Appendix D - Useful Development Tools

In this appendix, we talk about some useful development tools that the Rust project provides. We’ll look at automatic formatting, quick ways to apply warning fixes, a linter, and integrating with IDEs.

Automatic Formatting with rustfmt

The rustfmt tool reformats your code according to the community code style. Many collaborative projects use rustfmt to prevent arguments about which style to use when writing Rust: everyone formats their code using the tool.

To install rustfmt, enter the following:

$ rustup component add rustfmt

This command gives you rustfmt and cargo-fmt, similar to how Rust gives you both rustc and cargo. To format any Cargo project, enter the following:

$ cargo fmt

Running this command reformats all the Rust code in the current crate. This should only change the code style, not the code semantics. For more information on rustfmt, see its documentation at https://github.com/rust-lang/rustfmt/.

Fix Your Code with rustfix

The rustfix tool is included with Rust installations and can automatically fix some compiler warnings. If you’ve written code in Rust, you’ve probably seen compiler warnings. For example, consider this code:

Filename: src/main.rs

fn do\_something() {}

fn main() {

for i in 0..100 {

do\_something();

}

}

Here, we’re calling the do\_something function 100 times, but we never use the variable i in the body of the for loop. Rust warns us about that:

$ cargo build

Compiling myprogram v0.1.0 (file:///projects/myprogram)

warning: unused variable: `i`

--> src/main.rs:4:9

|

4 | for i in 1..100 {

| ^ help: consider using `\_i` instead

|

= note: #[warn(unused\_variables)] on by default

Finished dev [unoptimized + debuginfo] target(s) in 0.50s

The warning suggests that we use \_i as a name instead: the underscore indicates that we intend for this variable to be unused. We can automatically apply that suggestion using the rustfix tool by running the command cargo fix:

$ cargo fix

Checking myprogram v0.1.0 (file:///projects/myprogram)

Fixing src/main.rs (1 fix)

Finished dev [unoptimized + debuginfo] target(s) in 0.59s

When we look at src/main.rs again, we’ll see that cargo fix has changed the code:

Filename: src/main.rs

fn do\_something() {}

fn main() {

for \_i in 0..100 {

do\_something();

}

}

The for loop variable is now named \_i, and the warning no longer appears.

You can also use the cargo fix command to transition your code between different Rust editions. Editions are covered in Appendix E.

More Lints with Clippy

The Clippy tool is a collection of lints to analyze your code to catch common mistakes and improve your Rust code.

To install Clippy, enter the following:

$ rustup component add clippy

To run Clippy’s lints on any Cargo project, enter the following:

$ cargo clippy

For example, say you write a program that uses an approximation of a mathematical constant, such as pi, as this program does:

Filename: src/main.rs

fn main() {

let x = 3.1415;

let r = 8.0;

println!("the area of the circle is {}", x \* r \* r);

}

Running cargo clippy on this project results in this error:

error: approximate value of `f{32, 64}::consts::PI` found. Consider using it directly

--> src/main.rs:2:13

|

2 | let x = 3.1415;

| ^^^^^^

|

= note: #[deny(clippy::approx\_constant)] on by default

= help: for further information visit https://rust-lang-nursery.github.io/rust-clippy/master/index.html#approx\_constant

This error lets you know that Rust has this constant defined more precisely, and that your program would be more correct if you used the constant instead. You would then change your code to use the PI constant. The following code doesn’t result in any errors or warnings from Clippy:

Filename: src/main.rs

fn main() {

let x = std::f64::consts::PI;

let r = 8.0;

println!("the area of the circle is {}", x \* r \* r);

}

For more information on Clippy, see its documentation at https://github.com/rust-lang/rust-clippy/.

IDE Integration Using the Rust Language Server

To help IDE integration, the Rust project distributes the Rust Language Server (rls). This tool speaks the Language Server Protocol described at http://langserver.org/, which is a specification for IDEs and programming languages to communicate with each other. Different clients can use the rls, such as the Rust plug-in for Visual Studio Code available from https://marketplace.visualstudio.com/items?itemName=rust-lang.rust.

To install the rls, enter the following:

$ rustup component add rls

Then install the language server support in your particular IDE; you’ll gain abilities such as autocompletion, jump to definition, and inline errors.

For more information on the rls, see its documentation at https://github.com/rust-lang/rls/.

Appendix E - Editions

In Chapter 1, you saw that cargo new adds a bit of metadata to your Cargo.toml file about an edition. This appendix talks about what that means!

The Rust language and compiler have a six-week release cycle, meaning users get a constant stream of new features. Other programming languages release larger changes less often; Rust releases smaller updates more frequently. After a while, all of these tiny changes add up. But from release to release, it can be difficult to look back and say, “Wow, between Rust 1.10 and Rust 1.31, Rust has changed a lot!”

Every two or three years, the Rust team produces a new Rust edition. Each edition brings together the features that have landed into a clear package with fully updated documentation and tooling. New editions ship as part of the usual six-week release process.

Editions serve different purposes for different people:

For active Rust users, a new edition brings together incremental changes into an easy-to-understand package.

For non-users, a new edition signals that some major advancements have landed, which might make Rust worth another look.

For those developing Rust, a new edition provides a rallying point for the project as a whole.

At the time of this writing, two Rust editions are available: Rust 2015 and Rust 2018. This book is written using Rust 2018 edition idioms.

The edition key in Cargo.toml indicates which edition the compiler should use for your code. If the key doesn’t exist, Rust uses 2015 as the edition value for backward compatibility reasons.

Each project can opt in to an edition other than the default 2015 edition. Editions can contain incompatible changes, such as including a new keyword that conflicts with identifiers in code. However, unless you opt in to those changes, your code will continue to compile even as you upgrade the Rust compiler version you use.

All Rust compiler versions support any edition that existed prior to that compiler’s release, and they can link crates of any supported editions together. Edition changes only affect the way the compiler initially parses code. Therefore, if you’re using Rust 2015 and one of your dependencies uses Rust 2018, your project will compile and be able to use that dependency. The opposite situation, where your project uses Rust 2018 and a dependency uses Rust 2015, works as well.

To be clear: most features will be available on all editions. Developers using any Rust edition will continue to see improvements as new stable releases are made. However, in some cases, mainly when new keywords are added, some new features might only be available in later editions. You will need to switch editions if you want to take advantage of such features.

For more details, the Edition Guide at https://doc.rust-lang.org/stable/edition-guide/ is a complete book about editions that enumerates the differences between editions and explains how to automatically upgrade your code to a new edition via cargo fix.