Contributing

There are many ways to contribute to Rustfmt. This document lays out what they are and has information on how to get started. If you have any questions about contributing or need help with anything, please ask in the WG-Rustfmt channel on Discord. Feel free to also ask questions on issues, or file new issues specifically to get help.

All contributors are expected to follow our Code of Conduct.

Test and file issues

It would be really useful to have people use rustfmt on their projects and file issues where it does something you don't expect.

Create test cases

Having a strong test suite for a tool like this is essential. It is very easy to create regressions. Any tests you can add are very much appreciated.

The tests can be run with `cargo test`. This does a number of things: * runs the unit tests for a number of internal functions; * makes sure that rustfmt run on every file in `./tests/source/` is equal to its associated file in `./tests/target/`; * runs idempotence tests on the files in `./tests/target/`. These files should not be changed by rustfmt; * checks that rustfmt's code is not changed by running on itself. This ensures that the project bootstraps.

Creating a test is as easy as creating a new file in `./tests/source/` and an equally named one in `./tests/target/`. If it is only required that rustfmt leaves a piece of code unformatted, it may suffice to only create a target file.

Whenever there's a discrepancy between the expected output when running tests, a colourised diff will be printed so that the offending line(s) can quickly be identified.

Without explicit settings, the tests will be run using rustfmt's default configuration. It is possible to run a test using non-default settings in several ways. Firstly, you can include configuration parameters in comments at the top of the file. For example: to use 3 spaces per tab, start your test with `// rustfmt-tab_spaces: 3`. Just remember that the comment is part of the input, so include in both the source and target files! It is also possible to explicitly specify the name of the expected output file in the target directory. Use `// rustfmt-target: filename.rs` for this. You can also specify a custom configuration by using the `rustfmt-config` directive. Rustfmt will then use that toml file located in `./tests/config/` for its configuration. Including `// rustfmt-config: small_tabs.toml` will run your test with the configuration file found at `./tests/config/small_tabs.toml`. The final option is used when the test source file contains no configuration parameter comments. In this case, the test harness looks for a configuration file with the same filename

as the test file in the `./tests/config/` directory, so a test source file named `test-indent.rs` would need a configuration file named `test-indent.toml` in that directory. As an example, the `issue-1111.rs` test file is configured by the file `./tests/config/issue-1111.toml`.

Debugging

Some `rewrite_*` methods use the `debug!` macro for printing useful information. These messages can be printed by using the environment variable `RUSTFMT_LOG=rustfmt=DEBUG`. These traces can be helpful in understanding which part of the code was used and get a better grasp on the execution flow.

Hack!

Here are some good starting issues.

If you've found areas which need polish and don't have issues, please submit a PR, don't feel there needs to be an issue.

Guidelines

Rustfmt bootstraps, that is part of its test suite is running itself on its source code. So, basically, the only style guideline is that you must pass the tests. That ensures that the Rustfmt source code adheres to our own conventions.

Talking of tests, if you add a new feature or fix a bug, please also add a test. It's really easy, see above for details. Please run `cargo test` before submitting a PR to ensure your patch passes all tests, it's pretty quick.

Rustfmt is post-1.0 and within major version releases we strive for backwards compatibility (at least when using the default options). That means any code which changes Rustfmt's output must be guarded by either an option or a version check. The latter is implemented as an option called `option`. See the section on configuration below.

Please try to avoid leaving `TODO`s in the code. There are a few around, but I wish there weren't. You can leave `FIXME`s, preferably with an issue number.

Version-gate formatting changes

A change that introduces a different code-formatting should be gated on the `version` configuration. This is to ensure the formatting of the current major release is preserved, while allowing fixes to be implemented for the next release.

This is done by conditionally guarding the change like so:

```
if config.version() == Version::One { // if the current major release is 1.x
    // current formatting
} else {
```

```
    // new formatting
}
```

This allows the user to apply the next formatting explicitly via the configuration, while being stable by default.

When the next major release is done, the code block of the previous formatting can be deleted, e.g., the first block in the example above when going from `1.x` to `2.x`.

Note: Only formatting changes with default options need to be gated.

A quick tour of Rustfmt

Rustfmt is basically a pretty printer - that is, its mode of operation is to take an AST (abstract syntax tree) and print it in a nice way (including staying under the maximum permitted width for a line). In order to get that AST, we first have to parse the source text, we use the Rust compiler's parser to do that (see src/lib.rs). We shy away from doing anything too fancy, such as algebraic approaches to pretty printing, instead relying on an heuristic approach, 'manually' crafting a string for each AST node. This results in quite a lot of code, but it is relatively simple.

The AST is a tree view of source code. It carries all the semantic information about the code, but not all of the syntax. In particular, we lose white space and comments (although doc comments are preserved). Rustfmt uses a view of the AST before macros are expanded, so there are still macro uses in the code. The arguments to macros are not an AST, but raw tokens - this makes them harder to format.

There are different nodes for every kind of item and expression in Rust. For more details see the source code in the compiler - ast.rs - and/or the docs.

Many nodes in the AST (but not all, annoyingly) have a `Span`. A `Span` is a range in the source code, it can easily be converted to a snippet of source text. When the AST does not contain enough information for us, we rely heavily on `Span`s. For example, we can look between spans to try and find comments, or parse a snippet to see how the user wrote their source code.

The downside of using the AST is that we miss some information - primarily white space and comments. White space is sometimes significant, although mostly we want to ignore it and make our own. We strive to reproduce all comments, but this is sometimes difficult. The crufty corners of Rustfmt are where we hack around the absence of comments in the AST and try to recreate them as best we can.

Our primary tool here is to look between spans for text we've missed. For example, in a function call `foo(a, b)`, we have spans for `a` and `b`, in this case,

there is only a comma and a single space between the end of `a` and the start of `b`, so there is nothing much to do. But if we look at `foo(a /* a comment */, b)`, then between `a` and `b` we find the comment.

At a higher level, Rustfmt has machinery so that we account for text between 'top level' items. Then we can reproduce that text pretty much verbatim. We only count spans we actually reformat, so if we can't format a span it is not missed completely but is reproduced in the output without being formatted. This is mostly handled in src/missed_spans.rs. See also `FmtVisitor::last_pos` in src/visitor.rs.

Some important elements At the highest level, Rustfmt uses a `Visitor` implementation called `FmtVisitor` to walk the AST. This is in src/visitor.rs. This is really just used to walk items, rather than the bodies of functions. We also cover macros and attributes here. Most methods of the visitor call out to `Rewrite` implementations that then walk their own children.

The `Rewrite` trait is defined in src/rewrite.rs. It is implemented for many things that can be rewritten, mostly AST nodes. It has a single function, `rewrite`, which is called to rewrite `self` into an `Option<String>`. The arguments are `width` which is the horizontal space we write into and `offset` which is how much we are currently indented from the lhs of the page. We also take a context which contains information used for parsing, the current block indent, and a configuration (see below).

Rewrite and Indent To understand the indents, consider

```
impl Foo {
    fn foo(...) {
        bar(argument_one,
            baz());
    }
}
```

When formatting the `bar` call we will format the arguments in order, after the first one we know we are working on multiple lines (imagine it is longer than written). So, when we come to the second argument, the indent we pass to `rewrite` is 12, which puts us under the first argument. The current block indent (stored in the context) is 8. The former is used for visual indenting (when objects are vertically aligned with some marker), the latter is used for block indenting (when objects are tabbed in from the lhs). The width available for `baz()` will be the maximum width, minus the space used for indenting, minus the space used for the `);`. (Note that actual argument formatting does not quite work like this, but it's close enough).

The `rewrite` function returns an `Option` - either we successfully rewrite and return the rewritten string for the caller to use, or we fail to rewrite and return `None`. This could be because Rustfmt encounters something it doesn't know how

to reformat, but more often it is because Rustfmt can't fit the item into the required width. How to handle this is up to the caller. Often the caller just gives up, ultimately relying on the missed spans system to paste in the un-formatted source. A better solution (although not performed in many places) is for the caller to shuffle around some of its other items to make more width, then call the function again with more space.

Since it is common for callers to bail out when a callee fails, we often use a `?` operator to make this pattern more succinct.

One way we might find out that we don't have enough space is when computing how much space we have. Something like `available_space = budget - overhead`. Since widths are unsized integers, this would cause underflow. Therefore we use checked subtraction: `available_space = budget.checked_sub(overhead)?`. `checked_sub` returns an `Option`, and if we would underflow `?` returns `None`, otherwise, we proceed with the computed space.

Rewrite of list-like expressions Much of the syntax in Rust is lists: lists of arguments, lists of fields, lists of array elements, etc. We have some generic code to handle lists, including how to space them in horizontal and vertical space, indentation, comments between items, trailing separators, etc. However, since there are so many options, the code is a bit complex. Look in src/lists.rs. `write_list` is the key function, and `ListFormatting` the key structure for configuration. You'll need to make a `ListItems` for input, this is usually done using `itemize_list`.

Configuration Rustfmt strives to be highly configurable. Often the first part of a patch is creating a configuration option for the feature you are implementing. All handling of configuration options is done in src/config/mod.rs. Look for the `create_config!` macro at the end of the file for all the options. The rest of the file defines a bunch of enums used for options, and the machinery to produce the config struct and parse a config file, etc. Checking an option is done by accessing the correct field on the config struct, e.g., `config.max_width()`. Most functions have a `Config`, or one can be accessed via a visitor or context of some kind.