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.

Rust installations include rustfmt by default, so you should already have the programs rustfmt and cargo-fmt on your system. These two commands are analogous to rustc and cargo in that rustfmt allows finer-grained control and cargo-fmt understands conventions of a project that uses 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 compiler warnings that have a clear way to correct the problem that’s likely what you want. You’ve probably seen compiler warnings before. For example, consider this code:

src/main.rs

fn do\_something() {}

fn main() {

let mut x = 42;

println!("{x}");

}

Here, we’re defining the variable x as mutable, but we never actually mutate it. 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 0..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 remove the mut keyword. 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:

src/main.rs

fn do\_something() {}

fn main() {

let x = 42;

println!("{x}");

}

The variable x is now immutable, 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 so you can catch common mistakes and improve your Rust code. Clippy is included with standard Rust installations.

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:

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

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

|

2 | let x = 3.1415;

| ^^^^^^

|

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

= help: consider using the constant directly

= help: for further information visit https://rust-lang.github.io/rust-

clippy/master/index.html#approx\_constant

This error lets you know that Rust already has a more precise PI constant defined, 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:

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](https://github.com/rust-lang/rust-clippy).

IDE Integration Using rust-analyzer

To help with IDE integration, the Rust community recommends using rust-analyzer. This tool is a set of compiler-centric utilities that speak Language Server Protocol, which is a specification for IDEs and programming languages to communicate with each other. Different clients can use rust-analyzer, such as the Rust analyzer plug-in for Visual Studio Code at <https://marketplace.visualstudio.com/items?itemName=rust-lang.rust-analyzer>.

Visit the rust-analyzer project’s home page at <https://rust-analyzer.github.io> for installation instructions, then install the language server support in your particular IDE. Your IDE will gain capabilities such as autocompletion, jump to definition, and inline errors.