Adding a new lint

You are probably here because you want to add a new lint to Clippy. If this is the first time you're contributing to Clippy, this document guides you through creating an example lint from scratch.

To get started, we will create a lint that detects functions called `foo`, because that's clearly a non-descriptive name.

Setup

See the Basics documentation.

Getting Started

There is a bit of boilerplate code that needs to be set up when creating a new lint. Fortunately, you can use the clippy dev tools to handle this for you. We are naming our new lint `foo_functions` (lints are generally written in snake case), and we don't need type information so it will have an early pass type (more on this later on). If you're not sure if the name you chose fits the lint, take a look at our lint naming guidelines. To get started on this lint you can run `cargo dev new_lint --name=foo_functions --pass=early --category=pedantic` (category will default to nursery if not provided). This command will create two files: `tests/ui/foo_functions.rs` and `clippy_lints/src/foo_functions.rs`, as well as registering the lint. For

cargo lints, two project hierarchies (fail/pass) will be created by default under `tests/ui-cargo`.

Next, we'll open up these files and add our lint!

Testing

Let's write some tests first that we can execute while we iterate on our lint.

Clippy uses UI tests for testing. UI tests check that the output of Clippy is exactly as expected. Each test is just a plain Rust file that contains the code we want to check. The output of Clippy is compared against a `.stderr` file. Note that you don't have to create this file yourself, we'll get to generating the `.stderr` files further down.

We start by opening the test file created at `tests/ui/foo_functions.rs`.

Update the file with some examples to get started:

```
#![warn(clippy::foo_functions)]

// Impl methods
struct A;
impl A {
    pub fn fo(&self) {}
    pub fn foo(&self) {}
    pub fn food(&self) {}
}

// Default trait methods
trait B {
    fn fo(&self) {}
    fn foo(&self) {}
    fn food(&self) {}
}

// Plain functions
fn fo() {}
fn foo() {}
fn food() {}

fn main() {
    // We also don't want to lint method calls
    foo();
    let a = A;
    a.foo();
}
```

Now we can run the test with `TESTNAME=foo_functions cargo uitest`, currently this test is meaningless though.

While we are working on implementing our lint, we can keep running the UI test. That allows us to check if the output is turning into what we want.

Once we are satisfied with the output, we need to run `cargo dev bless` to update the `.stderr` file for our lint. Please note that, we should run `TESTNAME=foo_functions cargo uitest` every time before running `cargo dev bless`. Running `TESTNAME=foo_functions cargo uitest` should pass then. When we commit our lint, we need to commit the generated `.stderr` files, too. In general, you should only commit files changed by `cargo dev bless` for the specific lint you are creating/editing. Note that if the generated files are empty, they should be removed.

Note that you can run multiple test files by specifying a comma separated list: `TESTNAME=foo_functions,test2,test3`.

Cargo lints

For cargo lints, the process of testing differs in that we are interested in the `Cargo.toml` manifest file. We also need a minimal crate associated with that manifest.

If our new lint is named e.g. `foo_categories`, after running `cargo dev new_lint` we will find by default two new crates, each with its manifest file:

- `tests/ui-cargo/foo_categories/fail/Cargo.toml`: this file should cause the new lint to raise an error.
- `tests/ui-cargo/foo_categories/pass/Cargo.toml`: this file should not trigger the lint.

If you need more cases, you can copy one of those crates (under `foo_categories`) and rename it.

The process of generating the `.stderr` file is the same, and prepending the `TESTNAME` variable to `cargo uitest` works too.

Rustfix tests

If the lint you are working on is making use of structured suggestions, the test file should include a `// run-rustfix` comment at the top. This will additionally run rustfix for that test. Rustfix will apply the suggestions from the lint to the code of the test file and compare that to the contents of a `.fixed` file.

Use `cargo dev bless` to automatically generate the `.fixed` file after running the tests.

Edition 2018 tests

Some features require the 2018 edition to work (e.g. `async_await`), but compile-test tests run on the 2015 edition by default. To change this behavior add `// edition:2018` at the top of the test file (note that it's space-sensitive).

Testing manually

Manually testing against an example file can be useful if you have added some `println!`s and the test suite output becomes unreadable. To try Clippy with your local modifications, run

```
cargo dev lint input.rs
```

from the working copy root. With tests in place, let's have a look at implementing our lint now.

Lint declaration

Let's start by opening the new file created in the `clippy_lints` crate at `clippy_lints/src/foo_functions.rs`. That's the crate where all the lint code is. This file has already imported some initial things we will need:

```
use rustc_lint::{EarlyLintPass, EarlyContext};
use rustc_session::{declare_lint_pass, declare_tool_lint};
use rustc_ast::ast::*;
```

The next step is to update the lint declaration. Lints are declared using the `declare_clippy_lint!` macro, and we just need to update the auto-generated lint declaration to have a real description, something like this:

```
declare_clippy_lint! {
    /// ### What it does
    ///
    /// ### Why is this bad?
    ///
    /// ### Example
    /// ```rust
    /// // example code
    /// ```
    #[clippy::version = "1.29.0"]
    pub FOO_FUNCTIONS,
    pedantic,
    "function named `foo`, which is not a descriptive name"
}
```

- The section of lines prefixed with `///` constitutes the lint documentation section. This is the default documentation style and will be displayed like

this. To render and open this documentation locally in a browser, run `cargo dev serve`.
- The `#[clippy::version]` attribute will be rendered as part of the lint documentation. The value should be set to the current Rust version that the lint is developed in, it can be retrieved by running `rustc -vV` in the rust-clippy directory. The version is listed under *release*. (Use the version without the `-nightly`) suffix.
- `FOO_FUNCTIONS` is the name of our lint. Be sure to follow the lint naming guidelines here when naming your lint. In short, the name should state the thing that is being checked for and read well when used with `allow`/`warn`/`deny`.
- `pedantic` sets the lint level to `Allow`. The exact mapping can be found here
- The last part should be a text that explains what exactly is wrong with the code

The rest of this file contains an empty implementation for our lint pass, which in this case is `EarlyLintPass` and should look like this:

```
// clippy_lints/src/foo_functions.rs

// .. imports and lint declaration ..

declare_lint_pass!(FooFunctions => [FOO_FUNCTIONS]);

impl EarlyLintPass for FooFunctions {}
```

Lint registration

When using `cargo dev new_lint`, the lint is automatically registered and nothing more has to be done.

When declaring a new lint by hand and `cargo dev update_lints` is used, the lint pass may have to be registered manually in the `register_plugins` function in `clippy_lints/src/lib.rs`:

```
store.register_early_pass(|| Box::new(foo_functions::FooFunctions));
```

As one may expect, there is a corresponding `register_late_pass` method available as well. Without a call to one of `register_early_pass` or `register_late_pass`, the lint pass in question will not be run.

One reason that `cargo dev update_lints` does not automate this step is that multiple lints can use the same lint pass, so registering the lint pass may already be done when adding a new lint. Another reason that this step is not automated is that the order that the passes are registered determines the order the passes actually run, which in turn affects the order that any emitted lints are output in.

Lint passes

Writing a lint that only checks for the name of a function means that we only have to deal with the AST and don't have to deal with the type system at all. This is good, because it makes writing this particular lint less complicated.

We have to make this decision with every new Clippy lint. It boils down to using either `EarlyLintPass` or `LateLintPass`.

In short, the `LateLintPass` has access to type information while the `EarlyLintPass` doesn't. If you don't need access to type information, use the `EarlyLintPass`. The `EarlyLintPass` is also faster. However linting speed hasn't really been a concern with Clippy so far.

Since we don't need type information for checking the function name, we used `--pass=early` when running the new lint automation and all the imports were added accordingly.

Emitting a lint

With UI tests and the lint declaration in place, we can start working on the implementation of the lint logic.

Let's start by implementing the `EarlyLintPass` for our `FooFunctions`:

```
impl EarlyLintPass for FooFunctions {
    fn check_fn(&mut self, cx: &EarlyContext<'_>, fn_kind: FnKind<'_>, span: Span, _: NodeId
        // TODO: Emit lint here
    }
}
```

We implement the `check_fn` method from the `EarlyLintPass` trait. This gives us access to various information about the function that is currently being checked. More on that in the next section. Let's worry about the details later and emit our lint for *every* function definition first.

Depending on how complex we want our lint message to be, we can choose from a variety of lint emission functions. They can all be found in `clippy_utils/src/diagnostics.rs`.

`span_lint_and_help` seems most appropriate in this case. It allows us to provide an extra help message and we can't really suggest a better name automatically. This is how it looks:

```
impl EarlyLintPass for FooFunctions {
    fn check_fn(&mut self, cx: &EarlyContext<'_>, fn_kind: FnKind<'_>, span: Span, _: NodeId
        span_lint_and_help(
            cx,
            FOO_FUNCTIONS,
            span,
            "function named `foo`",
```

```
            None,
            "consider using a more meaningful name"
        );
    }
}
```

Running our UI test should now produce output that contains the lint message.

According to the rustc-dev-guide, the text should be matter of fact and avoid capitalization and periods, unless multiple sentences are needed. When code or an identifier must appear in a message or label, it should be surrounded with single grave accents ‘.

Adding the lint logic

Writing the logic for your lint will most likely be different from our example, so this section is kept rather short.

Using the `check_fn` method gives us access to `FnKind` that has the `FnKind::Fn` variant. It provides access to the name of the function/method via an `Ident`.

With that we can expand our `check_fn` method to:

```
impl EarlyLintPass for FooFunctions {
    fn check_fn(&mut self, cx: &EarlyContext<'_>, fn_kind: FnKind<'_>, span: Span, _: NodeId
        if is_foo_fn(fn_kind) {
            span_lint_and_help(
                cx,
                FOO_FUNCTIONS,
                span,
                "function named `foo`",
                None,
                "consider using a more meaningful name"
            );
        }
    }
}
```

We separate the lint conditional from the lint emissions because it makes the code a bit easier to read. In some cases this separation would also allow to write some unit tests (as opposed to only UI tests) for the separate function.

In our example, `is_foo_fn` looks like:

```
// use statements, impl EarlyLintPass, check_fn, ..

fn is_foo_fn(fn_kind: FnKind<'_>) -> bool {
    match fn_kind {
        FnKind::Fn(_, ident, ..) => {
            // check if `fn` name is `foo`
```

```
            ident.name.as_str() == "foo"
        }
        // ignore closures
        FnKind::Closure(..) => false
    }
}
```

Now we should also run the full test suite with `cargo test`. At this point running `cargo test` should produce the expected output. Remember to run `cargo dev bless` to update the `.stderr` file.

`cargo test` (as opposed to `cargo uitest`) will also ensure that our lint implementation is not violating any Clippy lints itself.

That should be it for the lint implementation. Running `cargo test` should now pass.

Specifying the lint's minimum supported Rust version (MSRV)

Sometimes a lint makes suggestions that require a certain version of Rust. For example, the `manual_strip` lint suggests using `str::strip_prefix` and `str::strip_suffix` which is only available after Rust 1.45. In such cases, you need to ensure that the MSRV configured for the project is >= the MSRV of the required Rust feature. If multiple features are required, just use the one with a lower MSRV.

First, add an MSRV alias for the required feature in `clippy_utils::msrvs`. This can be accessed later as `msrvs::STR_STRIP_PREFIX`, for example.

```
msrv_aliases! {
    ..
    1,45,0 { STR_STRIP_PREFIX }
}
```

In order to access the project-configured MSRV, you need to have an `msrv` field in the LintPass struct, and a constructor to initialize the field. The `msrv` value is passed to the constructor in `clippy_lints/lib.rs`.

```
pub struct ManualStrip {
    msrv: Option<RustcVersion>,
}

impl ManualStrip {
    #[must_use]
    pub fn new(msrv: Option<RustcVersion>) -> Self {
        Self { msrv }
    }
}
```

The project's MSRV can then be matched against the feature MSRV in the LintPass using the `meets_msrv` utility function.

```
if !meets_msrv(self.msrv.as_ref(), &msrvs::STR_STRIP_PREFIX) {
    return;
}
```

The project's MSRV can also be specified as an inner attribute, which overrides the value from `clippy.toml`. This can be accounted for using the `extract_msrv_attr!(LintContext)` macro and passing `LateContext`/`EarlyContext`.

```
impl<'tcx> LateLintPass<'tcx> for ManualStrip {
    fn check_expr(&mut self, cx: &LateContext<'tcx>, expr: &'tcx Expr<'_>) {
        ...
    }
    extract_msrv_attr!(LateContext);
}
```

Once the `msrv` is added to the lint, a relevant test case should be added to `tests/ui/min_rust_version_attr.rs` which verifies that the lint isn't emitted if the project's MSRV is lower.

As a last step, the lint should be added to the lint documentation. This is done in `clippy_lints/src/utils/conf.rs`:

```
define_Conf! {
    /// Lint: LIST, OF, LINTS, <THE_NEWLY_ADDED_LINT>. The minimum rust version that the pro
    (msrv: Option<String> = None),
    ...
}
```

Author lint

If you have trouble implementing your lint, there is also the internal `author` lint to generate Clippy code that detects the offending pattern. It does not work for all of the Rust syntax, but can give a good starting point.

The quickest way to use it, is the Rust playground: play.rust-lang.org. Put the code you want to lint into the editor and add the `#[clippy::author]` attribute above the item. Then run Clippy via `Tools -> Clippy` and you should see the generated code in the output below.

Here is an example on the playground.

If the command was executed successfully, you can copy the code over to where you are implementing your lint.

Documentation

The final thing before submitting our PR is to add some documentation to our lint declaration.

Please document your lint with a doc comment akin to the following:

```
declare_clippy_lint! {
    /// ### What it does
    /// Checks for ... (describe what the lint matches).
    ///
    /// ### Why is this bad?
    /// Supply the reason for linting the code.
    ///
    /// ### Example
    ///
    /// ```rust,ignore
    /// // Bad
    /// Insert a short example of code that triggers the lint
    ///
    /// // Good
    /// Insert a short example of improved code that doesn't trigger the lint
    /// ```
    #[clippy::version = "1.29.0"]
    pub FOO_FUNCTIONS,
    pedantic,
    "function named `foo`, which is not a descriptive name"
}
```

Once your lint is merged, this documentation will show up in the lint list.

Running rustfmt

Rustfmt is a tool for formatting Rust code according to style guidelines. Your code has to be formatted by `rustfmt` before a PR can be merged. Clippy uses nightly `rustfmt` in the CI.

It can be installed via `rustup`:

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

Use `cargo dev fmt` to format the whole codebase. Make sure that `rustfmt` is installed for the nightly toolchain.

Debugging

If you want to debug parts of your lint implementation, you can use the `dbg!` macro anywhere in your code. Running the tests should then include the debug output in the `stdout` part.

PR Checklist

Before submitting your PR make sure you followed all of the basic requirements:

- ☐ Followed lint naming conventions
- ☐ Added passing UI tests (including committed `.stderr` file)
- ☐ `cargo test` passes locally
- ☐ Executed `cargo dev update_lints`
- ☐ Added lint documentation
- ☐ Run `cargo dev fmt`

Adding configuration to a lint

Clippy supports the configuration of lints values using a `clippy.toml` file in the workspace directory. Adding a configuration to a lint can be useful for thresholds or to constrain some behavior that can be seen as a false positive for some users. Adding a configuration is done in the following steps:

1. Adding a new configuration entry to clippy_lints::utils::conf like this: `rust /// Lint: LINT_NAME. /// /// <The configuration field doc comment> (configuration_ident: Type = DefaultValue),` The doc comment is automatically added to the documentation of the listed lints. The default value will be formatted using the `Debug` implementation of the type.

2. Adding the configuration value to the lint impl struct:

    1. This first requires the definition of a lint impl struct. Lint impl structs are usually generated with the `declare_lint_pass!` macro. This struct needs to be defined manually to add some kind of metadata to it: “‘rust // Generated struct definition declare_lint_pass!(StructName => [LINT_NAME]);

        // New manual definition struct #[derive(Copy, Clone)] pub struct StructName {}

        impl_lint_pass!(StructName => [LINT_NAME]); “‘

    2. Next add the configuration value and a corresponding creation method like this: “‘rust #[derive(Copy, Clone)] pub struct StructName { configuration_ident: Type, }

        // ...

        impl StructName { pub fn new(configuration_ident: Type) -> Self { Self { configuration_ident, } } } “‘

3. Passing the configuration value to the lint impl struct:

    First find the struct construction in the clippy_lints lib file. The configuration value is now cloned or copied into a local value that is then

passed to the impl struct like this: “‘rust // Default generated registration: store.register_*_pass(|| box module::StructName);

// New registration with configuration value let configuration_ident = conf.configuration_ident.clone(); store.register_*_pass(move || box module::StructName::new(configuration_ident)); “‘

Congratulations the work is almost done. The configuration value can now be accessed in the linting code via `self.configuration_ident`.

4. Adding tests:
    1. The default configured value can be tested like any normal lint in `tests/ui`.
    2. The configuration itself will be tested separately in `tests/ui-toml`. Simply add a new subfolder with a fitting name. This folder contains a `clippy.toml` file with the configuration value and a rust file that should be linted by Clippy. The test can otherwise be written as usual.

Cheatsheet

Here are some pointers to things you are likely going to need for every lint:

- Clippy utils - Various helper functions. Maybe the function you need is already in here (`is_type_diagnostic_item`, `implements_trait`, `snippet`, etc)
- Clippy diagnostics
- The `if_chain` macro
- `from_expansion` and `in_external_macro`
- `Span`
- `Applicability`
- Common tools for writing lints helps with common operations
- The rustc-dev-guide explains a lot of internal compiler concepts
- The nightly rustc docs which has been linked to throughout this guide

For `EarlyLintPass` lints:

- `EarlyLintPass`
- `rustc_ast::ast`

For `LateLintPass` lints:

- `LateLintPass`
- `Ty::TyKind`

While most of Clippy's lint utils are documented, most of rustc's internals lack documentation currently. This is unfortunate, but in most cases you can probably get away with copying things from existing similar lints. If you are stuck, don't hesitate to ask on Zulip or in the issue/PR.