Contributing to Clippy

Hello fellow Rustacean! Great to see your interest in compiler internals and lints!

First: if you're unsure or afraid of *anything*, just ask or submit the issue or pull request anyway. You won't be yelled at for giving it your best effort. The worst that can happen is that you'll be politely asked to change something. We appreciate any sort of contributions, and don't want a wall of rules to get in the way of that.

Clippy welcomes contributions from everyone. There are many ways to contribute to Clippy and the following document explains how you can contribute and how to get started. If you have any questions about contributing or need help with anything, feel free to ask questions on issues or visit the `#clippy` on Zulip.

All contributors are expected to follow the Rust Code of Conduct.

Getting started

Note: If this is your first time contributing to Clippy, you should first read the Basics docs.

High level approach

1. Find something to fix/improve
2. Change code (likely some file in `clippy_lints/src/`)
3. Follow the instructions in the Basics docs to get set up
4. Run `cargo test` in the root directory and wiggle code until it passes
5. Open a PR (also can be done after 2. if you run into problems)

Finding something to fix/improve

All issues on Clippy are mentored, if you want help simply ask @Manishearth, @flip1995, @phansch or @llogiq directly by mentioning them in the issue or over on Zulip. This list may be out of date. All currently active mentors can be found here

Some issues are easier than others. The `good-first-issue` label can be used to find the easy issues. You can use `@rustbot claim` to assign the issue to yourself.

There are also some abandoned PRs, marked with `S-inactive-closed`. Pretty often these PRs are nearly completed and just need some extra steps (formatting, addressing review comments, ...) to be merged. If you want to complete such a PR, please leave a comment in the PR and open a new one based on it.

Issues marked `T-AST` involve simple matching of the syntax tree structure, and are generally easier than `T-middle` issues, which involve types and resolved paths.

`T-AST` issues will generally need you to match against a predefined syntax structure. To figure out how this syntax structure is encoded in the AST, it is recommended to run `rustc -Z ast-json` on an example of the structure and compare with the nodes in the AST docs. Usually the lint will end up to be a nested series of matches and ifs, like so. But we can make it nest-less by using if chain macro, like this.

`E-medium` issues are generally pretty easy too, though it's recommended you work on an `good-first-issue` first. Sometimes they are only somewhat involved code wise, but not difficult per-se. Note that `E-medium` issues may require some knowledge of Clippy internals or some debugging to find the actual problem behind the issue.

`T-middle` issues can be more involved and require verifying types. The `ty` module contains a lot of methods that are useful, though one of the most useful would be `expr_ty` (gives the type of an AST expression). `match_def_path()` in Clippy's `utils` module can also be useful.

Writing code

Have a look at the docs for writing lints for more details.

If you want to add a new lint or change existing ones apart from bugfixing, it's also a good idea to give the stability guarantees and lint categories sections of the Clippy 1.0 RFC a quick read.

Getting code-completion for rustc internals to work

IntelliJ Rust

Unfortunately, `IntelliJ Rust` does not (yet?) understand how Clippy uses compiler-internals using `extern crate` and it also needs to be able to read the source files of the rustc-compiler which are not available via a `rustup` component at the time of writing. To work around this, you need to have a copy of the rustc-repo available which can be obtained via `git clone https://github.com/rust-lang/rust/`. Then you can run a `cargo dev` command to automatically make Clippy use the rustc-repo via path-dependencies which `IntelliJ Rust` will be able to understand. Run `cargo dev setup intellij --repo-path <repo-path>` where `<repo-path>` is a path to the rustc repo you just cloned. The command will add path-dependencies pointing towards rustc-crates inside the rustc repo to Clippy's `Cargo.toml` s and should allow `IntelliJ Rust` to understand most of the types that Clippy uses. Just make sure to remove the dependencies again before finally making a pull request!

Rust Analyzer

As of #6869, `rust-analyzer` can understand that Clippy uses compiler-internals using `extern crate` when `package.metadata.rust-analyzer.rustc_private` is set to `true` in Clippy's `Cargo.toml.` You will require a `nightly` toolchain with the `rustc-dev` component installed. Make sure that in the `rust-analyzer` configuration, you set

```
{ "rust-analyzer.rustcSource": "discover" }
```

and

```
{ "rust-analyzer.updates.channel": "nightly" }
```

You should be able to see information on things like `Expr` or `EarlyContext` now if you hover them, also a lot more type hints. This will work with `rust-analyzer 2021-03-15` shipped in nightly `1.52.0-nightly (107896c32 2021-03-15)` or later.

How Clippy works

`clippy_lints/src/lib.rs` imports all the different lint modules and registers in the `LintStore`. For example, the `else_if_without_else` lint is registered like this:

```
// ./clippy_lints/src/lib.rs

// ...
pub mod else_if_without_else;
// ...

pub fn register_plugins(store: &mut rustc_lint::LintStore, sess: &Session, conf:
&Conf) {
    // ...
    store.register_early_pass(|| box else_if_without_else::ElseIfWithoutElse);
    // ...

    store.register_group(true, "clippy::restriction", Some("clippy_restriction"),
vec![
        // ...
        LintId::of(&else_if_without_else::ELSE_IF_WITHOUT_ELSE),
        // ...
    ]);
}
```

The `rustc_lint::LintStore` provides two methods to register lints: register_early_pass and register_late_pass. Both take an object that implements an `EarlyLintPass` or `LateLintPass` respectively. This is done in every single lint. It's worth noting that the majority of `clippy_lints/src/lib.rs` is autogenerated by `cargo dev update_lints`. When you are writing your own lint, you can use that script to save you some time.

```
// ./clippy_lints/src/else_if_without_else.rs

use rustc_lint::{EarlyLintPass, EarlyContext};

// ...

pub struct ElseIfWithoutElse;

// ...

impl EarlyLintPass for ElseIfWithoutElse {
```

```
    // ... the functions needed, to make the lint work
}
```

The difference between `EarlyLintPass` and `LateLintPass` is that the methods of the `EarlyLintPass` trait only provide AST information. The methods of the `LateLintPass` trait are executed after type checking and contain type information via the `LateContext` parameter.

That's why the `else_if_without_else` example uses the `register_early_pass` function. Because the actual lint logic does not depend on any type information.

Syncing changes between Clippy and `rust-lang/rust`

Clippy currently gets built with a pinned nightly version.

In the `rust-lang/rust` repository, where rustc resides, there's a copy of Clippy that compiler hackers modify from time to time to adapt to changes in the unstable API of the compiler.

We need to sync these changes back to this repository periodically, and the changes made to this repository in the meantime also need to be synced to the `rust-lang/rust` repository.

To avoid flooding the `rust-lang/rust` PR queue, this two-way sync process is done in a bi-weekly basis if there's no urgent changes. This is done starting on the day of the Rust stable release and then every other week. That way we guarantee that we keep this repo up to date with the latest compiler API, and every feature in Clippy is available for 2 weeks in nightly, before it can get to beta. For reference, the first sync following this cadence was performed the 2020-08-27.

This process is described in detail in the following sections. For general information about `subtree` s in the Rust repository see Rust's `CONTRIBUTING.md` .

Patching git-subtree to work with big repos

Currently, there's a bug in `git-subtree` that prevents it from working properly with the `rust-lang/rust` repo. There's an open PR to fix that, but it's stale. Before continuing with the following steps, we need to manually apply that fix to our local copy of `git-subtree` .

You can get the patched version of `git-subtree` from here. Put this file under `/usr/lib/git-core` (taking a backup of the previous file) and make sure it has the proper permissions:

```
sudo cp --backup /path/to/patched/git-subtree.sh /usr/lib/git-core/git-subtree
sudo chmod --reference=/usr/lib/git-core/git-subtree~ /usr/lib/git-core/git-subtree
sudo chown --reference=/usr/lib/git-core/git-subtree~ /usr/lib/git-core/git-subtree
```

Note: The first time running `git subtree push` a cache has to be built. This involves going through the complete Clippy history once. For this you have to increase the stack limit though, which you can do with `ulimit -s 60000` . Make sure to run the `ulimit` command from the same session you call git subtree.

Note: If you are a Debian user, `dash` is the shell used by default for scripts instead of `sh` . This shell has a hardcoded recursion limit set to 1000. In order to make this process work, you need to force the script to run `bash` instead. You can do this by editing the first line of the `git-subtree` script and changing `sh` to `bash` .

Performing the sync from `rust-lang/rust` to Clippy

Here is a TL;DR version of the sync process (all of the following commands have to be run inside the `rust` directory):

1. Clone the `rust-lang/rust` repository or make sure it is up to date.
2. Checkout the commit from the latest available nightly. You can get it using `rustup check`.
3. Sync the changes to the rust-copy of Clippy to your Clippy fork:

   ```
   # Make sure to change `your-github-name` to your github name in the following command. Also be
   # sure to either use a net-new branch, e.g. `sync-from-rust`, or delete the branch beforehand
   # because changes cannot be fast forwarded
   git subtree push -P src/tools/clippy git@github.com:your-github-name/rust-clippy sync-from-rust
   ```

   Note: This will directly push to the remote repository. You can also push to your local copy by replacing the remote address with `/path/to/rust-clippy` directory.

   Note: Most of the time you have to create a merge commit in the `rust-clippy` repo (this has to be done in the Clippy repo, not in the rust-copy of Clippy):

   ```
   git fetch origin && git fetch upstream
   git checkout sync-from-rust
   git merge upstream/master
   ```

4. Open a PR to `rust-lang/rust-clippy` and wait for it to get merged (to accelerate the process ping the `@rust-lang/clippy` team in your PR and/or ~~annoy~~ ask them in the Zulip stream.)

Performing the sync from Clippy to `rust-lang/rust`

All of the following commands have to be run inside the `rust` directory.

1. Make sure Clippy itself is up-to-date by following the steps outlined in the previous section if necessary.
2. Sync the `rust-lang/rust-clippy` master to the rust-copy of Clippy:

   ```
   git checkout -b sync-from-clippy
   git subtree pull -P src/tools/clippy https://github.com/rust-lang/rust-clippy master
   ```

3. Open a PR to `rust-lang/rust`

Defining remotes

You may want to define remotes, so you don't have to type out the remote addresses on every sync. You can do this with the following commands (these commands still have to be run inside the `rust` directory):

```
# Set clippy-upstream remote for pulls
$ git remote add clippy-upstream https://github.com/rust-lang/rust-clippy
```

```
# Make sure to not push to the upstream repo
$ git remote set-url --push clippy-upstream DISABLED
# Set clippy-origin remote to your fork for pushes
$ git remote add clippy-origin git@github.com:your-github-name/rust-clippy
# Set a local remote
$ git remote add clippy-local /path/to/rust-clippy
```

You can then sync with the remote names from above, e.g.:

```
$ git subtree push -P src/tools/clippy clippy-local sync-from-rust
```

Issue and PR triage

Clippy is following the Rust triage procedure for issues and pull requests.

However, we are a smaller project with all contributors being volunteers currently. Between writing new lints, fixing issues, reviewing pull requests and responding to issues there may not always be enough time to stay on top of it all.

Our highest priority is fixing crashes and bugs, for example an ICE in a popular crate that many other crates depend on. We don't want Clippy to crash on your code and we want it to be as reliable as the suggestions from Rust compiler errors.

We have prioritization labels and a sync-blocker label, which are described below.

- P-low: Requires attention (fix/response/evaluation) by a team member but isn't urgent.
- P-medium: Should be addressed by a team member until the next sync.
- P-high: Should be immediately addressed and will require an out-of-cycle sync or a backport.
- L-sync-blocker: An issue that "blocks" a sync. Or rather: before the sync this should be addressed, e.g. by removing a lint again, so it doesn't hit beta/stable.

Bors and Homu

We use a bot powered by Homu to help automate testing and landing of pull requests in Clippy. The bot's username is @bors.

You can find the Clippy bors queue here.

If you have @bors permissions, you can find an overview of the available commands here.

Contributions

Contributions to Clippy should be made in the form of GitHub pull requests. Each pull request will be reviewed by a core contributor (someone with permission to land patches) and either landed in the main tree or given feedback for changes that would be required.