust-lang/crates.io-index`

Downloading ripgrep v0.3.2

--snip--

Compiling ripgrep v0.3.2

Finished release [optimized + debuginfo] target(s) in 97.91 secs

Installing ~/.cargo/bin/rg

The last line of the output shows the location and the name of the installed binary, which in the case of ripgrep is rg. As long as the installation directory is in your $PATH, as mentioned previously, you can then run rg --help and start using a faster, rustier tool for searching files!

Extending Cargo with Custom Commands

Cargo is designed so you can extend it with new subcommands without having to modify Cargo. If a binary in your $PATH is named cargo-something, you can run it as if it was a Cargo subcommand by running cargo something. Custom commands like this are also listed when you run cargo --list. Being able to use cargo install to install extensions and then run them just like the built-in Cargo tools is a super convenient benefit of Cargo’s design!

Summary

Sharing code with Cargo and https://crates.io/ is part of what makes the Rust ecosystem useful for many different tasks. Rust’s standard library is small and stable, but crates are easy to share, use, and improve on a timeline different from the language. Don’t be shy about sharing code that’s useful to you on https://crates.io/; it’s likely that it will be useful to someone else as well!