build passing | build passing | crates.io v1.4.21 | travis example passing

rustfmt

A tool for formatting Rust code according to style guidelines.

If you'd like to help out (and you should, it's a fun project!), see Contributing.md and our Code of Conduct.

You can use rustfmt in Travis CI builds. We provide a minimal Travis CI configuration (see here) and verify its status using another repository. The status of that repository's build is reported by the "travis example" badge above.

Quick start

You can run `rustfmt` with Rust 1.24 and above.

On the Stable toolchain

To install:

```
rustup component add rustfmt
```

To run on a cargo project in the current working directory:

```
cargo fmt
```

On the Nightly toolchain

For the latest and greatest `rustfmt`, nightly is required.

To install:

```
rustup component add rustfmt --toolchain nightly
```

To run on a cargo project in the current working directory:

```
cargo +nightly fmt
```

Limitations

Rustfmt tries to work on as much Rust code as possible. Sometimes, the code doesn't even need to compile! In general, we are looking to limit areas of instability; in particular, post-1.0, the formatting of most code should not change as Rustfmt improves. However, there are some things that Rustfmt can't do or can't do well (and thus where formatting might change significantly, even post-1.0). We would like to reduce the list of limitations over time.

The following list enumerates areas where Rustfmt does not work or where the stability guarantees do not apply (we don't make a distinction between the two because in the future Rustfmt might work on code where it currently does not):

- a program where any part of the program does not parse (parsing is an early stage of compilation and in Rust includes macro expansion).

- Macro declarations and uses (current status: some macro declarations and uses are formatted).
- Comments, including any AST node with a comment 'inside' (Rustfmt does not currently attempt to format comments, it does format code with comments inside, but that formatting may change in the future).
- Rust code in code blocks in comments.
- Any fragment of a program (i.e., stability guarantees only apply to whole programs, even where fragments of a program can be formatted today).
- Code containing non-ascii unicode characters (we believe Rustfmt mostly works here, but do not have the test coverage or experience to be 100% sure).
- Bugs in Rustfmt (like any software, Rustfmt has bugs, we do not consider bug fixes to break our stability guarantees).

Installation

```
rustup component add rustfmt
```

Installing from source

To install from source (nightly required), first checkout to the tag or branch you want to install, then issue

```
cargo install --path .
```

This will install `rustfmt` in your `~/.cargo/bin`. Make sure to add `~/.cargo/bin` directory to your PATH variable.

Running

You can run Rustfmt by just typing `rustfmt filename` if you used `cargo install`. This runs rustfmt on the given file, if the file includes out of line modules, then we reformat those too. So to run on a whole module or crate, you just need to run on the root file (usually mod.rs or lib.rs). Rustfmt can also read data from stdin. Alternatively, you can use `cargo fmt` to format all binary and library targets of your crate.

You can run `rustfmt --help` for information about available arguments. The easiest way to run rustfmt against a project is with `cargo fmt`. `cargo fmt` works on both single-crate projects and cargo workspaces. Please see `cargo fmt --help` for usage information.

You can specify the path to your own `rustfmt` binary for cargo to use by setting the `RUSTFMT` environment variable. This was added in v1.4.22, so you must have this version or newer to leverage this feature (`cargo fmt --version`)

Running `rustfmt` directly

To format individual files or arbitrary codes from stdin, the `rustfmt` binary should be used. Some examples follow:

- `rustfmt lib.rs main.rs` will format "lib.rs" and "main.rs" in place
- `rustfmt` will read a code from stdin and write formatting to stdout
  - `echo "fn main() {}" | rustfmt` would emit "fn main() {}".

For more information, including arguments and emit options, see `rustfmt --help`.

Verifying code is formatted

When running with `--check`, Rustfmt will exit with `0` if Rustfmt would not make any formatting changes to the input, and `1` if Rustfmt would make changes. In other modes, Rustfmt will exit with `1` if there was some error during formatting (for example a parsing or internal error) and `0` if formatting completed without error (whether or not changes were made).

Running Rustfmt from your editor

- Vim
- Emacs
- Sublime Text 3
- Atom
- Visual Studio Code using vscode-rust, vsc-rustfmt or rls_vscode through RLS.
- IntelliJ or CLion

Checking style on a CI server

To keep your code base consistently formatted, it can be helpful to fail the CI build when a pull request contains unformatted code. Using `--check` instructs rustfmt to exit with an error code if the input is not formatted correctly. It will also print any found differences. (Older versions of Rustfmt don't support `--check`, use `--write-mode diff`).

A minimal Travis setup could look like this (requires Rust 1.31.0 or greater):

```
language: rust
before_script:
- rustup component add rustfmt
script:
- cargo build
- cargo test
- cargo fmt --all -- --check
```

See this blog post for more info.

How to build and test

`cargo build` to build.

`cargo test` to run all tests.

To run rustfmt after this, use `cargo run --bin rustfmt -- filename`. See the notes above on running rustfmt.

Configuring Rustfmt

Rustfmt is designed to be very configurable. You can create a TOML file called `rustfmt.toml` or `.rustfmt.toml`, place it in the project or any other parent directory and it will apply the options in that file. See `rustfmt --help=config` for the options which are available, or if you prefer to see visual style previews, GitHub page.

By default, Rustfmt uses a style which conforms to the Rust style guide that has been formalized through the style RFC process.

Configuration options are either stable or unstable. Stable options can always be used, while unstable ones are only available on a nightly toolchain, and opt-in. See GitHub page for details.

Rust's Editions

Rustfmt is able to pick up the edition used by reading the `Cargo.toml` file if executed through the Cargo's formatting tool `cargo fmt`. Otherwise, the edition needs to be specified in `rustfmt.toml`, e.g., with `edition = "2018"`.

Tips

- For things you do not want rustfmt to mangle, use `#[rustfmt::skip]`
- To prevent rustfmt from formatting a macro or an attribute, use `#[rustfmt::skip::macros(target_macro_name)]` or `#[rustfmt::skip::attributes(target_attribute_name)]`

  Example:

  ```
  #![rustfmt::skip::attributes(custom_attribute)]

  #[custom_attribute(formatting , here , should , be , Skipped)]
  #[rustfmt::skip::macros(html)]
  fn main() {
      let macro_result1 = html! { <div>
  Hello</div>
      }.to_string();
  ```

- When you run rustfmt, place a file named `rustfmt.toml` or `.rustfmt.toml` in target file directory or its parents to override the default settings of rustfmt. You can generate a file containing the default configuration with `rustfmt --print-config default rustfmt.toml` and customize as needed.
- After successful compilation, a `rustfmt` executable can be found in the target directory.
- If you're having issues compiling Rustfmt (or compile errors when trying to install), make sure you have the most recent version of Rust installed.
- You can change the way rustfmt emits the changes with the --emit flag:

  Example:

  ```
  cargo fmt -- --emit files
  ```

  Options:

Flag	Description	Nightly Only
files	overwrites output to files	No
stdout	writes output to stdout	No

coverage	displays how much of the input file was processed	Yes
checkstyle	emits in a checkstyle format	Yes
json	emits diffs in a json format	Yes

License