Clippy

A collection of lints to catch common mistakes and improve your Rust code.

There are over 750 lints included in this crate!

Lints are divided into categories, each with a default lint level. You can choose how much Clippy is supposed to ~~annoy~~ help you by changing the lint level by category.

Category	Description	Default level
`clippy::all`	all lints that are on by default (correctness, suspicious, style, complexity, perf)	**warn/deny**
`clippy::correctness`	code that is outright wrong or useless	**deny**
`clippy::suspicious`	code that is most likely wrong or useless	**warn**
`clippy::style`	code that should be written in a more idiomatic way	**warn**
`clippy::complexity`	code that does something simple but in a complex way	**warn**
`clippy::perf`	code that can be written to run faster	**warn**
`clippy::pedantic`	lints which are rather strict or have occasional false positives	allow
`clippy::restriction`	lints which prevent the use of language and library features[1]	allow
`clippy::nursery`	new lints that are still under development	allow
`clippy::cargo`	lints for the cargo manifest	allow

More to come, please file an issue if you have ideas!

The `restriction` category should, *emphatically*, not be enabled as a whole. The contained lints may lint against perfectly reasonable code, may not have an alternative suggestion, and may contradict any other lints (including other categories). Lints should be considered on a case-by-case basis before enabling.

---

1. Some use cases for `restriction` lints include:
   - Strict coding styles (e.g. `clippy::else_if_without_else`).
   - Additional restrictions on CI (e.g. `clippy::todo`).
   - Preventing panicking in certain functions (e.g. `clippy::unwrap_used`).
   - Running a lint only on a subset of code (e.g. `#[forbid(clippy::float_arithmetic)]` on a module).

   ↩

Installation

If you're using `rustup` to install and manage your Rust toolchains, Clippy is usually **already installed**. In that case you can skip this chapter and go to the Usage chapter.

> Note: If you used the `minimal` profile when installing a Rust toolchain, Clippy is not automatically installed.

Using Rustup

If Clippy was not installed for a toolchain, it can be installed with

```
$ rustup component add clippy [--toolchain=<name>]
```

From Source

Take a look at the Basics chapter in the Clippy developer guide to find step-by-step instructions on how to build and install Clippy from source.

Usage

This chapter describes how to use Clippy to get the most out of it. Clippy can be used as a `cargo` subcommand or, like `rustc`, directly with the `clippy-driver` binary.

Note: This chapter assumes that you have Clippy installed already. If you're not sure, take a look at the Installation chapter.

Cargo subcommand

The easiest and most common way to run Clippy is through `cargo`. To do that, just run

```
cargo clippy
```

Lint configuration

The above command will run the default set of lints, which are included in the lint group `clippy::all`. You might want to use even more lints, or you may not agree with every Clippy lint, and for that there are ways to configure lint levels.

Note: Clippy is meant to be used with a generous sprinkling of `#[allow(..)]` s through your code. So if you disagree with a lint, don't feel bad disabling them for parts of your code or the whole project.

Command line

You can configure lint levels on the command line by adding `-A/W/D clippy::lint_name` like this:

```
cargo clippy -- -Aclippy::style -Wclippy::box_default -Dclippy::perf
```

For CI all warnings can be elevated to errors which will in turn fail the build and cause Clippy to exit with a code other than `0`.

```
cargo clippy -- -Dwarnings
```

Note: Adding `-D warnings` will cause your build to fail if **any** warnings are found in your code. That includes warnings found by rustc (e.g. `dead_code`, etc.).

For more information on configuring lint levels, see the rustc documentation.

Even more lints

Clippy has lint groups which are allow-by-default. This means, that you will have to enable the lints in those groups manually.

For a full list of all lints with their description and examples, please refer to Clippy's lint list. The two most important allow-by-default groups are described below:

`clippy::pedantic`

The first group is the `pedantic` group. This group contains really opinionated lints, that may have some intentional false positives in order to prevent false negatives. So while this group is ready to be used in production, you can expect to sprinkle multiple `#[allow(..)]` s in your code. If you find any false positives, you're still welcome to report them to us for future improvements.

FYI: Clippy uses the whole group to lint itself.

`clippy::restriction`

The second group is the `restriction` group. This group contains lints that "restrict" the language in some way. For example the `clippy::unwrap` lint from this group won't allow you to use `.unwrap()` in your code. You may want to look through the lints in this group and enable the ones that fit your need.

Note: You shouldn't enable the whole lint group, but cherry-pick lints from this group. Some lints in this group will even contradict other Clippy lints!

Too many lints

The most opinionated warn-by-default group of Clippy is the `clippy::style` group. Some people prefer to disable this group completely and then cherry-pick some lints they like from this group. The same is of course possible with every other of Clippy's lint groups.

Note: We try to keep the warn-by-default groups free from false positives (FP). If you find that a lint wrongly triggers, please report it in an issue (if there isn't an issue for that FP already)

Source Code

You can configure lint levels in source code the same way you can configure `rustc` lints:

```
#![allow(clippy::style)]

#[warn(clippy::box_default)]
fn main() {
    let _ = Box::<String>::new(Default::default());
    // ^ warning: `Box::new(_)` of default value
}
```

Automatically applying Clippy suggestions

Clippy can automatically apply some lint suggestions, just like the compiler. Note that `--fix` implies `--all-targets`, so it can fix as much code as it can.

```
cargo clippy --fix
```

Workspaces

All the usual workspace options should work with Clippy. For example the following command will run Clippy on the `example` crate in your workspace:

```
cargo clippy -p example
```

As with `cargo check`, this includes dependencies that are members of the workspace, like path dependencies. If you want to run Clippy **only** on the given crate, use the `--no-deps` option like this:

```
cargo clippy -p example -- --no-deps
```

Using Clippy without `cargo`: `clippy-driver`

Clippy can also be used in projects that do not use cargo. To do so, run `clippy-driver` with the same arguments you use for `rustc`. For example:

```
clippy-driver --edition 2018 -Cpanic=abort foo.rs
```

Note: `clippy-driver` is designed for running Clippy and should not be used as a general replacement for `rustc`. `clippy-driver` may produce artifacts that are not optimized as expected, for example.

Configuring Clippy

> **Note:** The configuration file is unstable and may be deprecated in the future.

Some lints can be configured in a TOML file named `clippy.toml` or `.clippy.toml`, which is searched for in:

1. The directory specified by the `CLIPPY_CONF_DIR` environment variable, or
2. The directory specified by the CARGO_MANIFEST_DIR environment variable, or
3. The current directory.

It contains a basic `variable = value` mapping e.g.

```
avoid-breaking-exported-api = false
disallowed-names = ["toto", "tata", "titi"]
```

The table of configurations contains all config values, their default, and a list of lints they affect. Each configurable lint, also contains information about these values.

For configurations that are a list type with default values such as disallowed-names, you can use the unique value `".."` to extend the default values instead of replacing them.

```
# default of disallowed-names is ["foo", "baz", "quux"]
disallowed-names = ["bar", ".."] # -> ["bar", "foo", "baz", "quux"]
```

To deactivate the "for further information visit *lint-link*" message you can define the `CLIPPY_DISABLE_DOCS_LINKS` environment variable.

Allowing/Denying Lints

Attributes in Code

You can add attributes to your code to `allow` / `warn` / `deny` Clippy lints:

- the whole set of `warn`-by-default lints using the `clippy` lint group (`#![allow(clippy::all)]`)
- all lints using both the `clippy` and `clippy::pedantic` lint groups (`#![warn(clippy::all, clippy::pedantic)]`. Note that `clippy::pedantic` contains some very aggressive lints prone to false positives.
- only some lints (`#![deny(clippy::single_match, clippy::box_vec)]`, etc.)
- `allow` / `warn` / `deny` can be limited to a single function or module using `#[allow(...)]`, etc.

Note: `allow` means to suppress the lint for your code. With `warn` the lint will only emit a warning, while with `deny` the lint will emit an error, when triggering for your code. An error causes Clippy to exit with an error code, so is most useful in scripts used in CI/CD.

Command Line Flags

If you do not want to include your lint levels in the code, you can globally enable/disable lints by passing extra flags to Clippy during the run:

To allow `lint_name`, run

```
cargo clippy -- -A clippy::lint_name
```

And to warn on `lint_name`, run

```
cargo clippy -- -W clippy::lint_name
```

This also works with lint groups. For example, you can run Clippy with warnings for all pedantic lints enabled:

```
cargo clippy -- -W clippy::pedantic
```

If you care only about a certain lints, you can allow all others and then explicitly warn on the lints you are interested in:

```
cargo clippy -- -A clippy::all -W clippy::useless_format -W clippy::...
```

Lints Section in `Cargo.toml`

Finally, lints can be allowed/denied using the lints section) in the `Cargo.toml` file:

To deny `clippy::enum_glob_use`, put the following in the `Cargo.toml`:

```
[lints.clippy]
enum_glob_use = "deny"
```

For more details and options, refer to the Cargo documentation.

Specifying the minimum supported Rust version

Projects that intend to support old versions of Rust can disable lints pertaining to newer features by specifying the minimum supported Rust version (MSRV) in the Clippy configuration file.

```
msrv = "1.30.0"
```

The MSRV can also be specified as an attribute, like below.

```
#![feature(custom_inner_attributes)]
#![clippy::msrv = "1.30.0"]

fn main() {
    ...
}
```

You can also omit the patch version when specifying the MSRV, so `msrv = 1.30` is equivalent to `msrv = 1.30.0`.

Note: `custom_inner_attributes` is an unstable feature, so it has to be enabled explicitly.

Lints that recognize this configuration option can be found here

Disabling evaluation of certain code

> **Note:** This should only be used in cases where other solutions, like `#[allow(clippy::all)]`, are not sufficient.

Very rarely, you may wish to prevent Clippy from evaluating certain sections of code entirely. You can do this with conditional compilation by checking that the `clippy` cfg is not set. You may need to provide a stub so that the code compiles:

```
#[cfg(not(clippy))]
include!(concat!(env!("OUT_DIR"), "/my_big_function-generated.rs"));

#[cfg(clippy)]
fn my_big_function(_input: &str) -> Option<MyStruct> {
    None
}
```

Lint Configuration Options

The following list shows each configuration option, along with a description, its default value, an example and lints affected.

---

absolute-paths-allowed-crates

Which crates to allow absolute paths from

Default Value: `[]`

Affected lints:

- absolute_paths

absolute-paths-max-segments

The maximum number of segments a path can have before being linted, anything above this will be linted.

Default Value: `2`

Affected lints:

- absolute_paths

accept-comment-above-attributes

Whether to accept a safety comment to be placed above the attributes for the `unsafe` block

Default Value: `true`

Affected lints:

- undocumented_unsafe_blocks

accept-comment-above-statement

Whether to accept a safety comment to be placed above the statement containing the `unsafe` block

Default Value: `true`

Affected lints:

- undocumented_unsafe_blocks

allow-comparison-to-zero

Don't lint when comparing the result of a modulo operation to zero.

Default Value: `true`

Affected lints:

- modulo_arithmetic

allow-dbg-in-tests

Whether `dbg!` should be allowed in test functions or `#[cfg(test)]`

Default Value: `false`

Affected lints:

- dbg_macro

allow-exact-repetitions

Whether an item should be allowed to have the same name as its containing module

Default Value: `true`

Affected lints:

- module_name_repetitions

allow-expect-in-consts

Whether `expect` should be allowed in code always evaluated at compile time

Default Value: `true`

Affected lints:

- expect_used

allow-expect-in-tests

Whether `expect` should be allowed in test functions or `#[cfg(test)]`

Default Value: `false`

Affected lints:

- expect_used

allow-indexing-slicing-in-tests

Whether `indexing_slicing` should be allowed in test functions or `#[cfg(test)]`

Default Value: `false`

Affected lints:

- indexing_slicing

allow-mixed-uninlined-format-args

Whether to allow mixed uninlined format args, e.g. `format!("{} {}", a, foo.bar)`

Default Value: `true`

Affected lints:

- uninlined_format_args

allow-one-hash-in-raw-strings

Whether to allow `r#""#` when `r""` can be used

Default Value: `false`

Affected lints:

- needless_raw_string_hashes

allow-panic-in-tests

Whether `panic` should be allowed in test functions or `#[cfg(test)]`

Default Value: `false`

Affected lints:

- panic

allow-print-in-tests

Whether print macros (ex. `println!`) should be allowed in test functions or `#[cfg(test)]`

Default Value: `false`

Affected lints:

- print_stderr
- print_stdout

allow-private-module-inception

Whether to allow module inception if it's not public.

Default Value: `false`

Affected lints:

- module_inception

allow-renamed-params-for

List of trait paths to ignore when checking renamed function parameters.

Example

```
allow-renamed-params-for = [ "std::convert::From" ]
```

Noteworthy

- By default, the following traits are ignored: `From` , `TryFrom` , `FromStr`
- `".."` can be used as part of the list to indicate that the configured values should be appended to the default configuration of Clippy. By default, any configuration will replace the default value.

Default Value: `["core::convert::From", "core::convert::TryFrom", "core::str::FromStr"]`

Affected lints:

- renamed_function_params

allow-unwrap-in-consts

Whether `unwrap` should be allowed in code always evaluated at compile time

Default Value: `true`

Affected lints:

- unwrap_used

allow-unwrap-in-tests

Whether `unwrap` should be allowed in test functions or `#[cfg(test)]`

Default Value: `false`

Affected lints:

- unwrap_used

allow-useless-vec-in-tests

Whether `useless_vec` should ignore test functions or `#[cfg(test)]`

Default Value: `false`

Affected lints:

- useless_vec

allowed-dotfiles

Additional dotfiles (files or directories starting with a dot) to allow

Default Value: `[]`

Affected lints:

- path_ends_with_ext

allowed-duplicate-crates

A list of crate names to allow duplicates of

Default Value: `[]`

Affected lints:

- multiple_crate_versions

allowed-idents-below-min-chars

Allowed names below the minimum allowed characters. The value `".."` can be used as part of the list to indicate, that the configured values should be appended to the default configuration of Clippy. By default, any configuration will replace the default value.

Default Value: `["i", "j", "x", "y", "z", "w", "n"]`

Affected lints:

- min_ident_chars

allowed-prefixes

List of prefixes to allow when determining whether an item's name ends with the module's name. If the rest of an item's name is an allowed prefix (e.g. item `ToFoo` or `to_foo` in module `foo`), then don't emit a warning.

Example

```
allowed-prefixes = [ "to", "from" ]
```

Noteworthy

- By default, the following prefixes are allowed: `to`, `as`, `into`, `from`, `try_into` and `try_from`
- PascalCase variant is included automatically for each snake_case variant (e.g. if `try_into` is included, `TryInto` will also be included)
- Use `".."` as part of the list to indicate that the configured values should be appended to the default configuration of Clippy. By default, any configuration will replace the default value

Default Value: `["to", "as", "into", "from", "try_into", "try_from"]`

Affected lints:

- module_name_repetitions

allowed-scripts

The list of unicode scripts allowed to be used in the scope.

Default Value: `["Latin"]`

Affected lints:

- disallowed_script_idents

allowed-wildcard-imports

List of path segments allowed to have wildcard imports.

Example

```toml
allowed-wildcard-imports = [ "utils", "common" ]
```

Noteworthy

1. This configuration has no effects if used with `warn_on_all_wildcard_imports = true`.
2. Paths with any segment that containing the word 'prelude' are already allowed by default.

Default Value: `[]`

Affected lints:

- wildcard_imports

arithmetic-side-effects-allowed

Suppress checking of the passed type names in all types of operations.

If a specific operation is desired, consider using `arithmetic_side_effects_allowed_binary` or `arithmetic_side_effects_allowed_unary` instead.

Example

```toml
arithmetic-side-effects-allowed = ["SomeType", "AnotherType"]
```

Noteworthy

A type, say `SomeType`, listed in this configuration has the same behavior of `["SomeType" , "*"], ["*", "SomeType"]` in `arithmetic_side_effects_allowed_binary`.

Default Value: `[]`

Affected lints:

- arithmetic_side_effects

arithmetic-side-effects-allowed-binary

Suppress checking of the passed type pair names in binary operations like addition or multiplication.

Supports the "*" wildcard to indicate that a certain type won't trigger the lint regardless of the involved counterpart. For example, `["SomeType", "*"]` or `["*", "AnotherType"]`.

Pairs are asymmetric, which means that `["SomeType", "AnotherType"]` is not the same as `["AnotherType", "SomeType"]`.

Example

```
arithmetic-side-effects-allowed-binary = [["SomeType" , "f32"], ["AnotherType", "*"]]
```

Default Value: `[]`

Affected lints:

- arithmetic_side_effects

arithmetic-side-effects-allowed-unary

Suppress checking of the passed type names in unary operations like "negation" (`-`).

Example

```
arithmetic-side-effects-allowed-unary = ["SomeType", "AnotherType"]
```

Default Value: `[]`

Affected lints:

- arithmetic_side_effects

array-size-threshold

The maximum allowed size for arrays on the stack

Default Value: `16384`

Affected lints:

- large_const_arrays
- large_stack_arrays

avoid-breaking-exported-api

Suppress lints whenever the suggested change would cause breakage for other crates.

Default Value: `true`

Affected lints:

- box_collection
- enum_variant_names
- large_types_passed_by_value
- linkedlist
- needless_pass_by_ref_mut
- option_option
- owned_cow
- rc_buffer
- rc_mutex
- redundant_allocation
- ref_option
- single_call_fn
- trivially_copy_pass_by_ref
- unnecessary_box_returns
- unnecessary_wraps
- unused_self
- upper_case_acronyms
- vec_box
- wrong_self_convention

await-holding-invalid-types

The list of types which may not be held across an await point.

Default Value: `[]`

Affected lints:

- await_holding_invalid_type

cargo-ignore-publish

For internal testing only, ignores the current `publish` settings in the Cargo manifest.

Default Value: `false`

Affected lints:

- cargo_common_metadata

check-incompatible-msrv-in-tests

Whether to check MSRV compatibility in `#[test]` and `#[cfg(test)]` code.

Default Value: `false`

Affected lints:

- incompatible_msrv

check-inconsistent-struct-field-initializers

Whether to suggest reordering constructor fields when initializers are present.

Warnings produced by this configuration aren't necessarily fixed by just reordering the fields. Even if the suggested code would compile, it can change semantics if the initializer expressions have side effects. The following example from rust-clippy#11846 shows how the suggestion can run into borrow check errors:

```
struct MyStruct {
    vector: Vec<u32>,
    length: usize
}
fn main() {
    let vector = vec![1,2,3];
    MyStruct { length: vector.len(), vector};
}
```

Default Value: `false`

Affected lints:

- inconsistent_struct_constructor

check-private-items

Whether to also run the listed lints on private items.

Default Value: `false`

Affected lints:

- missing_errors_doc
- missing_panics_doc
- missing_safety_doc
- unnecessary_safety_doc

cognitive-complexity-threshold

The maximum cognitive complexity a function can have

Default Value: `25`

Affected lints:

- cognitive_complexity

const-literal-digits-threshold

The minimum digits a const float literal must have to supress the `excessive_precicion` lint

Default Value: `30`

Affected lints:

- excessive_precision

disallowed-macros

The list of disallowed macros, written as fully qualified paths.

Fields:

- `path` (required): the fully qualified path to the macro that should be disallowed
- `reason` (optional): explanation why this macro is disallowed
- `replacement` (optional): suggested alternative macro
- `allow-invalid` (optional, `false` by default): when set to `true`, it will ignore this entry if the path doesn't exist, instead of emitting an error

Default Value: `[]`

Affected lints:

- disallowed_macros

disallowed-methods

The list of disallowed methods, written as fully qualified paths.

Fields:

- `path` (required): the fully qualified path to the method that should be disallowed
- `reason` (optional): explanation why this method is disallowed
- `replacement` (optional): suggested alternative method
- `allow-invalid` (optional, `false` by default): when set to `true`, it will ignore this entry if the path doesn't exist, instead of emitting an error

Default Value: `[]`

Affected lints:

- disallowed_methods

disallowed-names

The list of disallowed names to lint about. NB: `bar` is not here since it has legitimate uses. The value `".."` can be used as part of the list to indicate that the configured values should be appended to the default configuration of Clippy. By default, any configuration will replace the default value.

Default Value: `["foo", "baz", "quux"]`

Affected lints:

- disallowed_names

disallowed-types

The list of disallowed types, written as fully qualified paths.

Fields:

- `path` (required): the fully qualified path to the type that should be disallowed
- `reason` (optional): explanation why this type is disallowed
- `replacement` (optional): suggested alternative type
- `allow-invalid` (optional, `false` by default): when set to `true`, it will ignore this entry if the path doesn't exist, instead of emitting an error

Default Value: `[]`

Affected lints:

- disallowed_types

doc-valid-idents

The list of words this lint should not consider as identifiers needing ticks. The value `".."` can be used as part of the list to indicate, that the configured values should be appended to the default configuration of Clippy. By default, any configuration will replace the default value. For example:

- `doc-valid-idents = ["ClipPy"]` would replace the default list with `["ClipPy"]`.
- `doc-valid-idents = ["ClipPy", ".."]` would append `ClipPy` to the default list.

Default Value: `["KiB", "MiB", "GiB", "TiB", "PiB", "EiB", "MHz", "GHz", "THz", "AccessKit", "CoAP", "CoreFoundation", "CoreGraphics", "CoreText", "DevOps", "Direct2D", "Direct3D", "DirectWrite", "DirectX", "ECMAScript", "GPLv2", "GPLv3", "GitHub", "GitLab", "IPv4", "IPv6", "InfiniBand", "RoCE", "ClojureScript", "CoffeeScript", "JavaScript", "PostScript", "PureScript", "TypeScript", "PowerPC", "WebAssembly", "NaN", "NaNs", "OAuth", "GraphQL", "OCaml", "OpenAL", "OpenDNS", "OpenGL", "OpenMP", "OpenSSH", "OpenSSL", "OpenStreetMap", "OpenTelemetry", "OpenType", "WebGL", "WebGL2", "WebGPU", "WebRTC", "WebSocket", "WebTransport", "WebP", "OpenExr", "YCbCr", "sRGB", "TensorFlow", "TrueType", "iOS", "macOS", "FreeBSD", "NetBSD", "OpenBSD", "NixOS", "TeX", "LaTeX", "BibTeX", "BibLaTeX", "MinGW", "CamelCase"]`

Affected lints:

- doc_markdown

enable-raw-pointer-heuristic-for-send

Whether to apply the raw pointer heuristic to determine if a type is `Send`.

Default Value: `true`

Affected lints:

- non_send_fields_in_send_ty

enforce-iter-loop-reborrow

Whether to recommend using implicit into iter for reborrowed values.

Example

```
let mut vec = vec![1, 2, 3];
let rmvec = &mut vec;
for _ in rmvec.iter() {}
for _ in rmvec.iter_mut() {}
```

Use instead:

```
let mut vec = vec![1, 2, 3];
let rmvec = &mut vec;
for _ in &*rmvec {}
for _ in &mut *rmvec {}
```

Default Value: `false`

Affected lints:

- explicit_iter_loop

enforced-import-renames

The list of imports to always rename, a fully qualified path followed by the rename.

Default Value: `[]`

Affected lints:

- missing_enforced_import_renames

enum-variant-name-threshold

The minimum number of enum variants for the lints about variant names to trigger

Default Value: `3`

Affected lints:

- enum_variant_names

enum-variant-size-threshold

The maximum size of an enum's variant to avoid box suggestion

Default Value: `200`

Affected lints:

- large_enum_variant

excessive-nesting-threshold

The maximum amount of nesting a block can reside in

Default Value: `0`

Affected lints:

- excessive_nesting

future-size-threshold

The maximum byte size a `Future` can have, before it triggers the `clippy::large_futures` lint

Default Value: `16384`

Affected lints:

- large_futures

ignore-interior-mutability

A list of paths to types that should be treated as if they do not contain interior mutability

Default Value: `["bytes::Bytes"]`

Affected lints:

- borrow_interior_mutable_const
- declare_interior_mutable_const
- ifs_same_cond
- mutable_key_type

inherent-impl-lint-scope

Sets the scope ("crate", "file", or "module") in which duplicate inherent `impl` blocks for the same type are linted.

Default Value: `"crate"`

Affected lints:

- multiple_inherent_impl

large-error-threshold

The maximum size of the `Err`-variant in a `Result` returned from a function

Default Value: `128`

Affected lints:

- `result_large_err`

`lint-commented-code`

Whether collapsible `if` and `else if` chains are linted if they contain comments inside the parts that would be collapsed.

Default Value: `false`

Affected lints:

- `collapsible_else_if`
- `collapsible_if`

`literal-representation-threshold`

The lower bound for linting decimal literals

Default Value: `16384`

Affected lints:

- `decimal_literal_representation`

`matches-for-let-else`

Whether the matches should be considered by the lint, and whether there should be filtering for common types.

Default Value: `"WellKnownTypes"`

Affected lints:

- `manual_let_else`

`max-fn-params-bools`

The maximum number of bool parameters a function can have

Default Value: `3`

Affected lints:

- fn_params_excessive_bools

max-include-file-size

The maximum size of a file included via `include_bytes!()` or `include_str!()`, in bytes

Default Value: `1000000`

Affected lints:

- large_include_file

max-struct-bools

The maximum number of bool fields a struct can have

Default Value: `3`

Affected lints:

- struct_excessive_bools

max-suggested-slice-pattern-length

When Clippy suggests using a slice pattern, this is the maximum number of elements allowed in the slice pattern that is suggested. If more elements are necessary, the lint is suppressed. For example, `[_, _, _, e, ..]` is a slice pattern with 4 elements.

Default Value: `3`

Affected lints:

- index_refutable_slice

max-trait-bounds

The maximum number of bounds a trait can have to be linted

Default Value: `3`

Affected lints:

- `type_repetition_in_bounds`

`min-ident-chars-threshold`

Minimum chars an ident can have, anything below or equal to this will be linted.

Default Value: `1`

Affected lints:

- `min_ident_chars`

`missing-docs-allow-unused`

Whether to allow fields starting with an underscore to skip documentation requirements

Default Value: `false`

Affected lints:

- `missing_docs_in_private_items`

`missing-docs-in-crate-items`

Whether to **only** check for missing documentation in items visible within the current crate. For example, `pub(crate)` items.

Default Value: `false`

Affected lints:

- `missing_docs_in_private_items`

`module-item-order-groupings`

The named groupings of different source item kinds within modules.

Default Value: `[["modules", ["extern_crate", "mod", "foreign_mod"]], ["use", ["use"]], ["macros", ["macro"]], ["global_asm", ["global_asm"]], ["UPPER_SNAKE_CASE", ["static", "const"]], ["PascalCase", ["ty_alias", "enum", "struct", "union", "trait", "trait_alias", "impl"]], ["lower_snake_case", ["fn"]]]`

Affected lints:

- arbitrary_source_item_ordering

module-items-ordered-within-groupings

Whether the items within module groups should be ordered alphabetically or not.

This option can be configured to "all", "none", or a list of specific grouping names that should be checked (e.g. only "enums").

Default Value: `"none"`

Affected lints:

- arbitrary_source_item_ordering

msrv

The minimum rust version that the project supports. Defaults to the `rust-version` field in `Cargo.toml`

Default Value: `current version`

Affected lints:

- allow_attributes
- allow_attributes_without_reason
- almost_complete_range
- approx_constant
- assigning_clones
- borrow_as_ptr
- cast_abs_to_unsigned
- checked_conversions
- cloned_instead_of_copied
- collapsible_match
- collapsible_str_replace
- deprecated_cfg_attr
- derivable_impls
- err_expect
- filter_map_next
- from_over_into
- if_then_some_else_none
- index_refutable_slice
- inefficient_to_string
- io_other_error
- iter_kv_map
- legacy_numeric_constants
- lines_filter_map_ok

- manual_abs_diff
- manual_bits
- manual_c_str_literals
- manual_clamp
- manual_div_ceil
- manual_flatten
- manual_hash_one
- manual_is_ascii_check
- manual_is_power_of_two
- manual_let_else
- manual_midpoint
- manual_non_exhaustive
- manual_option_as_slice
- manual_pattern_char_comparison
- manual_range_contains
- manual_rem_euclid
- manual_repeat_n
- manual_retain
- manual_slice_fill
- manual_slice_size_calculation
- manual_split_once
- manual_str_repeat
- manual_strip
- manual_try_fold
- map_clone
- map_unwrap_or
- map_with_unused_argument_over_ranges
- match_like_matches_macro
- mem_replace_option_with_some
- mem_replace_with_default
- missing_const_for_fn
- needless_borrow
- non_std_lazy_statics
- option_as_ref_deref
- or_fun_call
- ptr_as_ptr
- question_mark
- redundant_field_names
- redundant_static_lifetimes
- repeat_vec_with_capacity
- same_item_push
- seek_from_current
- to_digit_is_some
- transmute_ptr_to_ref
- tuple_array_conversions
- type_repetition_in_bounds
- unchecked_time_subtraction
- uninlined_format_args

- unnecessary_lazy_evaluations
- unnecessary_unwrap
- unnested_or_patterns
- unused_trait_names
- use_self
- zero_ptr

pass-by-value-size-limit

The minimum size (in bytes) to consider a type for passing by reference instead of by value.

Default Value: `256`

Affected lints:

- large_types_passed_by_value

pub-underscore-fields-behavior

Lint "public" fields in a struct that are prefixed with an underscore based on their exported visibility, or whether they are marked as "pub".

Default Value: `"PubliclyExported"`

Affected lints:

- pub_underscore_fields

recursive-self-in-type-definitions

Whether the type itself in a struct or enum should be replaced with `Self` when encountering recursive types.

Default Value: `true`

Affected lints:

- use_self

semicolon-inside-block-ignore-singleline

Whether to lint only if it's multiline.

Default Value: `false`

Affected lints:

- semicolon_inside_block

semicolon-outside-block-ignore-multiline

Whether to lint only if it's singleline.

Default Value: `false`

Affected lints:

- semicolon_outside_block

single-char-binding-names-threshold

The maximum number of single char bindings a scope may have

Default Value: `4`

Affected lints:

- many_single_char_names

source-item-ordering

Which kind of elements should be ordered internally, possible values being `enum`, `impl`, `module`, `struct`, `trait`.

Default Value: `["enum", "impl", "module", "struct", "trait"]`

Affected lints:

- arbitrary_source_item_ordering

stack-size-threshold

The maximum allowed stack size for functions in bytes

Default Value: `512000`

Affected lints:

- `large_stack_frames`

`standard-macro-braces`

Enforce the named macros always use the braces specified.

A `MacroMatcher` can be added like so `{ name = "macro_name", brace = "(" }`. If the macro could be used with a full path two `MacroMatcher`s have to be added one with the full path `crate_name::macro_name` and one with just the macro name.

Default Value: `[]`

Affected lints:

- `nonstandard_macro_braces`

`struct-field-name-threshold`

The minimum number of struct fields for the lints about field names to trigger

Default Value: `3`

Affected lints:

- `struct_field_names`

`suppress-restriction-lint-in-const`

Whether to suppress a restriction lint in constant code. In same cases the restructured operation might not be unavoidable, as the suggested counterparts are unavailable in constant code. This configuration will cause restriction lints to trigger even if no suggestion can be made.

Default Value: `false`

Affected lints:

- `indexing_slicing`

`too-large-for-stack`

The maximum size of objects (in bytes) that will be linted. Larger objects are ok on the heap

Default Value: `200`

Affected lints:

- boxed_local
- useless_vec

too-many-arguments-threshold

The maximum number of argument a function or method can have

Default Value: 7

Affected lints:

- too_many_arguments

too-many-lines-threshold

The maximum number of lines a function or method can have

Default Value: 100

Affected lints:

- too_many_lines

trait-assoc-item-kinds-order

The order of associated items in traits.

Default Value: ["const", "type", "fn"]

Affected lints:

- arbitrary_source_item_ordering

trivial-copy-size-limit

The maximum size (in bytes) to consider a `Copy` type for passing by value instead of by reference.

Default Value: target_pointer_width

Affected lints:

- trivially_copy_pass_by_ref

type-complexity-threshold

The maximum complexity a type can have

Default Value: `250`

Affected lints:

- type_complexity

unnecessary-box-size

The byte size a `T` in `Box<T>` can have, below which it triggers the `clippy::unnecessary_box` lint

Default Value: `128`

Affected lints:

- unnecessary_box_returns

unreadable-literal-lint-fractions

Should the fraction of a decimal be linted to include separators.

Default Value: `true`

Affected lints:

- unreadable_literal

upper-case-acronyms-aggressive

Enables verbose mode. Triggers if there is more than one uppercase char next to each other

Default Value: `false`

Affected lints:

- upper_case_acronyms

vec-box-size-threshold

The size of the boxed type in bytes, where boxing in a `Vec` is allowed

Default Value: `4096`

Affected lints:

- `vec_box`

verbose-bit-mask-threshold

The maximum allowed size of a bit mask before suggesting to use 'trailing_zeros'

Default Value: `1`

Affected lints:

- `verbose_bit_mask`

warn-on-all-wildcard-imports

Whether to emit warnings on all wildcard imports, including those from `prelude`, from `super` in tests, or for `pub use` reexports.

Default Value: `false`

Affected lints:

- `wildcard_imports`

warn-unsafe-macro-metavars-in-private-macros

Whether to also emit warnings for unsafe blocks with metavariable expansions in **private** macros.

Default Value: `false`

Affected lints:

- `macro_metavars_in_unsafe`

Clippy's Lints

Clippy offers a bunch of additional lints, to help its users write more correct and idiomatic Rust code. A full list of all lints, that can be filtered by category, lint level or keywords, can be found in the Clippy lint documentation.

This chapter will give an overview of the different lint categories, which kind of lints they offer and recommended actions when you should see a lint out of that category. For examples, see the Clippy lint documentation and filter by category.

The different lint groups were defined in the Clippy 1.0 RFC.

Correctness

The `clippy::correctness` group is the only lint group in Clippy which lints are deny-by-default and abort the compilation when triggered. This is for good reason: If you see a `correctness` lint, it means that your code is outright wrong or useless, and you should try to fix it.

Lints in this category are carefully picked and should be free of false positives. So just `#[allow]` ing those lints is not recommended.

Suspicious

The `clippy::suspicious` group is similar to the correctness lints in that it contains lints that trigger on code that is really *sus* and should be fixed. As opposed to correctness lints, it might be possible that the linted code is intentionally written like it is.

It is still recommended to fix code that is linted by lints out of this group instead of `#[allow]` ing the lint. In case you intentionally have written code that offends the lint you should specifically and locally `#[allow]` the lint and add give a reason why the code is correct as written.

Complexity

The `clippy::complexity` group offers lints that give you suggestions on how to simplify your code. It mostly focuses on code that can be written in a shorter and more readable way, while preserving the semantics.

If you should see a complexity lint, it usually means that you can remove or replace some code, and it is recommended to do so. However, if you need the more complex code for some expressiveness reason, it is recommended to allow complexity lints on a case-by-case basis.

Perf

The `clippy::perf` group gives you suggestions on how you can increase the performance of your

code. Those lints are mostly about code that the compiler can't trivially optimize, but has to be written in a slightly different way to make the optimizer job easier.

Perf lints are usually easy to apply, and it is recommended to do so.

Style

The `clippy::style` group is mostly about writing idiomatic code. Because style is subjective, this lint group is the most opinionated warn-by-default group in Clippy.

If you see a style lint, applying the suggestion usually makes your code more readable and idiomatic. But because we know that this is opinionated, feel free to sprinkle `#[allow]` s for style lints in your code or `#![allow]` a style lint on your whole crate if you disagree with the suggested style completely.

Pedantic

The `clippy::pedantic` group makes Clippy even more *pedantic*. You can enable the whole group with `#![warn(clippy::pedantic)]` in the `lib.rs` / `main.rs` of your crate. This lint group is for Clippy power users that want an in depth check of their code.

> *Note:* Instead of enabling the whole group (like Clippy itself does), you may want to cherry-pick lints out of the pedantic group.

If you enable this group, expect to also use `#[allow]` attributes generously throughout your code. Lints in this group are designed to be pedantic and false positives sometimes are intentional in order to prevent false negatives.

Restriction

The `clippy::restriction` group contains lints that will *restrict* you from using certain parts of the Rust language. It is **not** recommended to enable the whole group, but rather cherry-pick lints that are useful for your code base and your use case.

> *Note:* Clippy will produce a warning if it finds a `#![warn(clippy::restriction)]` attribute in your code!

Lints from this group will restrict you in some way. If you enable a restriction lint for your crate it is recommended to also fix code that this lint triggers on. However, those lints are really strict by design, and you might want to `#[allow]` them in some special cases, with a comment justifying that.

Cargo

The `clippy::cargo` group gives you suggestions on how to improve your `Cargo.toml` file. This might be especially interesting if you want to publish your crate and are not sure if you have all useful information in your `Cargo.toml` .

Nursery

The `clippy::nursery` group contains lints which are buggy or need more work. It is **not** recommended to enable the whole group, but rather cherry-pick lints that are useful for your code base and your use case.

Deprecated

The `clippy::deprecated` is empty lints that exist to ensure that `#[allow(lintname)]` still compiles after the lint was deprecated. Deprecation "removes" lints by removing their functionality and marking them as deprecated, which may cause further warnings but cannot cause a compiler error.

Attributes for Crate Authors

In some cases it is possible to extend Clippy coverage to 3rd party libraries. To do this, Clippy provides attributes that can be applied to items in the 3rd party crate.

`#[clippy::format_args]`

Available since Clippy v1.85

This attribute can be added to a macro that supports `format!`, `println!`, or similar syntax. It tells Clippy that the macro is a formatting macro, and that the arguments to the macro should be linted as if they were arguments to `format!`. Any lint that would apply to a `format!` call will also apply to the macro call. The macro may have additional arguments before the format string, and these will be ignored.

Example

```
/// A macro that prints a message if a condition is true.
#[macro_export]
#[clippy::format_args]
macro_rules! print_if {
    ($condition:expr, $($args:tt)+) => {{
        if $condition {
            println!($($args)+)
        }
    }};
}
```

`#[clippy::has_significant_drop]`

Available since Clippy v1.60

The `clippy::has_significant_drop` attribute can be added to types whose Drop impls have an important side effect, such as unlocking a mutex, making it important for users to be able to accurately understand their lifetimes. When a temporary is returned in a function call in a match scrutinee, its lifetime lasts until the end of the match block, which may be surprising.

Example

```rust
#[clippy::has_significant_drop]
struct CounterWrapper<'a> {
    counter: &'a Counter,
}

impl<'a> Drop for CounterWrapper<'a> {
    fn drop(&mut self) {
        self.counter.i.fetch_sub(1, Ordering::Relaxed);
    }
}
```

Continuous Integration

It is recommended to run Clippy on CI with `-Dwarnings`, so that Clippy lints prevent CI from passing. To enforce errors on warnings on all `cargo` commands not just `cargo clippy`, you can set the env var `RUSTFLAGS="-Dwarnings"`.

We recommend to use Clippy from the same toolchain, that you use for compiling your crate for maximum compatibility. E.g. if your crate is compiled with the `stable` toolchain, you should also use `stable` Clippy.

> *Note:* New Clippy lints are first added to the `nightly` toolchain. If you want to help with improving Clippy and have CI resources left, please consider adding a `nightly` Clippy check to your CI and report problems like false positives back to us. With that we can fix bugs early, before they can get to stable.

This chapter will give an overview on how to use Clippy on different popular CI providers.

GitHub Actions

GitHub hosted runners using the latest stable version of Rust have Clippy pre-installed. It is as simple as running `cargo clippy` to run lints against the codebase.

```yaml
on: push
name: Clippy check

# Make sure CI fails on all warnings, including Clippy lints
env:
  RUSTFLAGS: "-Dwarnings"

jobs:
  clippy_check:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v5
      - name: Run Clippy
        run: cargo clippy --all-targets --all-features
```

GitLab CI

You can add Clippy to GitLab CI by using the latest stable rust docker image, as it is shown in the `.gitlab-ci.yml` CI configuration file below,

```
# Make sure CI fails on all warnings, including Clippy lints
variables:
  RUSTFLAGS: "-Dwarnings"

clippy_check:
  image: rust:latest
  script:
    - rustup component add clippy
    - cargo clippy --all-targets --all-features
```

Travis CI

You can add Clippy to Travis CI in the same way you use it locally:

```
language: rust
rust:
  - stable
  - beta
before_script:
  - rustup component add clippy
script:
  - cargo clippy
  # if you want the build job to fail when encountering warnings, use
  - cargo clippy -- -D warnings
  # in order to also check tests and non-default crate features, use
  - cargo clippy --all-targets --all-features -- -D warnings
  - cargo test
  # etc.
```

Clippy Development

Hello fellow Rustacean! If you made it here, you're probably interested in making Clippy better by contributing to it. In that case, welcome to the project!

> *Note:* If you're just interested in using Clippy, there's nothing to see from this point onward, and you should return to one of the earlier chapters.

Getting started

If this is your first time contributing to Clippy, you should first read the Basics docs. This will explain the basics on how to get the source code and how to compile and test the code.

Additional Readings for Beginners

If a dear reader of this documentation has never taken a class on compilers and interpreters, it might be confusing as to why AST level deals with only the language's syntax. And some readers might not even understand what lexing, parsing, and AST mean.

This documentation serves by no means as a crash course on compilers or language design. And for details specifically related to Rust, the Rustc Development Guide is a far better choice to peruse.

The Syntax and AST chapter and the High-Level IR chapter are great introduction to the concepts mentioned in this chapter.

Some readers might also find the introductory chapter of Robert Nystrom's *Crafting Interpreters* a helpful overview of compiled and interpreted languages before jumping back to the Rustc guide.

Writing code

If you have done the basic setup, it's time to start hacking.

The Adding lints chapter is a walk through on how to add a new lint to Clippy. This is also interesting if you just want to fix a lint, because it also covers how to test lints and gives an overview of the bigger picture.

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. The lint categories are also described earlier in this book.

> *Note:* Some higher level things about contributing to Clippy are still covered in the `CONTRIBUTING.md` document. Some of those will be moved to the book over time, like:
>
> - Finding something to fix
> - IDE setup

- High level overview on how Clippy works
- Triage procedure

Basics for hacking on Clippy

This document explains the basics for hacking on Clippy. Besides others, this includes how to build and test Clippy. For a more in depth description on the codebase take a look at Adding Lints or Common Tools.

Get the Code

First, make sure you have checked out the latest version of Clippy. If this is your first time working on Clippy, create a fork of the repository and clone it afterwards with the following command:

```
git clone git@github.com:<your-username>/rust-clippy
```

If you've already cloned Clippy in the past, update it to the latest version:

```
# If the upstream remote has not been added yet
git remote add upstream https://github.com/rust-lang/rust-clippy
# upstream has to be the remote of the rust-lang/rust-clippy repo
git fetch upstream
# make sure that you are on the master branch
git checkout master
# rebase your master branch on the upstream master
git rebase upstream/master
# push to the master branch of your fork
git push
```

Building and Testing

You can build and test Clippy like every other Rust project:

```
cargo build  # builds Clippy
cargo test   # tests Clippy
```

Since Clippy's test suite is pretty big, there are some commands that only run a subset of Clippy's tests:

```
# only run UI tests
cargo uitest
# only run UI tests starting with `test_`
TESTNAME="test_" cargo uitest
# only run dogfood tests
cargo dev dogfood
```

If the output of a UI test differs from the expected output, you can update the reference file with:

```
cargo bless
```

For example, this is necessary if you fix a typo in an error message of a lint, or if you modify a test file to add a test case.

Note: This command may update more files than you intended. In that case only commit the files you wanted to update.

`cargo dev`

Clippy has some dev tools to make working on Clippy more convenient. These tools can be accessed through the `cargo dev` command. Available tools are listed below. To get more information about these commands, just call them with `--help`.

```
# formats the whole Clippy codebase and all tests
cargo dev fmt
# register or update lint names/groups/...
cargo dev update_lints
# create a new lint and register it
cargo dev new_lint
# deprecate a lint and attempt to remove code relating to it
cargo dev deprecate
# automatically formatting all code before each commit
cargo dev setup git-hook
# (experimental) Setup Clippy to work with RustRover
cargo dev setup intellij
# runs the `dogfood` tests
cargo dev dogfood
```

More about intellij command usage and reasons.

lintcheck

`cargo lintcheck` will build and run Clippy on a fixed set of crates and generate a log of the results. You can `git diff` the updated log against its previous version and see what impact your lint made on a small set of crates. If you add a new lint, please audit the resulting warnings and make sure there are no false positives and that the suggestions are valid.

Refer to the tools README for more details.

PR

We follow a rustc no merge-commit policy. See https://rustc-dev-guide.rust-lang.org/contributing.html#opening-a-pr.

Common Abbreviations

Abbreviation	Meaning
UB	Undefined Behavior
FP	False Positive
FN	False Negative
ICE	Internal Compiler Error
AST	Abstract Syntax Tree
MIR	Mid-Level Intermediate Representation
HIR	High-Level Intermediate Representation
TCX	Type context

This is a concise list of abbreviations that can come up during Clippy development. An extensive general list can be found in the rustc-dev-guide glossary. Always feel free to ask if an abbreviation or meaning is unclear to you.

Install from source

If you are hacking on Clippy and want to install it from source, do the following:

From the Clippy project root, run the following command to build the Clippy binaries and copy them into the toolchain directory. This will create a new toolchain called `clippy` by default, see `cargo dev setup toolchain --help` for other options.

```
cargo dev setup toolchain
```

Now you may run `cargo +clippy clippy` in any project using the new toolchain.

```
cd my-project
cargo +clippy clippy
```

...or `clippy-driver`

```
clippy-driver +clippy <filename>
```

If you no longer need the toolchain it can be uninstalled using `rustup`:

```
rustup toolchain uninstall clippy
```

DO NOT install using `cargo install --path . --force` since this will overwrite rustup proxies. That is, `~/.cargo/bin/cargo-clippy` and `~/.cargo/bin/clippy-driver` should be hard or soft links to `~/.cargo/bin/rustup`. You can repair these by running `rustup update`.

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). If you're unsure if the name you chose fits the lint, take a look at our lint naming guidelines.

Defining Our Lint

To get started, there are two ways to define our lint.

Standalone

Command: `cargo dev new_lint --name=foo_functions --pass=early --category=pedantic` (category will default to nursery if not provided)

This command will create a new file: `clippy_lints/src/foo_functions.rs`, as well as register the lint.

Specific Type

Command: `cargo dev new_lint --name=foo_functions --type=functions --category=pedantic`

This command will create a new file: `clippy_lints/src/{type}/foo_functions.rs`.

Notice how this command has a `--type` flag instead of `--pass`. Unlike a standalone definition, this lint won't be registered in the traditional sense. Instead, you will call your lint from within the type's lint pass, found in `clippy_lints/src/{type}/mod.rs`.

A "type" is just the name of a directory in `clippy_lints/src`, like `functions` in the example command. These are groupings of lints with common behaviors, so if your lint falls into one, it would be best to add it to that type.

Tests Location

Both commands will create a file: `tests/ui/foo_functions.rs`. 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:

```rust
#![allow(unused)]
#![warn(clippy::foo_functions)]

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

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

// Plain functions
fn fo() {}
fn foo() {}
//~^ foo_functions
fn food() {}

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

Note that we are adding comment annotations with the name of our lint to mark lines where we expect an error. Except for very specific situations (`//@check-pass`), at least one error marker must be present in a test file for it to be accepted.

Once we have implemented our lint we can run `TESTNAME=foo_functions cargo uibless` to generate the `.stderr` file. If our lint makes use of structured suggestions then this command will also generate the corresponding `.fixed` file.

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 by checking the `.stderr` file that gets updated on every test run.

Once we have implemented our lint running `TESTNAME=foo_functions cargo uitest` should pass on its own. When we commit our lint, we need to commit the generated `.stderr` and if applicable `.fixed` files, too. In general, you should only commit files changed by `cargo bless` for the specific lint you are creating/editing.

Note: 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 --name=foo_categories --type=cargo --category=cargo` 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 will create a `.fixed` file by running 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 bless` to automatically generate the `.fixed` file while running the tests.

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 the following from the Clippy directory:

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

To run Clippy on an existing project rather than a single file you can use

```
cargo dev lint /path/to/project
```

Or set up a rustup toolchain that points to the local Clippy binaries

```
cargo dev setup toolchain

# Then in `/path/to/project` you can run
cargo +clippy clippy
```

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;
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_lints` 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`.

`EarlyLintPass` runs before type checking and HIR lowering, while `LateLintPass` runs after these stages, providing access to type information. The `cargo dev new_lint` command defaults to the recommended `LateLintPass`, but you can specify `--pass=early` if your lint only needs AST level analysis.

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(_, _, 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 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: Msrv,
}

impl ManualStrip {
    pub fn new(conf: &'static Conf) -> Self {
        Self { msrv: conf.msrv }
    }
}
```

The project's MSRV can then be matched against the feature MSRV in the LintPass using the `Msrv::meets` method.

```
if !self.msrv.meets(cx, msrvs::STR_STRIP_PREFIX) {
    return;
}
```

Early lint passes should instead use `MsrvStack` coupled with `extract_msrv_attr!()`

Once the `msrv` is added to the lint, a relevant test case should be added to the lint's test file, `tests/ui/manual_strip.rs` in this example. It should have a case for the version below the MSRV and one with the same contents but for the MSRV version itself.

```
...

#[clippy::msrv = "1.44"]
fn msrv_1_44() {
    /* something that would trigger the lint */
}

#[clippy::msrv = "1.45"]
fn msrv_1_45() {
    /* something that would trigger the lint */
}
```

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

```
define_Conf! {
    #[lints(
        allow_attributes,
        allow_attributes_without_reason,
        ..
        <the newly added lint name>,
        ..
        unused_trait_names,
        use_self,
    )]
    msrv: Msrv = Msrv::default(),
    ...
}
```

Afterwards update the documentation for the book as described in Adding configuration to a lint.

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 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.

Print HIR lint

To implement a lint, it's helpful to first understand the internal representation that rustc uses. Clippy has the `#[clippy::dump]` attribute that prints the *High-Level Intermediate Representation (HIR)* of the item, statement, or expression that the attribute is attached to. To attach the attribute to expressions you often need to enable `#![feature(stmt_expr_attributes)]`.

Here you can find an example, just select *Tools* and run *Clippy*.

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
    /// // A short example of code that triggers the lint
    /// ```
    ///
    /// Use instead:
    /// ```rust,ignore
    /// // 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"
}
```

If the lint is in the `restriction` group because it lints things that are not necessarily “bad” but are more of a style choice, then replace the “Why is this bad?” section heading with “Why restrict this?”, to avoid writing “Why is this bad? It isn't, but ...”.

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.

Conflicting lints

There are several lints that deal with the same pattern but suggest different approaches. In other words, some lints may suggest modifications that go in the opposite direction to what some other lints already propose for the same code, creating conflicting diagnostics.

When you are creating a lint that ends up in this scenario, the following tips should be encouraged to guide classification:

- The only case where they should be in the same category is if that category is `restriction`. For example, `semicolon_inside_block` and `semicolon_outside_block`.
- For all the other cases, they should be in different categories with different levels of allowance. For example, `implicit_return` (restriction, allow) and `needless_return` (style, warn).

For lints that are in different categories, it is also recommended that at least one of them should be in the `restriction` category. The reason for this is that the `restriction` group is the only group where we don't recommend to enable the entire set, but cherry pick lints out of.

PR Checklist

Before submitting your PR make sure you followed all 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 which is searched for in:

1. The directory specified by the `CLIPPY_CONF_DIR` environment variable, or
2. The directory specified by the CARGO_MANIFEST_DIR environment variable, or
3. The current 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_config::conf` like this:

   ```
   /// 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:

      ```
      // Generated struct definition
      declare_lint_pass!(StructName => [
          LINT_NAME
      ]);

      // New manual definition struct
      pub struct StructName {}

      impl_lint_pass!(StructName => [
          LINT_NAME
      ]);
      ```

   2. Next add the configuration value and a corresponding creation method like this:

      ```
      pub struct StructName {
          configuration_ident: Type,
      }

      // ...

      impl StructName {
          pub fn new(conf: &'static Conf) -> Self {
              Self {
                  configuration_ident: conf.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:

```
// Default generated registration:
store.register_*_pass(|| box module::StructName);

// New registration with configuration value
store.register_*_pass(move || box module::StructName::new(conf));
```

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.

5. Update Lint Configuration

    Run `cargo bless --test config-metadata` to generate documentation changes for the book.

Cheat Sheet

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 (`implements_trait`, `snippet`, etc)
- Clippy diagnostics
- Let chains
- `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.

Define New Lints

The first step in the journey of a new lint is the definition and registration of the lint in Clippy's codebase. We can use the Clippy dev tools to handle this step since setting up the lint involves some boilerplate code.

Lint types

A lint type is the category of items and expressions in which your lint focuses on.

As of the writing of this documentation update, there are 11 *types* of lints besides the numerous standalone lints living under `clippy_lints/src/` :

- `cargo`
- `casts`
- `functions`
- `loops`
- `matches`
- `methods`
- `misc_early`
- `operators`
- `transmute`
- `types`
- `unit_types`

These types group together lints that share some common behaviors. For instance, `functions` groups together lints that deal with some aspects of functions in Rust, like definitions, signatures and attributes.

For more information, feel free to compare the lint files under any category with All Clippy lints or ask one of the maintainers.

Lint name

A good lint name is important, make sure to check the lint naming guidelines. Don't worry, if the lint name doesn't fit, a Clippy team member will alert you in the PR process.

We'll name our example lint that detects functions named "foo" `foo_functions` . Check the lint naming guidelines to see why this name makes sense.

Add and Register the Lint

Now that a name is chosen, we shall register `foo_functions` as a lint to the codebase. There are two ways to register a lint.

Standalone

If you believe that this new lint is a standalone lint (that doesn't belong to any specific type like `functions` or `loops`), you can run the following command in your Clippy project:

```
$ cargo dev new_lint --name=lint_name --pass=late --category=pedantic
```

There are two things to note here:

1. `--pass`: We set `--pass=late` in this command to do a late lint pass. The alternative is an `early` lint pass. We will discuss this difference in the Lint Passes chapter.
2. `--category`: If not provided, the `category` of this new lint will default to `nursery`.

The `cargo dev new_lint` command will create a new file: `clippy_lints/src/foo_functions.rs` as well as register the lint.

Overall, you should notice that the following files are modified or created:

```
$ git status
On branch foo_functions
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   CHANGELOG.md
    modified:   clippy_lints/src/lib.register_lints.rs
    modified:   clippy_lints/src/lib.register_pedantic.rs
    modified:   clippy_lints/src/lib.rs

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    clippy_lints/src/foo_functions.rs
    tests/ui/foo_functions.rs
```

Specific Type

> **Note**: Lint types are listed in the "Lint types" section

If you believe that this new lint belongs to a specific type of lints, you can run `cargo dev new_lint` with a `--type` option.

Since our `foo_functions` lint is related to function calls, one could argue that we should put it into a group of lints that detect some behaviors of functions, we can put it in the `functions` group.

Let's run the following command in your Clippy project:

```
$ cargo dev new_lint --name=foo_functions --type=functions --category=pedantic
```

This command will create, among other things, a new file: `clippy_lints/src/{type}/foo_functions.rs`. In our case, the path will be `clippy_lints/src/functions/foo_functions.rs`.

Notice how this command has a `--type` flag instead of `--pass`. Unlike a standalone definition, this lint won't be registered in the traditional sense. Instead, you will call your lint from within the type's lint pass, found in `clippy_lints/src/{type}/mod.rs`.

A *type* is just the name of a directory in `clippy_lints/src`, like `functions` in the example command. Clippy groups together some lints that share common behaviors, so if your lint falls into one, it would be best to add it to that type.

Overall, you should notice that the following files are modified or created:

```
$ git status
On branch foo_functions
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   CHANGELOG.md
    modified:   clippy_lints/src/declared_lints.rs
    modified:   clippy_lints/src/functions/mod.rs

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    clippy_lints/src/functions/foo_functions.rs
    tests/ui/foo_functions.rs
```

The `declare_clippy_lint` macro

After `cargo dev new_lint`, you should see a macro with the name `declare_clippy_lint`. It will be in the same file if you defined a standalone lint, and it will be in `mod.rs` if you defined a type-specific lint.

The macro looks something like this:

```
declare_clippy_lint! {
    /// ### What it does
    ///
    /// // Describe here what does the lint do.
    ///
    /// Triggers when detects...
    ///
    /// ### Why is this bad?
    ///
    /// // Describe why this pattern would be bad
    ///
    /// It can lead to...
    ///
    /// ### Example
    /// ```rust
    /// // example code where Clippy issues a warning
    /// ```
    /// Use instead:
    /// ```rust
    /// // example code which does not raise Clippy warning
    /// ```
    #[clippy::version = "1.70.0"] // <- In which version was this implemented, keep it
up to date!
    pub LINT_NAME, // <- The lint name IN_ALL_CAPS
    pedantic, // <- The lint group
    "default lint description" // <- A lint description, e.g. "A function has an unit
return type."
}
```

Lint registration

If we run the `cargo dev new_lint` command for a new lint, the lint will be automatically registered and there is nothing more to do.

However, sometimes we might want to declare a new lint by hand. In this case, we'd use `cargo dev update_lints` command afterwards.

When a lint is manually declared, we might need to register the lint pass manually in the `register_lints` function in `clippy_lints/src/lib.rs`:

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

As you might have guessed, where there's something late, there is something early: in Clippy there is a `register_early_pass` method as well. More on early vs. late passes in the Lint Passes chapter.

Without a call to one of `register_early_pass` or `register_late_pass`, the lint pass in question will not be run.

Testing

Developing lints for Clippy is a Test-Driven Development (TDD) process because our first task before implementing any logic for a new lint is to write some test cases.

Develop Lints with Tests

When we develop Clippy, we enter a complex and chaotic realm full of programmatic issues, stylistic errors, illogical code and non-adherence to convention. Tests are the first layer of order we can leverage to define when and where we want a new lint to trigger or not.

Moreover, writing tests first help Clippy developers to find a balance for the first iteration of and further enhancements for a lint. With test cases on our side, we will not have to worry about over-engineering a lint on its first version nor missing out some obvious edge cases of the lint. This approach empowers us to iteratively enhance each lint.

Clippy UI Tests

We use **UI tests** for testing in Clippy. These UI tests check that the output of Clippy is exactly as we expect it to be. 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 with the command `cargo bless` (seen later on).

Write Test Cases

Let us now think about some tests for our imaginary `foo_functions` lint. We start by opening the test file `tests/ui/foo_functions.rs` that was created by `cargo dev new_lint`.

Update the file with some examples to get started:

```
#![warn(clippy::foo_functions)] // < Add this, so the lint is guaranteed to be enabled
in this file

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

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

// Plain functions
fn fo() {}
fn foo() {}
//~^ foo_functions
fn food() {}

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

Without actual lint logic to emit the lint when we see a `foo` function name, this test will fail, because we expect errors at lines marked with `//~^ foo_functions`. However, we can now run the test with the following command:

```
$ TESTNAME=foo_functions cargo uitest
```

Clippy will compile and it will fail complaining it didn't receive any errors:

```
...Clippy warnings and test outputs...
error: diagnostic code `clippy::foo_functions` not found on line 8
 --> tests/ui/foo_functions.rs:9:10
  |
9 |     //~^ foo_functions
  |          ^^^^^^^^^^^^^ expected because of this pattern
  |

error: diagnostic code `clippy::foo_functions` not found on line 16
  --> tests/ui/foo_functions.rs:17:10
   |
17 |     //~^ foo_functions
   |          ^^^^^^^^^^^^^ expected because of this pattern
   |

error: diagnostic code `clippy::foo_functions` not found on line 23
  --> tests/ui/foo_functions.rs:24:6
   |
24 | //~^ foo_functions
   |      ^^^^^^^^^^^^^ expected because of this pattern
   |
```

This is normal. After all, we wrote a bunch of Rust code but we haven't really implemented any logic for Clippy to detect `foo` functions and emit a lint.

As we gradually implement our lint logic, we will keep running this UI test command. Clippy will begin outputting information that allows us to check if the output is turning into what we want it to be.

Example output

As our `foo_functions` lint is tested, the output would look something like this:

```
failures:
---- compile_test stdout ----
normalized stderr:
error: function called "foo"
  --> tests/ui/foo_functions.rs:6:12
   |
LL |     pub fn foo(&self) {}
   |            ^^^
   |
   = note: `-D clippy::foo-functions` implied by `-D warnings`
error: function called "foo"
  --> tests/ui/foo_functions.rs:13:8
   |
LL |     fn foo(&self) {}
   |        ^^^
error: function called "foo"
  --> tests/ui/foo_functions.rs:19:4
   |
LL | fn foo() {}
   |    ^^^
error: aborting due to 3 previous errors
```

Note the *failures* label at the top of the fragment, we'll get rid of it (saving this output) in the next section.

Note: You can run multiple test files by specifying a comma separated list: `TESTNAME=foo_functions,bar_methods,baz_structs`.

`cargo bless`

Once we are satisfied with the output, we need to run this command to generate or update the `.stderr` file for our lint:

```
$ TESTNAME=foo_functions cargo uibless
```

This writes the emitted lint suggestions and fixes to the `.stderr` file, with the reason for the lint, suggested fixes, and line numbers, etc.

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 bless` for the specific lint you are creating/editing.

Note: If the generated `.stderr`, and `.fixed` files are empty, they should be removed.

`toml` Tests

Some lints can be configured through a `clippy.toml` file. Those configuration values are tested in `tests/ui-toml`.

To add a new test there, create a new directory and add the files:

- `clippy.toml`: Put here the configuration value you want to test.
- `lint_name.rs`: A test file where you put the testing code, that should see a different lint behavior according to the configuration set in the `clippy.toml` file.

The potential `.stderr` and `.fixed` files can again be generated with `cargo bless`.

Cargo Lints

The process of testing is different for Cargo lints in that now we are interested in the `Cargo.toml` manifest file. In this case, we also need a minimal crate associated with that manifest. Those tests are generated in `tests/ui-cargo`.

Imagine we have a new example lint that is named `foo_categories`, we can run:

```
$ cargo dev new_lint --name=foo_categories --pass=late --category=cargo
```

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 as for other lints and prepending the `TESTNAME` variable to `cargo uitest` works for Cargo lints too.

Rustfix Tests

If the lint you are working on is making use of structured suggestions, `rustfix` will apply the suggestions from the lint to the test file code and compare that to the contents of a `.fixed` file.

Structured suggestions tell a user how to fix or re-write certain code that has been linted with `span_lint_and_sugg` .

Should `span_lint_and_sugg` be used to generate a suggestion, but not all suggestions lead to valid code, you can use the `//@no-rustfix` comment on top of the test file, to not run `rustfix` on that file.

We'll talk about suggestions more in depth in a later chapter.

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

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 from the working copy root.

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

Lint passes

Before working on the logic of a new lint, there is an important decision that every Clippy developer must make: to use `EarlyLintPass` or `LateLintPass`.

In short, the `LateLintPass` has access to type and symbol information while the `EarlyLintPass` doesn't. If you don't need access to type information, use the `EarlyLintPass`.

Let us expand on these two traits more below.

`EarlyLintPass`

If you examine the documentation on `EarlyLintPass` closely, you'll see that every method defined for this trait utilizes a `EarlyContext`. In `EarlyContext`'s documentation, it states:

> Context for lint checking of the AST, after expansion, before lowering to HIR.

Voilà. `EarlyLintPass` works only on the Abstract Syntax Tree (AST) level. And AST is generated during the lexing and parsing phase of code compilation. Therefore, it doesn't know what a symbol means or information about types, and it should be our trait choice for a new lint if the lint only deals with syntax-related issues.

While linting speed has not been a concern for Clippy, the `EarlyLintPass` is faster, and it should be your choice if you know for sure a lint does not need type information.

As a reminder, run the following command to generate boilerplate for lints that use `EarlyLintPass`:

```
$ cargo dev new_lint --name=<your_new_lint> --pass=early --category=
<your_category_choice>
```

Example for `EarlyLintPass`

Take a look at the following code:

```
let x = OurUndefinedType;
x.non_existing_method();
```

From the AST perspective, both lines are "grammatically" correct. The assignment uses a `let` and ends with a semicolon. The invocation of a method looks fine, too. As programmers, we might raise a few questions already, but the parser is okay with it. This is what we mean when we say `EarlyLintPass` deals with only syntax on the AST level.

Alternatively, think of the `foo_functions` lint we mentioned in the Define New Lints chapter.

We want the `foo_functions` lint to detect functions with `foo` as their name. Writing a lint that only checks for the name of a function means that we only work with the AST and don't have to access the type system at all (the type system is where `LateLintPass` comes into the picture).

`LateLintPass`

In contrast to `EarlyLintPass`, `LateLintPass` contains type information.

If you examine the documentation on `LateLintPass` closely, you see that every method defined in this trait utilizes a `LateContext`.

In `LateContext`'s documentation we will find methods that deal with type-checking, which do not exist in `EarlyContext`, such as:

- `maybe_typeck_results`
- `typeck_results`

Example for `LateLintPass`

Let us take a look with the following example:

```
let x = OurUndefinedType;
x.non_existing_method();
```

These two lines of code are syntactically correct code from the perspective of the AST. We have an assignment and invoke a method on the variable that is of a type. Grammatically, everything is in order for the parser.

However, going down a level and looking at the type information, the compiler will notice that both `OurUndefinedType` and `non_existing_method()` **are undefined**.

As Clippy developers, to access such type information, we must implement `LateLintPass` on our lint. When you browse through Clippy's lints, you will notice that almost every lint is implemented in a `LateLintPass`, specifically because we often need to check not only for syntactic issues but also type information.

Another limitation of the `EarlyLintPass` is that the nodes are only identified by their position in the AST. This means that you can't just get an `id` and request a certain node. For most lints that is fine, but we have some lints that require the inspection of other nodes, which is easier at the HIR level. In these cases, `LateLintPass` is the better choice.

As a reminder, run the following command to generate boilerplate for lints that use `LateLintPass`:

```
$ cargo dev new_lint --name=<your_new_lint> --pass=late --category=
<your_category_choice>
```

Emitting a lint

Once we have defined a lint, written UI tests and chosen the lint pass for the lint, we can begin the implementation of the lint logic so that we can emit it and gradually work towards a lint that behaves as expected.

Note that we will not go into concrete implementation of a lint logic in this chapter. We will go into details in later chapters as well as in two examples of real Clippy lints.

To emit a lint, we must implement a pass (see Lint Passes) for the lint that we have declared. In this example we'll implement a "late" lint, so take a look at the LateLintPass documentation, which provides an abundance of methods that we can implement for our lint.

```
pub trait LateLintPass<'tcx>: LintPass {
    // Trait methods
}
```

By far the most common method used for Clippy lints is `check_expr` method, this is because Rust is an expression language and, more often than not, the lint we want to work on must examine expressions.

> *Note:* If you don't fully understand what expressions are in Rust, take a look at the official documentation on expressions

---

Other common ones include the `check_fn` method and the `check_item` method.

Emitting a lint

Inside the trait method that we implement, we can write down the lint logic and emit the lint with suggestions.

Clippy's diagnostics provides quite a few diagnostic functions that we can use to emit lints. Take a look at the documentation to pick one that suits your lint's needs the best. Some common ones you will encounter in the Clippy repository includes:

- `span_lint` : Emits a lint without providing any other information
- `span_lint_and_note` : Emits a lint and adds a note
- `span_lint_and_help` : Emits a lint and provides a helpful message
- `span_lint_and_sugg` : Emits a lint and provides a suggestion to fix the code
- `span_lint_and_then` : Like `span_lint` , but allows for a lot of output customization.

```
impl<'tcx> LateLintPass<'tcx> for LintName {
    fn check_expr(&mut self, cx: &LateContext<'tcx>, expr: &'tcx Expr<'_>)  {
        // Imagine that `some_lint_expr_logic` checks for requirements for emitting
the lint
        if some_lint_expr_logic(expr) {
            span_lint_and_help(
                cx, // < The context
                LINT_NAME, // < The name of the lint in ALL CAPS
                expr.span, // < The span to lint
                "message on why the lint is emitted",
                None, // < An optional help span (to highlight something in the lint)
                "message that provides a helpful suggestion",
            );
        }
    }
}
```

Note: The message should be matter of fact and avoid capitalization and punctuation. If multiple sentences are needed, the messages should probably be split up into an error + a help / note / suggestion message.

Suggestions: Automatic fixes

Some lints know what to change in order to fix the code. For example, the lint `range_plus_one` warns for ranges where the user wrote `x..y + 1` instead of using an inclusive range (`x..=y`). The fix to this code would be changing the `x..y + 1` expression to `x..=y` . **This is where suggestions come in**.

A suggestion is a change that the lint provides to fix the issue it is linting. The output looks something like this (from the example earlier):

```
error: an inclusive range would be more readable
  --> tests/ui/range_plus_minus_one.rs:37:14
   |
LL |     for _ in 1..1 + 1 {}
   |              ^^^^^^^ help: use: `1..=1`
```

Not all suggestions are always right, some of them require human supervision, that's why we have Applicability.

Applicability indicates confidence in the correctness of the suggestion, some are always right (`Applicability::MachineApplicable`), but we use `Applicability::MaybeIncorrect` and others when talking about a suggestion that may be incorrect.

Example

The same lint `LINT_NAME` but that emits a suggestion would look something like this:

```
impl<'tcx> LateLintPass<'tcx> for LintName {
    fn check_expr(&mut self, cx: &LateContext<'tcx>, expr: &'tcx Expr<'_>)  {
        // Imagine that `some_lint_expr_logic` checks for requirements for emitting
the lint
        if some_lint_expr_logic(expr) {
            span_lint_and_sugg( // < Note this change
                cx,
                LINT_NAME,
                span,
                "message on why the lint is emitted",
                "use",
                format!("foo + {} * bar", snippet(cx, expr.span, "<default>")), // <
Suggestion
                Applicability::MachineApplicable,
            );
        }
    }
}
```

Suggestions generally use the `format!` macro to interpolate the old values with the new ones. To get code snippets, use one of the `snippet*` functions from `clippy_utils::source`.

How to choose between notes, help messages and suggestions

Notes are presented separately from the main lint message, they provide useful information that the user needs to understand why the lint was activated. They are the most helpful when attached to a span.

Examples:

Notes

```
error: calls to `std::mem::forget` with a reference instead of an owned value.
Forgetting a reference does nothing.
  --> tests/ui/drop_forget_ref.rs:10:5
   |
10 |     forget(&SomeStruct);
   |     ^^^^^^^^^^^^^^^^^^^
   |
   = note: `-D clippy::forget-ref` implied by `-D warnings`
note: argument has type &SomeStruct
  --> tests/ui/drop_forget_ref.rs:10:12
   |
10 |     forget(&SomeStruct);
   |            ^^^^^^^^^^^
```

Help Messages

Help messages are specifically to help the user. These are used in situation where you can't provide a specific machine applicable suggestion. They can also be attached to a span.

Example:

```
error: constant division of 0.0 with 0.0 will always result in NaN
  --> tests/ui/zero_div_zero.rs:6:25
   |
6  |     let other_f64_nan = 0.0f64 / 0.0;
   |                         ^^^^^^^^^^^^
   |
   = help: consider using `f64::NAN` if you would like a constant representing NaN
```

Suggestions

Suggestions are the most helpful, they are changes to the source code to fix the error. The magic in suggestions is that tools like `rustfix` can detect them and automatically fix your code.

Example:

```
error: This `.fold` can be more succinctly expressed as `.any`
--> tests/ui/methods.rs:390:13
    |
390 |     let _ = (0..3).fold(false, |acc, x| acc || x > 2);
    |                    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ help: try: `.any(|x| x > 2)`
    |
```

Snippets

Snippets are pieces of the source code (as a string), they are extracted generally using the `snippet` function.

For example, if you want to know how an item looks (and you know the item's span), you could use `snippet(cx, span, "..")`.

Final: Run UI Tests to Emit the Lint

Now, if we run our UI test, we should see that Clippy now produces output that contains the lint message we designed.

The next step is to implement the logic properly, which is a detail that we will cover in the next chapters.

Type Checking

When we work on a new lint or improve an existing lint, we might want to retrieve the type `Ty` of an expression `Expr` for a variety of reasons. This can be achieved by utilizing the `LateContext` that is available for `LateLintPass`.

`LateContext` and `TypeckResults`

The lint context `LateContext` and `TypeckResults` (returned by `LateContext::typeck_results`) are the two most useful data structures in `LateLintPass`. They allow us to jump to type definitions and other compilation stages such as HIR.

> Note: `LateContext.typeck_results`'s return value is `TypeckResults` and is created in the type checking step, it includes useful information such as types of expressions, ways to resolve methods and so on.

`TypeckResults` contains useful methods such as `expr_ty`, which gives us access to the underlying structure `Ty` of a given expression.

```
pub fn expr_ty(&self, expr: &Expr<'_>) -> Ty<'tcx>
```

As a side note, besides `expr_ty`, `TypeckResults` contains a `pat_ty()` method that is useful for retrieving a type from a pattern.

Ty

`Ty` struct contains the type information of an expression. Let's take a look at `rustc_middle`'s `Ty` struct to examine this struct:

```
pub struct Ty<'tcx>(Interned<'tcx, WithStableHash<TyS<'tcx>>>);
```

At a first glance, this struct looks quite esoteric. But at a closer look, we will see that this struct contains many useful methods for type checking.

For instance, `is_char` checks if the given `Ty` struct corresponds to the primitive character type.

`is_*` Usage

In some scenarios, all we need to do is check if the `Ty` of an expression is a specific type, such as `char` type, so we could write the following:

```
impl LateLintPass<'_> for MyStructLint {
    fn check_expr(&mut self, cx: &LateContext<'_>, expr: &Expr<'_>) {
        // Get type of `expr`
        let ty = cx.typeck_results().expr_ty(expr);

        // Check if the `Ty` of this expression is of character type
        if ty.is_char() {
            println!("Our expression is a char!");
        }
    }
}
```

Furthermore, if we examine the source code for `is_char`, we find something very interesting:

```
#[inline]
pub fn is_char(self) -> bool {
    matches!(self.kind(), Char)
}
```

Indeed, we just discovered `Ty`'s `kind()` method, which provides us with `TyKind` of a `Ty`.

TyKind

`TyKind` defines the kinds of types in Rust's type system. Peeking into `TyKind` documentation, we will see that it is an enum of over 25 variants, including items such as `Bool`, `Int`, `Ref`, etc.

`kind` Usage

The `TyKind` of `Ty` can be returned by calling `Ty.kind()` method. We often use this method to perform pattern matching in Clippy.

For instance, if we want to check for a `struct`, we could examine if the `ty.kind` corresponds to an `Adt` (algebraic data type) and if its `AdtDef` is a struct:

```
impl LateLintPass<'_> for MyStructLint {
    fn check_expr(&mut self, cx: &LateContext<'_>, expr: &Expr<'_>) {
        // Get type of `expr`
        let ty = cx.typeck_results().expr_ty(expr);
        // Match its kind to enter the type
        match ty.kind() {
            ty::Adt(adt_def, _) if adt_def.is_struct() => println!("Our `expr` is a
struct!"),
            _ => ()
        }
    }
}
```

`hir::Ty` and `ty::Ty`

We've been talking about `ty::Ty` this whole time without addressing `hir::Ty`, but the latter is also important to understand.

`hir::Ty` would represent *what* the user wrote, while `ty::Ty` is how the compiler sees the type and has more information. Example:

```
fn foo(x: u32) -> u32 { x }
```

Here the HIR sees the types without "thinking" about them, it knows that the function takes an `u32` and returns an `u32`. As far as `hir::Ty` is concerned those might be different types. But at the `ty::Ty` level the compiler understands that they're the same type, in-depth lifetimes, etc...

To get from a `hir::Ty` to a `ty::Ty`, you can use the `lower_ty` function outside of bodies or the `TypeckResults::node_type()` method inside of bodies.

Warning: Don't use `lower_ty` inside of bodies, because this can cause ICEs.

Creating Types programmatically

A common usecase for creating types programmatically is when we want to check if a type implements a trait (see Trait Checking).

Here's an example of how to create a `Ty` for a slice of `u8`, i.e. `[u8]`

```
use rustc_middle::ty::Ty;
// assume we have access to a LateContext
let ty = Ty::new_slice(cx.tcx, Ty::new_u8());
```

In general, we rely on `Ty::new_*` methods. These methods define the basic building-blocks that the type-system and trait-system use to define and understand the written code.

Useful Links

Below are some useful links to further explore the concepts covered in this chapter:

- Stages of compilation
- Diagnostic items
- Type checking
- Ty module

Trait Checking

Besides type checking, we might want to examine if a specific type `Ty` implements certain trait when implementing a lint. There are three approaches to achieve this, depending on if the target trait that we want to examine has a diagnostic item, lang item, or neither.

Using Diagnostic Items

As explained in the Rust Compiler Development Guide, diagnostic items are introduced for identifying types via Symbols.

For instance, if we want to examine whether an expression implements the `Iterator` trait, we could simply write the following code, providing the `LateContext` (`cx`), our expression at hand, and the symbol of the trait in question:

```
use clippy_utils::sym;
use clippy_utils::ty::implements_trait;
use rustc_hir::Expr;
use rustc_lint::{LateContext, LateLintPass};

impl LateLintPass<'_> for CheckIteratorTraitLint {
    fn check_expr(&mut self, cx: &LateContext<'_>, expr: &Expr<'_>) {
        let implements_iterator = (cx.tcx.get_diagnostic_item(sym::Iterator))
            .is_some_and(|id| implements_trait(cx, cx.typeck_results().expr_ty(expr),
id, &[]));
        if implements_iterator {
            // [...]
        }

    }
}
```

Note: Refer to this index for all the defined `Symbol` s.

Using Lang Items

Besides diagnostic items, we can also use `lang_items` . Take a look at the documentation to find that `LanguageItems` contains all language items defined in the compiler.

Using one of its `*_trait` method, we could obtain the DefId of any specific item, such as `Clone` , `Copy` , `Drop` , `Eq` , which are familiar to many Rustaceans.

For instance, if we want to examine whether an expression `expr` implements `Drop` trait, we could access `LanguageItems` via our `LateContext` 's TyCtxt, which provides a `lang_items` method that will return the id of `Drop` trait to us. Then, by calling Clippy utils function `implements_trait` we can check that the `Ty` of the `expr` implements the trait:

```
use clippy_utils::ty::implements_trait;
use rustc_hir::Expr;
use rustc_lint::{LateContext, LateLintPass};

impl LateLintPass<'_> for CheckDropTraitLint {
    fn check_expr(&mut self, cx: &LateContext<'_>, expr: &Expr<'_>) {
        let ty = cx.typeck_results().expr_ty(expr);
        if cx.tcx.lang_items()
            .drop_trait()
            .map_or(false, |id| implements_trait(cx, ty, id, &[])) {
                println!("`expr` implements `Drop` trait!");
            }
    }
}
```

Using Type Path

If neither diagnostic item nor a language item is available, we can use `clippy_utils::paths` to determine get a trait's `DefId`.

> **Note**: This approach should be avoided if possible, the best thing to do would be to make a PR to `rust-lang/rust` adding a diagnostic item.

Below, we check if the given `expr` implements `core::iter::Step`:

```
use clippy_utils::paths;
use clippy_utils::ty::implements_trait;
use rustc_hir::Expr;
use rustc_lint::{LateContext, LateLintPass};

impl LateLintPass<'_> for CheckIterStep {
    fn check_expr(&mut self, cx: &LateContext<'_>, expr: &Expr<'_>) {
        let ty = cx.typeck_results().expr_ty(expr);
        if let Some(trait_def_id) = paths::ITER_STEP.first(cx)
            && implements_trait(cx, ty, trait_def_id, &[])
        {
            println!("`expr` implements the `core::iter::Step` trait!");
        }
    }
}
```

Creating Types Programmatically

Traits are often generic over a type parameter, e.g. `Borrow<T>` is generic over `T`. Rust allows us to implement a trait for a specific type. For example, we can implement `Borrow<[u8]>` for a hypothetical type `Foo`. Let's suppose that we would like to find whether our type actually implements `Borrow<[u8]>`.

To do so, we can use the same `implements_trait` function as above, and supply a type parameter that represents `[u8]`. Since `[u8]` is a specialization of `[T]`, we can use the `Ty::new_slice` method to create a type that represents `[T]` and supply `u8` as a type parameter. To create a `ty::Ty`

programmatically, we rely on `Ty::new_*` methods. These methods create a `TyKind` and then wrap it in a `Ty` struct. This means we have access to all the primitive types, such as `Ty::new_char`, `Ty::new_bool`, `Ty::new_int`, etc. We can also create more complex types, such as slices, tuples, and references out of these basic building blocks.

For trait checking, it is not enough to create the types, we need to convert them into GenericArg. In rustc, a generic is an entity that the compiler understands and has three kinds, type, const and lifetime. By calling `.into()` on a constructed Ty, we wrap the type into a generic which can then be used by the query system to decide whether the specialized trait is implemented.

The following code demonstrates how to do this:

```
use rustc_middle::ty::Ty;
use clippy_utils::sym;
use clippy_utils::ty::implements_trait;

let ty = todo!("Get the `Foo` type to check for a trait implementation");
let borrow_id = cx.tcx.get_diagnostic_item(sym::Borrow).unwrap(); // avoid unwrap in
real code
let slice_of_bytes_t = Ty::new_slice(cx.tcx, cx.tcx.types.u8);
let generic_param = slice_of_bytes_t.into();
if implements_trait(cx, ty, borrow_id, &[generic_param]) {
    todo!("Rest of lint implementation")
}
```

In essence, the Ty struct allows us to create types programmatically in a representation that can be used by the compiler and the query engine. We then use the `rustc_middle::Ty` of the type we are interested in, and query the compiler to see if it indeed implements the trait we are interested in.

Method Checking

In some scenarios we might want to check for methods when developing a lint. There are two kinds of questions that we might be curious about:

- Invocation: Does an expression call a specific method?
- Definition: Does an `impl` define a method?

Checking if an `expr` is calling a specific method

Suppose we have an `expr`, we can check whether it calls a specific method, e.g. `our_fancy_method`, by performing a pattern match on the `ExprKind` that we can access from `expr.kind`:

```
use rustc_hir as hir;
use rustc_lint::{LateContext, LateLintPass};
use clippy_utils::res::{MaybeDef, MaybeTypeckRes};
use clippy_utils::sym;

impl<'tcx> LateLintPass<'tcx> for OurFancyMethodLint {
    fn check_expr(&mut self, cx: &LateContext<'tcx>, expr: &'tcx hir::Expr<'_>) {
        // Check our expr is calling a method with pattern matching
        if let hir::ExprKind::MethodCall(path, _, [self_arg, ..], _) = &expr.kind
            // Check if the name of this method is `our_fancy_method`
            && path.ident.name == sym::our_fancy_method
            // We can check the type of the self argument whenever necessary.
            // (It's necessary if we want to check that method is specifically
belonging to a specific trait,
            // for example, a `map` method could belong to user-defined trait instead
of to `Iterator`)
            // See the next section for more information.
            && cx.ty_based_def(self_arg).opt_parent(cx).is_diag_item(cx,
sym::OurFancyTrait)
        {
            println!("`expr` is a method call for `our_fancy_method`");
        }
    }
}
```

Take a closer look at the `ExprKind` enum variant `MethodCall` for more information on the pattern matching. As mentioned in Define Lints, the `methods` lint type is full of pattern matching with `MethodCall` in case the reader wishes to explore more.

New symbols such as `our_fancy_method` need to be added to the `clippy_utils::sym` module. This module extends the list of symbols already provided by the compiler crates in `rustc_span::sym`.

Checking if a `impl` block implements a method

While sometimes we want to check whether a method is being called or not, other times we want to know if our `Ty` defines a method.

To check if our `impl` block defines a method `our_fancy_method`, we will utilize the `check_impl_item` method that is available in our beloved `LateLintPass` (for more information,

refer to the "Lint Passes" chapter in the Clippy book). This method provides us with an `ImplItem` struct, which represents anything within an `impl` block.

Let us take a look at how we might check for the implementation of `our_fancy_method` on a type:

```
use clippy_utils::{return_ty, sym};
use clippy_utils::res::MaybeDef;
use rustc_hir::{ImplItem, ImplItemKind};
use rustc_lint::{LateContext, LateLintPass};

impl<'tcx> LateLintPass<'tcx> for MyTypeImpl {
    fn check_impl_item(&mut self, cx: &LateContext<'tcx>, impl_item: &'tcx
ImplItem<'_>) {
        // Check if item is a method/function
        if let ImplItemKind::Fn(ref signature, _) = impl_item.kind
            // Check the method is named `our_fancy_method`
            && impl_item.ident.name.as_str() == "our_fancy_method"
            // We can also check it has a parameter `self`
            && signature.decl.implicit_self.has_implicit_self()
            // We can go even further and even check if its return type is `String`
            && return_ty(cx, impl_item.hir_id).is_diag_item(cx, sym::String)
        {
            println!("`our_fancy_method` is implemented!");
        }
    }
}
```

Dealing with macros and expansions

Sometimes we might encounter Rust macro expansions while working with Clippy. While macro expansions are not as dramatic and profound as the expansion of our universe, they can certainly bring chaos to the orderly world of code and logic.

The general rule of thumb is that we should ignore code with macro expansions when working with Clippy because the code can be dynamic in ways that are difficult or impossible for us to foresee.

False Positives

What exactly do we mean by *dynamic in ways that are difficult to foresee*?

Macros are expanded in the `EarlyLintPass` level, so the Abstract Syntax Tree (AST) is generated in place of macros. This means the code which we work with in Clippy is already expanded.

If we wrote a new lint, there is a possibility that the lint is triggered in macro-generated code. Since this expanded macro code is not written by the macro's user but really by the macro's author, the user cannot and should not be responsible for fixing the issue that triggers the lint.

Besides, a Span in a macro can be changed by the macro author. Therefore, any lint check related to lines or columns should be avoided since they might be changed at any time and become unreliable or incorrect information.

Because of these unforeseeable or unstable behaviors, macro expansion should often not be regarded as a part of the stable API. This is also why most lints check if they are inside a macro or not before emitting suggestions to the end user to avoid false positives.

How to Work with Macros

Several functions are available for working with macros.

The `Span::from_expansion` method

We could utilize a `span` 's `from_expansion` method, which detects if the `span` is from a macro expansion / desugaring. This is a very common first step in a lint:

```
if expr.span.from_expansion() {
    // We most likely want to ignore it.
    return;
}
```

`Span::ctxt` method

The `span` 's context, given by the method `ctxt` and returning SyntaxContext, represents if the span is from a macro expansion and, if it is, which macro call expanded this span.

Sometimes, it is useful to check if the context of two spans are equal. For instance, suppose we have the following line of code that would expand into `1 + 0`:

```
// The following code expands to `1 + 0` for both `EarlyLintPass` and `LateLintPass`
1 + mac!()
```

Assuming that we'd collect the `1` expression as a variable `left` and the `0` / `mac!()` expression as a variable `right`, we can simply compare their contexts. If the context is different, then we most likely are dealing with a macro expansion and should just ignore it:

```
if left.span.ctxt() != right.span.ctxt() {
    // The code author most likely cannot modify this expression
    return;
}
```

Note: Code that is not from expansion is in the "root" context. So any spans whose `from_expansion` returns `false` can be assumed to have the same context. Because of this, using `span.from_expansion()` is often sufficient.

Going a bit deeper, in a simple expression such as `a == b`, `a` and `b` have the same context. However, in a `macro_rules!` with `a == $b`, `$b` is expanded to an expression that contains a different context from `a`.

Take a look at the following macro `m`:

```
macro_rules! m {
    ($a:expr, $b:expr) => {
        if $a.is_some() {
            $b;
        }
    }
}

let x: Option<u32> = Some(42);
m!(x, x.unwrap());
```

If the `m!(x, x.unwrap());` line is expanded, we would get two expanded expressions:

- `x.is_some()` (from the `$a.is_some()` line in the `m` macro)
- `x.unwrap()` (corresponding to `$b` in the `m` macro)

Suppose `x.is_some()` expression's span is associated with the `x_is_some_span` variable and `x.unwrap()` expression's span is associated with `x_unwrap_span` variable, we could assume that these two spans do not share the same context:

```
// x.is_some() is from inside the macro
// x.unwrap() is from outside the macro
assert_ne!(x_is_some_span.ctxt(), x_unwrap_span.ctxt());
```

The `in_external_macro` function

`Span` provides a method (`in_external_macro`) that can detect if the given span is from a macro

defined in a foreign crate.

Therefore, if we really want a new lint to work with macro-generated code, this is the next line of defense to avoid macros not defined inside the current crate since it is unfair to the user if Clippy lints code which the user cannot change.

For example, assume we have the following code that is being examined by Clippy:

```
#[macro_use]
extern crate a_foreign_crate_with_macros;

// `foo` macro is defined in `a_foreign_crate_with_macros`
foo!("bar");
```

Also assume that we get the corresponding variable `foo_span` for the `foo` macro call, we could decide not to lint if `in_external_macro` results in `true` (note that `cx` can be `EarlyContext` or `LateContext`):

```
if foo_span.in_external_macro(cx.sess().source_map()) {
    // We should ignore macro from a foreign crate.
    return;
}
```

The `is_from_proc_macro` function

A common point of confusion is the existence of `is_from_proc_macro` and how it differs from the other `in_external_macro` / `from_expansion` functions.

While `in_external_macro` and `from_expansion` both work perfectly fine for detecting expanded code from *declarative* macros (i.e. `macro_rules!` and macros 2.0), detecting *proc macro*-generated code is a bit more tricky, as proc macros can (and often do) freely manipulate the span of returned tokens.

In practice, this often happens through the use of `quote::quote_spanned!` with a span from the input tokens.

In those cases, there is no *reliable* way for the compiler (and tools like Clippy) to distinguish code that comes from such a proc macro from code that the user wrote directly, and `in_external_macro` will return `false`.

This is usually not an issue for the compiler and actually helps proc macro authors create better error messages, as it allows associating parts of the expansion with parts of the macro input and lets the compiler point the user to the relevant code in case of a compile error.

However, for Clippy this is inconvenient, because most of the time *we don't* want to lint proc macro-generated code and this makes it impossible to tell what is and isn't proc macro code.

> NOTE: this is specifically only an issue when a proc macro explicitly sets the span to that of an **input span**.
>
> For example, other common ways of creating `TokenStream`s, such as `"fn foo() {...}".parse::<TokenStream>()`, sets each token's span to `Span::call_site()`, which already marks the span as coming from a proc macro and the usual span methods have no problem

detecting that as a macro span.

As such, Clippy has its own `is_from_proc_macro` function which tries to *approximate* whether a span comes from a proc macro, by checking whether the source text at the given span lines up with the given AST node.

This function is typically used in combination with the other mentioned macro span functions, but is usually called much later into the condition chain as it's a bit heavier than most other conditions, so that the other cheaper conditions can fail faster. For example, the `borrow_deref_ref` lint:

```
impl<'tcx> LateLintPass<'tcx> for BorrowDerefRef {
    fn check_expr(&mut self, cx: &LateContext<'tcx>, e: &rustc_hir::Expr<'tcx>) {
        if let ... = ...
            && ...
            && !e.span.from_expansion()
            && ...
            && ...
            && !is_from_proc_macro(cx, e)
            && ...
        {
            ...
        }
    }
}
```

Testing lints with macro expansions

To test that all of these cases are handled correctly in your lint, we have a helper auxiliary crate that exposes various macros, used by tests like so:

```
//@aux-build:proc_macros.rs

extern crate proc_macros;

fn main() {
    proc_macros::external!{ code_that_should_trigger_your_lint }
    proc_macros::with_span!{ span code_that_should_trigger_your_lint }
}
```

This exercises two cases:

- `proc_macros::external!` is a simple proc macro that echos the input tokens back but with a macro span: this represents the usual, common case where an external macro expands to code that your lint would trigger, and is correctly handled by `in_external_macro` and `Span::from_expansion`.
- `proc_macros::with_span!` echos back the input tokens starting from the second token with the span of the first token: this is where the other functions will fail and `is_from_proc_macro` is needed

Common tools for writing lints

You may need following tooltips to catch up with common operations.

Useful Rustc dev guide links:

- Stages of compilation
- Diagnostic items
- Type checking
- Ty module

Retrieving the type of expression

Sometimes you may want to retrieve the type `Ty` of an expression `Expr`, for example to answer following questions:

- which type does this expression correspond to (using its `TyKind`)?
- is it a sized type?
- is it a primitive type?
- does it implement a trait?

This operation is performed using the `expr_ty()` method from the `TypeckResults` struct, that gives you access to the underlying structure `Ty`.

Example of use:

```
impl LateLintPass<'_> for MyStructLint {
    fn check_expr(&mut self, cx: &LateContext<'_>, expr: &Expr<'_>) {
        // Get type of `expr`
        let ty = cx.typeck_results().expr_ty(expr);
        // Match its kind to enter its type
        match ty.kind() {
            ty::Adt(adt_def, _) if adt_def.is_struct() => println!("Our `expr` is a
struct!"),
            _ => ()
        }
    }
}
```

Similarly, in `TypeckResults` methods, you have the `pat_ty()` method to retrieve a type from a pattern.

Two noticeable items here:

- `cx` is the lint context `LateContext`. The two most useful data structures in this context are

`tcx` and the `TypeckResults` returned by `LateContext::typeck_results`, allowing us to jump to type definitions and other compilation stages such as HIR.
- `typeck_results`'s return value is `TypeckResults` and is created by type checking step, it includes useful information such as types of expressions, ways to resolve methods and so on.

Checking if an expr is calling a specific method

Starting with an `expr`, you can check whether it is calling a specific method `some_method`:

```
impl<'tcx> LateLintPass<'tcx> for MyStructLint {
    fn check_expr(&mut self, cx: &LateContext<'tcx>, expr: &'tcx hir::Expr<'_>) {
        // Check our expr is calling a method
        if let hir::ExprKind::MethodCall(path, _, _self_arg, ..) = &expr.kind
            // Check the name of this method is `some_method`
            && path.ident.name == sym::some_method
            // Optionally, check the type of the self argument.
            // - See "Checking for a specific type"
        {
                // ...
        }
    }
}
```

Checking for a specific type

There are three ways to check if an expression type is a specific type we want to check for. All of these methods only check for the base type, generic arguments have to be checked separately.

```
use clippy_utils::{paths, sym};
use clippy_utils::res::MaybeDef;
use rustc_hir::LangItem;

impl LateLintPass<'_> for MyStructLint {
    fn check_expr(&mut self, cx: &LateContext<'_>, expr: &Expr<'_>) {
        // Getting the expression type
        let ty = cx.typeck_results().expr_ty(expr);

        // 1. Using diagnostic items
        // The last argument is the diagnostic item to check for
        if ty.is_diag_item(cx, sym::Option) {
            // The type is an `Option`
        }

        // 2. Using lang items
        if ty.is_lang_item(cx, LangItem::RangeFull) {
            // The type is a full range like `.drain(..)`
        }

        // 3. Using the type path
        // This method should be avoided if possible
        if paths::RESULT.matches_ty(cx, ty) {
            // The type is a `core::result::Result`
        }
    }
}
```

Prefer using diagnostic items and lang items where possible.

Checking if a type implements a specific trait

There are three ways to do this, depending on if the target trait has a diagnostic item, lang item or neither.

```
use clippy_utils::sym;
use clippy_utils::ty::implements_trait;

impl LateLintPass<'_> for MyStructLint {
    fn check_expr(&mut self, cx: &LateContext<'_>, expr: &Expr<'_>) {

        // 1. Get the `DefId` of the trait.
        // via lang items
        let trait_id = cx.tcx.lang_items().drop_trait();
        // via diagnostic items
        let trait_id = cx.tcx.get_diagnostic_item(sym::Eq);

        // 2. Check for the trait implementation via the `implements_trait` util.
        let ty = cx.typeck_results().expr_ty(expr);
        if trait_id.is_some_and(|id| implements_trait(cx, ty, id, &[])) {
            // `ty` implements the trait.
        }

        // 3. If the trait requires additional generic arguments
        let trait_id = cx.tcx.lang_items().eq_trait();
        if trait_id.is_some_and(|id| implements_trait(cx, ty, id, &[ty])) {
            // `ty` implements `PartialEq<Self>`
        }
    }
}
```

Prefer using diagnostic and lang items, if the target trait has one.

We access lang items through the type context `tcx`. `tcx` is of type `TyCtxt` and is defined in the `rustc_middle` crate. A list of defined paths for Clippy can be found in paths.rs

Checking if a type defines a specific method

To check if our type defines a method called `some_method`:

```
use clippy_utils::ty::is_type_lang_item;
use clippy_utils::{sym, return_ty};

impl<'tcx> LateLintPass<'tcx> for MyTypeImpl {
    fn check_impl_item(&mut self, cx: &LateContext<'tcx>, impl_item: &'tcx
ImplItem<'_>) {
        // Check if item is a method/function
        if let ImplItemKind::Fn(ref signature, _) = impl_item.kind
            // Check the method is named `some_method`
            //
            // Add `some_method` to `clippy_utils::sym` if it's not already there
            && impl_item.ident.name == sym::some_method
            // We can also check it has a parameter `self`
            && signature.decl.implicit_self.has_implicit_self()
            // We can go further and even check if its return type is `String`
            && return_ty(cx, impl_item.hir_id).is_lang_item(cx, LangItem::String)
        {
            // ...
        }
    }
}
```

Dealing with macros and expansions

Keep in mind that macros are already expanded and desugaring is already applied to the code representation that you are working with in Clippy. This unfortunately causes a lot of false positives because macro expansions are "invisible" unless you actively check for them. Generally speaking, code with macro expansions should just be ignored by Clippy because that code can be dynamic in ways that are difficult or impossible to see. Use the following functions to deal with macros:

- `span.from_expansion()` : detects if a span is from macro expansion or desugaring. Checking this is a common first step in a lint.

  ```
  if expr.span.from_expansion() {
      // just forget it
      return;
  }
  ```

- `span.ctxt()` : the span's context represents whether it is from expansion, and if so, which macro call expanded it. It is sometimes useful to check if the context of two spans are equal.

  ```
  // expands to `1 + 0`, but don't lint
  1 + mac!()
  ```

  ```
  if left.span.ctxt() != right.span.ctxt() {
      // the coder most likely cannot modify this expression
      return;
  }
  ```

Note: Code that is not from expansion is in the "root" context. So any spans where `from_expansion` returns `true` can be assumed to have the same context. And so just

using `span.from_expansion()` is often good enough.

- `span.in_external_macro(sm)` : detect if the given span is from a macro defined in a foreign crate. If you want the lint to work with macro-generated code, this is the next line of defense to avoid macros not defined in the current crate. It doesn't make sense to lint code that the coder can't change.

  You may want to use it for example to not start linting in macros from other crates

  ```
  use a_crate_with_macros::foo;

  // `foo` is defined in `a_crate_with_macros`
  foo!("bar");

  // if we lint the `match` of `foo` call and test its span
  assert_eq!(match_span.in_external_macro(cx.sess().source_map()), true);
  ```

- `span.ctxt()` : the span's context represents whether it is from expansion, and if so, what expanded it

  One thing `SpanContext` is useful for is to check if two spans are in the same context. For example, in `a == b`, `a` and `b` have the same context. In a `macro_rules!` with `a == $b`, `$b` is expanded to some expression with a different context from `a`.

  ```
  macro_rules! m {
      ($a:expr, $b:expr) => {
          if $a.is_some() {
              $b;
          }
      }
  }

  let x: Option<u32> = Some(42);
  m!(x, x.unwrap());

  // These spans are not from the same context
  // x.is_some() is from inside the macro
  // x.unwrap() is from outside the macro
  assert_eq!(x_is_some_span.ctxt(), x_unwrap_span.ctxt());
  ```

Infrastructure

In order to deploy Clippy over `rustup`, some infrastructure is necessary. This chapter describes the different parts of the Clippy infrastructure that need to be maintained to make this possible.

The most important part is the sync between the `rust-lang/rust` repository and the Clippy repository that takes place every two weeks. This process is described in the Syncing changes between Clippy and `rust-lang/rust` section.

A new Clippy release is done together with every Rust release, so every six weeks. The release process is described in the Release a new Clippy Version section. During a release cycle a changelog entry for the next release has to be written. The format of that and how to do that is documented in the Changelog Update section.

Note: The Clippy CI should also be described in this chapter, but for now is left as a TODO.

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 the rustc-dev-guide.

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` (making 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 1,000. 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` .

> Note: The following sections assume that you have set up remotes following the instructions in defining remotes.

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

Here is a TL;DR version of the sync process (all 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:

   ```
   # Be sure to either use a net-new branch, e.g. `rustup`, or delete the branch
   beforehand
   # because changes cannot be fast forwarded and you have to run this command again.
   git subtree push -P src/tools/clippy clippy-local rustup
   ```

   > *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 upstream  # assuming upstream is the rust-lang/rust remote
   git switch rustup
   git merge upstream/master --no-ff
   ```

   > Note: This is one of the few instances where a merge commit is allowed in a PR.

4. Bump the nightly version in the Clippy repository by running these commands:

   ```
   cargo dev sync update_nightly
   git commit -m "Bump nightly version -> YYYY-MM-DD" rust-toolchain.toml
   clippy_utils/README.md
   ```

5. 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 ask them in the Zulip stream.)

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

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

1. Make sure you have checked out the latest `master` of `rust-lang/rust`.
2. Sync the `rust-lang/rust-clippy` master to the rust-copy of Clippy:

   ```
   git switch -c clippy-subtree-update
   git subtree pull -P src/tools/clippy clippy-upstream master
   ```

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

Backport Changes

Sometimes it is necessary to backport changes to the beta release of Clippy. Backports in Clippy are rare and should be approved by the Clippy team. For example, a backport is done, if a crucial ICE was fixed or a lint is broken to a point, that it has to be disabled, before landing on stable.

Note: If you think a PR should be backported you can label it with `beta-nominated`. This has to be done before the Thursday the week before the release.

Filtering PRs to backport

First, find all labeled PRs using this filter.

Next, look at each PR individually. There are a few things to check. Those need some explanation and are quite subjective. Good judgement is required.

1. **Is the fix worth a backport?**

   This is really subjective. An ICE fix usually is. Moving a lint to a *lower* group (from warn- to allow-by-default) usually as well. An FP fix usually not (on its own). If a backport is done anyway, FP fixes might also be included. If the PR has a lot of changes, backports must be considered more carefully.

2. **Is the problem that was fixed by the PR already in `beta`?**

   It could be that the problem that was fixed by the PR hasn't made it to the `beta` branch of the Rust repo yet. If that's the case, and the fix is already synced to the Rust repo, the fix doesn't need to be backported, as it will hit stable together with the commit that introduced the problem. If the fix PR is not synced yet, the fix PR either needs to be "backported" to the Rust `master` branch or to `beta` in the next backport cycle.

3. **Make sure that the fix is on `master` before porting to `beta`**

   The fix must already be synced to the Rust `master` branch. Otherwise, the next `beta` will be missing this fix again. If it is not yet in `master` it should probably not be backported. If the backport is really important, do an out-of-cycle sync first. However, the out-of-cycle sync should be small, because the changes in that sync will get right into `beta`, without being tested in `nightly` first.

Preparation

Note: All commands in this chapter will be run in the Rust clone.

Follow the instructions in defining remotes to define the `clippy-upstream` remote in the Rust repository.

After that, fetch the remote with

```
git fetch clippy-upstream master
```

Then, switch to the `beta` branch:

```
git switch beta
git fetch upstream
git reset --hard upstream/beta
```

Backport the changes

When a PR is merged with the GitHub merge queue, the PR is closed with the message

> <PR title> (#<PR number>)

This commit needs to be backported. To do that, find the `<sha1>` of that commit and run the following command in the clone of the **Rust repository**:

```
git cherry-pick -m 1 `<sha1>`
```

Do this for all PRs that should be backported.

Open PR in the Rust repository

Next, open the PR for the backport. Make sure, the PR is opened towards the `beta` branch and not the `master` branch. The PR description should look like this:

```
[beta] Clippy backports

r? @Mark-Simulacrum

Backports:
- <Link to the Clippy PR>
- ...

<Short summary of what is backported and why>
```

Mark is from the release team and they ultimately have to merge the PR before branching a new `beta` version. Tag them to take care of the backport. Next, list all the backports and give a short summary what's backported and why it is worth backporting this.

Relabel backported PRs

When a PR is backported to Rust `beta`, label the PR with `beta-accepted`. This will then get picked up when writing the changelog.

Changelog Update

If you want to help with updating the changelog, you're in the right place.

When to update

Typos and other small fixes/additions are *always* welcome.

Special care needs to be taken when it comes to updating the changelog for a new Rust release. For that purpose, the changelog is ideally updated during the week before an upcoming stable release. You can find the release dates on the Rust Forge.

Most of the time we only need to update the changelog for minor Rust releases. It's been very rare that Clippy changes were included in a patch release.

Changelog update walkthrough

1. Finding the relevant Clippy commits

Each Rust release ships with its own version of Clippy. The Clippy subtree can be found in the `tools` directory of the Rust repository.

Depending on the current time and what exactly you want to update, the following bullet points might be helpful:

- When writing the release notes for the **upcoming stable release** you need to check out the Clippy commit of the current Rust `beta` branch. Link
- When writing the release notes for the **upcoming beta release**, you need to check out the Clippy commit of the current Rust `master` . Link
- When writing the (forgotten) release notes for a **past stable release**, you need to check out the Rust release tag of the stable release. Link

Usually you want to write the changelog of the **upcoming stable release**. Make sure though, that `beta` was already branched in the Rust repository.

To find the commit hash, issue the following command when in a `rust-lang/rust` checkout (most of the time on the `upstream/beta` branch):

```
git log --oneline -- src/tools/clippy/ | grep -o "Merge commit '[a-f0-9]*' into .*" |
head -1 | sed -e "s/Merge commit '\([a-f0-9]*\)' into .*/\1/g"
```

2. Fetching the PRs between those commits

Once you've got the correct commit range, run

```
util/fetch_prs_between.sh start_commit end_commit > changes.txt
```

where `end_commit` is the commit hash from the previous command and `start_commit` is the commit hash from the current CHANGELOG file. Open `changes.txt` file in your editor of choice.

3. Authoring the final changelog

The above script should have dumped all the relevant PRs to the file you specified. It should have filtered out most of the irrelevant PRs already, but it's a good idea to do a manual cleanup pass and choose valuable PRs. If you're not sure about some PRs, just leave them in for the review and ask for feedback.

With the PRs filtered, you can start to take each PR and move the `changelog:` content to `CHANGELOG.md`. Adapt the wording as you see fit but try to keep it somewhat coherent.

The sections order should roughly be:

```
### New Lints
* Added [`LINT`] to `GROUP`

### Moves and Deprecations
* Moved [`LINT`] to `GROUP` (From `GROUP`, now LEVEL-by-default)
* Renamed `LINT` to [`LINT`]

### Enhancements
### False Positive Fixes
### ICE Fixes
### Documentation Improvements
### Others
```

Please also be sure to update the `Unreleased/Beta/In Rust Nightly` section at the top with the relevant commits ranges and to add the `Rust <version>` section with release date and PR ranges.

4. Include `beta-accepted` PRs

Look for the `beta-accepted` label and make sure to also include the PRs with that label in the changelog. If you can, remove the `beta-accepted` labels **after** the changelog PR was merged.

> *Note:* Some of those PRs might even get backported to the previous `beta`. Those have to be included in the changelog of the *previous* release.

---

5. Update `clippy::version` attributes

Next, make sure to check that the `#[clippy::version]` attributes for the added lints contain the correct version. In order to find lints that need a version update, go through the lints in the "New Lints" section and run the following command for each lint name:

```
grep -rB1 "pub $LINT_NAME" .
```

The version shown should match the version of the release the changelog is written for. If not, update the version to the changelog version.

Release a new Clippy Version

NOTE: This document is probably only relevant to you, if you're a member of the Clippy team.

Clippy is released together with stable Rust releases. The dates for these releases can be found at the Rust Forge. This document explains the necessary steps to create a Clippy release.

Defining Remotes

You may want to define the `upstream` remote of the Clippy project to simplify the following steps. However, this is optional and you can replace `upstream` with the full URL instead.

```
git remote add upstream git@github.com:rust-lang/rust-clippy
```

Bump Version

When a release needs to be done, `cargo test` will fail, if the versions in the `Cargo.toml` are not correct. During that sync, the versions need to be bumped. This is done by running:

```
cargo dev release bump_version
```

This will increase the version number of each relevant `Cargo.toml` file. After that, just commit the updated files with:

```
git commit -m "Bump Clippy version -> 0.1.XY" **/*Cargo.toml
```

`XY` should be exchanged with the corresponding version

Find the Clippy commit

For both updating the `beta` and the `stable` branch, the first step is to find the Clippy commit of the last Clippy sync done in the respective Rust branch.

Running the following commands *in the Rust repo* will get the commit for the specified `<branch>` :

```
git switch <branch>
SHA=$(git log --oneline -- src/tools/clippy/ | grep -o "Merge commit '[a-f0-9]*' into
.*" | head -1 | sed -e "s/Merge commit '\([a-f0-9]*\)' into .*/\1/g")
```

Where `<branch>` is one of `stable`, `beta`, or `master`.

Update the `beta` branch

After getting the commit of the `beta` branch, the `beta` branch in the Clippy repository can be updated.

```
git checkout beta
git reset --hard $SHA
git push upstream beta
```

Update the `stable` branch

After getting the commit of the `stable` branch, the `stable` branch in the Clippy repository can be updated.

```
git checkout stable
git reset --hard $SHA
git push upstream stable
```

Tag the `stable` commit

After updating the `stable` branch, tag the HEAD commit and push it to the Clippy repo.

```
git tag rust-1.XX.0               # XX should be exchanged with the corresponding
version
git push upstream rust-1.XX.0     # `upstream` is the `rust-lang/rust-clippy` remote
```

After this, the release should be available on the Clippy tags page.

Publish `clippy_utils`

The `clippy_utils` crate is published to `crates.io` without any stability guarantees. To do this, after the sync and the release is done, switch back to the `upstream/master` branch and publish `clippy_utils`:

Note: The Rustup PR bumping the nightly and Clippy version **must** be merged before doing this.

```
git switch master && git pull upstream master
cargo publish --manifest-path clippy_utils/Cargo.toml
```

Update `CHANGELOG.md`

For this see the document on how to update the changelog.

If you don't have time to do a complete changelog update right away, just update the following parts:

- Remove the `(beta)` from the new stable version:

  ```
  ## Rust 1.XX (beta) -> ## Rust 1.XX
  ```

- Update the release date line of the new stable version:

  ```
  Current beta, release 20YY-MM-DD -> Current stable, released 20YY-MM-DD
  ```

- Update the release date line of the previous stable version:

  ```
  Current stable, released 20YY-MM-DD -> Released 20YY-MM-DD
  ```

The Clippy Book

This document explains how to make additions and changes to the Clippy book, the guide to Clippy that you're reading right now. The Clippy book is formatted with Markdown and generated by mdBook.

Get mdBook

While not strictly necessary since the book source is simply Markdown text files, having mdBook locally will allow you to build, test and serve the book locally to view changes before you commit them to the repository. You likely already have `cargo` installed, so the easiest option is to:

```
cargo install mdbook
```

See the mdBook installation instructions for other options.

Make changes

The book's src directory contains all the markdown files used to generate the book. If you want to see your changes in real time, you can use the mdBook `serve` command to run a web server locally that will automatically update changes as they are made. From the top level of your `rust-clippy` directory:

```
mdbook serve book --open
```

Then navigate to `http://localhost:3000` to see the generated book. While the server is running, changes you make will automatically be updated.

For more information, see the mdBook guide.

Benchmarking Clippy

Benchmarking Clippy is similar to using our Lintcheck tool, in fact, it even uses the same tool! Just by adding a `--perf` flag it will transform Lintcheck into a very simple but powerful benchmarking tool!

It requires having the `perf` tool installed, as `perf` is what's actually profiling Clippy under the hood.

The lintcheck `--perf` tool generates a series of `perf.data` in the `target/lintcheck/sources/<package>-<version>` directories. Each `perf.data` corresponds to the package which is contained.

Lintcheck uses the `-g` flag, meaning that you can get stack traces for richer analysis, including with tools such as flamegraph (or `flamegraph-rs`).

Currently, we only measure instruction count, as it's the most reproducible metric and rustc-perf also considers it the main number to focus on.

Benchmarking a PR

Having a benchmarking tool directly implemented into lintcheck gives us the ability to benchmark any given PR just by making a before and after

Here's the way you can get into any PR, benchmark it, and then benchmark `master`.

The first `perf.data` will not have any numbers appended, but any subsequent benchmark will be written to `perf.data.number` with a number growing for 0. All benchmarks are compressed so that you can

```
git fetch upstream pull/<PR_NUMBER>/head:<BRANCH_NAME>
git switch BRANCHNAME

# Bench
cargo lintcheck --perf

# Get last common commit, checkout that
LAST_COMMIT=$(git log BRANCHNAME..master --oneline | tail -1 | cut -c 1-11)
git switch -c temporary $LAST_COMMIT

# We're now on master

# Bench
cargo lintcheck --perf
perf diff ./target/lintcheck/sources/CRATE/perf.data ./target/lintcheck/sources/CRATE
/perf.data.0
```

Proposals

This chapter is about accepted proposals for changes that should be worked on in or around Clippy in the long run.

Besides adding more and more lints and improve the lints that Clippy already has, Clippy is also interested in making the experience of its users, developers and maintainers better over time. Projects that address bigger picture things like this usually take more time, and it is useful to have a proposal for those first. This is the place where such proposals are collected, so that we can refer to them when working on them.

Roadmap 2021

Summary

This Roadmap lays out the plans for Clippy in 2021:

- Improving usability and reliability
- Improving experience of contributors and maintainers
- Develop and specify processes

Members of the Clippy team will be assigned tasks from one or more of these topics. The team member is then responsible to complete the assigned tasks. This can either be done by implementing them or by providing mentorship to interested contributors.

Motivation

With the ongoing growth of the Rust language and with that of the whole ecosystem, also Clippy gets more and more users and contributors. This is good for the project, but also brings challenges along. Some of these challenges are:

- More issues about reliability or usability are popping up
- Traffic is hard to handle for a small team
- Bigger projects don't get completed due to the lack of processes and/or time of the team members

Additionally, according to the Rust Roadmap 2021, clear processes should be defined by every team and unified across teams. This Roadmap is the first step towards this.

Explanation

This section will explain the things that should be done in 2021. It is important to note, that this document focuses on the "What?", not the "How?". The later will be addressed in follow-up tracking issue, with an assigned team member.

The following is split up in two major sections. The first section covers the user facing plans, the second section the internal plans.

User Facing

Clippy should be as pleasant to use and configure as possible. This section covers plans that should be implemented to improve the situation of Clippy in this regard.

Usability

In the following, plans to improve the usability are covered.

No Output After `cargo check`

Currently, when `cargo clippy` is run after `cargo check`, it does not produce any output. This is especially problematic since `rust-analyzer` is on the rise, and it uses `cargo check` for checking code. A fix is already implemented, but it still has to be pushed over the finish line. This also includes the stabilization of the `cargo clippy --fix` command or the support of multi-span suggestions in `rustfix`.

- #4612

`lints.toml` Configuration

This is something that comes up every now and then: a reusable configuration file, where lint levels can be defined. Discussions about this often lead to nothing specific or to "we need an RFC for this". And this is exactly what needs to be done. Get together with the cargo team and write an RFC and implement such a configuration file somehow and somewhere.

- #3164
- cargo#5034
- IRLO

Lint Groups

There are more and more issues about managing lints in Clippy popping up. Lints are hard to implement with a guarantee of no/few false positives (FPs). One way to address this might be to introduce more lint groups to give users the ability to better manage lints, or improve the process of classifying lints, so that disabling lints due to FPs becomes rare. It is important to note, that Clippy lints are less conservative than `rustc` lints, which won't change in the future.

- #5537
- #6366

Reliability

In the following, plans to improve the reliability are covered.

False Positive Rate

In the worst case, new lints are only available in nightly for 2 weeks, before hitting beta and ultimately stable. This and the fact that fewer people use nightly Rust nowadays makes it more probable that a lint with many FPs hits stable. This leads to annoyed users, that will disable these new lints in the best case and to more annoyed users, that will stop using Clippy in the worst. A process should be developed and implemented to prevent this from happening.

- #6429

Internal

(The end of) 2020 has shown, that Clippy has to think about the available resources, especially

regarding management and maintenance of the project. This section address issues affecting team members and contributors.

Management

In 2020 Clippy achieved over 1000 open issues with regularly between 25-35 open PRs. This is simultaneously a win and a loss. More issues and PRs means more people are interested in Clippy and in contributing to it. On the other hand, it means for team members more work and for contributors longer wait times for reviews. The following will describe plans how to improve the situation for both team members and contributors.

Clear Expectations for Team Members

According to the Rust Roadmap 2021, a document specifying what it means to be a member of the team should be produced. This should not put more pressure on the team members, but rather help them and interested folks to know what the expectations are. With this it should also be easier to recruit new team members and may encourage people to get in touch, if they're interested to join.

Scaling up the Team

More people means less work for each individual. Together with the document about expectations for team members, a document defining the process of how to join the team should be produced. This can also increase the stability of the team, in case of current members dropping out (temporarily). There can also be different roles in the team, like people triaging vs. people reviewing.

Regular Meetings

Other teams have regular meetings. Clippy is big enough that it might be worth to also do them. Especially if more people join the team, this can be important for sync-ups. Besides the asynchronous communication, that works well for working on separate lints, a meeting adds a synchronous alternative at a known time. This is especially helpful if there are bigger things that need to be discussed (like the projects in this roadmap). For starters bi-weekly meetings before Rust syncs might make sense.

Triaging

To get a handle on the influx of open issues, a process for triaging issues and PRs should be developed. Officially, Clippy follows the Rust triage process, but currently no one enforces it. This can be improved by sharing triage teams across projects or by implementing dashboards / tools which simplify triaging.

Development

Improving the developer and contributor experience is something the Clippy team works on regularly. Though, some things might need special attention and planing. These topics are listed in the following.

Process for New and Existing Lints

As already mentioned above, classifying new lints gets quite hard, because the probability of a buggy lint getting into stable is quite high. A process should be implemented on how to classify lints. In addition, a test system should be developed to find out which lints are currently problematic in real world code to fix or disable them.

- #6429 (comment)
- #6429 (comment)

Processes

Related to the point before, a process for suggesting and discussing major changes should be implemented. It's also not clearly defined when a lint should be enabled or disabled by default. This can also be improved by the test system mentioned above.

Dev-Tools

There's already `cargo dev` which makes Clippy development easier and more pleasant. This can still be expanded, so that it covers more areas of the development process.

- #5394

Contributor Guide

Similar to a Clippy Book, which describes how to use Clippy, a book about how to contribute to Clippy might be helpful for new and existing contributors. There's already the `doc` directory in the Clippy repo, this can be turned into a `mdbook`.

`rustc` integration

Recently Clippy was integrated with `git subtree` into the `rust-lang/rust` repository. This made syncing between the two repositories easier. A `#[non_exhaustive]` list of things that still can be improved is:

1. Use the same `rustfmt` version and configuration as `rustc`.
2. Make `cargo dev` work in the Rust repo, just as it works in the Clippy repo. E.g. `cargo dev bless` or `cargo dev update_lints`. And even add more things to it that might be useful for the Rust repo, e.g. `cargo dev deprecate`.
3. Easier sync process. The `subtree` situation is not ideal.

Prioritization

The most pressing issues for users of Clippy are of course the user facing issues. So there should be a priority on those issues, but without losing track of the internal issues listed in this document.

Getting the FP rate of warn/deny-by-default lints under control should have the highest priority. Other user facing issues should also get a high priority, but shouldn't be in the way of addressing internal issues.

To better manage the upcoming projects, the basic internal processes, like meetings, tracking issues and documentation, should be established as soon as possible. They might even be necessary to properly manage the projects, regarding the user facing issues.

Prior Art

Rust Roadmap

Rust's roadmap process was established by RFC 1728 in 2016. Since then every year a roadmap was published, that defined the bigger plans for the coming years. This year roadmap can be found here.

Drawbacks

Big Roadmap

This roadmap is pretty big and not all items listed in this document might be addressed during 2021. Because this is the first roadmap for Clippy, having open tasks at the end of 2021 is fine, but they should be revisited in the 2022 roadmap.

- Feature Name: syntax-tree-patterns
- Start Date: 2019-03-12
- RFC PR: #3875

Summary

Introduce a domain-specific language (similar to regular expressions) that allows to describe lints using *syntax tree patterns*.

Motivation

Finding parts of a syntax tree (AST, HIR, ...) that have certain properties (e.g. "*an if that has a block as its condition*") is a major task when writing lints. For non-trivial lints, it often requires nested pattern matching of AST / HIR nodes. For example, testing that an expression is a boolean literal requires the following checks:

```
if let ast::ExprKind::Lit(lit) = &expr.node {
    if let ast::LitKind::Bool(_) = &lit.node {
        ...
    }
}
```

Writing this kind of matching code quickly becomes a complex task and the resulting code is often hard to comprehend. The code below shows a simplified version of the pattern matching required by the `collapsible_if` lint:

```
// simplified version of the collapsible_if lint
if let ast::ExprKind::If(check, then, None) = &expr.node {
    if then.stmts.len() == 1 {
        if let ast::StmtKind::Expr(inner) | ast::StmtKind::Semi(inner) =
&then.stmts[0].node {
            if let ast::ExprKind::If(check_inner, content, None) = &inner.node {
                ...
            }
        }
    }
}
```

The `if_chain` macro can improve readability by flattening the nested if statements, but the resulting code is still quite hard to read:

```
// simplified version of the collapsible_if lint
if_chain! {
    if let ast::ExprKind::If(check, then, None) = &expr.node;
    if then.stmts.len() == 1;
    if let ast::StmtKind::Expr(inner) | ast::StmtKind::Semi(inner) =
&then.stmts[0].node;
    if let ast::ExprKind::If(check_inner, content, None) = &inner.node;
    then {
        ...
    }
}
```

The code above matches if expressions that contain only another if expression (where both ifs don't have an else branch). While it's easy to explain what the lint does, it's hard to see that from looking

at the code samples above.

Following the motivation above, the first goal this RFC is to **simplify writing and reading lints**.

The second part of the motivation is clippy's dependence on unstable compiler-internal data structures. Clippy lints are currently written against the compiler's AST / HIR which means that even small changes in these data structures might break a lot of lints. The second goal of this RFC is to **make lints independent of the compiler's AST / HIR data structures**.

Approach

A lot of complexity in writing lints currently seems to come from having to manually implement the matching logic (see code samples above). It's an imperative style that describes *how* to match a syntax tree node instead of specifying *what* should be matched against declaratively. In other areas, it's common to use declarative patterns to describe desired information and let the implementation do the actual matching. A well-known example of this approach are regular expressions. Instead of writing code that detects certain character sequences, one can describe a search pattern using a domain-specific language and search for matches using that pattern. The advantage of using a declarative domain-specific language is that its limited domain (e.g. matching character sequences in the case of regular expressions) allows to express entities in that domain in a very natural and expressive way.

While regular expressions are very useful when searching for patterns in flat character sequences, they cannot easily be applied to hierarchical data structures like syntax trees. This RFC therefore proposes a pattern matching system that is inspired by regular expressions and designed for hierarchical syntax trees.

Guide-level explanation

This proposal adds a `pattern!` macro that can be used to specify a syntax tree pattern to search for. A simple pattern is shown below:

```
pattern!{
    my_pattern: Expr =
        Lit(Bool(false))
}
```

This macro call defines a pattern named `my_pattern` that can be matched against an `Expr` syntax tree node. The actual pattern (`Lit(Bool(false))` in this case) defines which syntax trees should match the pattern. This pattern matches expressions that are boolean literals with value `false` .

The pattern can then be used to implement lints in the following way:

```
...

impl EarlyLintPass for MyAwesomeLint {
    fn check_expr(&mut self, cx: &EarlyContext, expr: &syntax::ast::Expr) {

        if my_pattern(expr).is_some() {
            cx.span_lint(
                MY_AWESOME_LINT,
                expr.span,
                "This is a match for a simple pattern. Well done!",
            );
        }

    }
}
```

The `pattern!` macro call expands to a function `my_pattern` that expects a syntax tree expression as its argument and returns an `Option` that indicates whether the pattern matched.

Note: The result type is explained in more detail in a later section. For now, it's enough to know that the result is `Some` if the pattern matched and `None` otherwise.

Pattern syntax

The following examples demonstrate the pattern syntax:

Any (_)

The simplest pattern is the any pattern. It matches anything and is therefore similar to regex's `*`.

```
pattern!{
    // matches any expression
    my_pattern: Expr =
        _
}
```

Node (<node-name>(<args>))

Nodes are used to match a specific variant of an AST node. A node has a name and a number of arguments that depends on the node type. For example, the `Lit` node has a single argument that describes the type of the literal. As another example, the `If` node has three arguments describing the if's condition, then block and else block.

```
pattern!{
    // matches any expression that is a literal
    my_pattern: Expr =
        Lit(_)
}

pattern!{
    // matches any expression that is a boolean literal
    my_pattern: Expr =
        Lit(Bool(_))
}

pattern!{
    // matches if expressions that have a boolean literal in their condition
    // Note: The `_?` syntax here means that the else branch is optional and can be
anything.
    //       This is discussed in more detail in the section `Repetition`.
    my_pattern: Expr =
        If( Lit(Bool(_)) , _, _?)
}
```

Literal (`<lit>`)

A pattern can also contain Rust literals. These literals match themselves.

```
pattern!{
    // matches the boolean literal false
    my_pattern: Expr =
        Lit(Bool(false))
}

pattern!{
    // matches the character literal 'x'
    my_pattern: Expr =
        Lit(Char('x'))
}
```

Alternations (`a | b`)

```
pattern!{
    // matches if the literal is a boolean or integer literal
    my_pattern: Lit =
        Bool(_) | Int(_)
}

pattern!{
    // matches if the expression is a char literal with value 'x' or 'y'
    my_pattern: Expr =
        Lit( Char('x' | 'y') )
}
```

Empty (`()`)

The empty pattern represents an empty sequence or the `None` variant of an optional.

```
pattern!{
    // matches if the expression is an empty array
    my_pattern: Expr =
        Array( () )
}

pattern!{
    // matches if expressions that don't have an else clause
    my_pattern: Expr =
        If(_, _, ())
}
```

Sequence (`<a> <b>`)

```
pattern!{
    // matches the array [true, false]
    my_pattern: Expr =
        Array( Lit(Bool(true)) Lit(Bool(false)) )
}
```

Repetition (`<a>*`, `<a>+`, `<a>?`, `<a>{n}`, `<a>{n,m}`, `<a>{n,}`)

Elements may be repeated. The syntax for specifying repetitions is identical to regex's syntax.

```
pattern!{
    // matches arrays that contain 2 'x's as their last or second-last elements
    // Examples:
    //     ['x', 'x']                          match
    //     ['x', 'x', 'y']                     match
    //     ['a', 'b', 'c', 'x', 'x', 'y']      match
    //     ['x', 'x', 'y', 'z']                no match
    my_pattern: Expr =
        Array( _* Lit(Char('x')){2} _? )
}

pattern!{
    // matches if expressions that **may or may not** have an else block
    // Attn: `If(_, _, _)` matches only ifs that **have** an else block
    //
    //              | if with else block | if without else block
    // If(_, _, _)  |       match        |       no match
    // If(_, _, _?) |       match        |        match
    // If(_, _, ()) |      no match      |        match
    my_pattern: Expr =
        If(_, _, _?)
}
```

Named submatch (`<a>#<name>`)

```
pattern!{
    // matches character literals and gives the literal the name foo
    my_pattern: Expr =
        Lit(Char(_)#foo)
}

pattern!{
    // matches character literals and gives the char the name bar
    my_pattern: Expr =
        Lit(Char(_#bar))
}

pattern!{
    // matches character literals and gives the expression the name baz
    my_pattern: Expr =
        Lit(Char(_))#baz
}
```

The reason for using named submatches is described in the section The result type.

Summary

The following table gives an summary of the pattern syntax:

Syntax	Concept	Examples
`_`	Any	`_`
`<node-name>(<args>)`	Node	`Lit(Bool(true))`, `If(_, _, _)`
`<lit>`	Literal	`'x'`, `false`, `101`
`<a> | <b>`	Alternation	`Char(_) | Bool(_)`
`()`	Empty	`Array( () )`
`<a> <b>`	Sequence	`Tuple( Lit(Bool(_)) Lit(Int(_)) Lit(_) )`

`<a>*`
`<a>+`
`<a>?`
`<a>{n}`
`<a>{n,m}`
`<a>{n,}`
	Repetition	`Array( _* )`,
		`Block( Semi(_)+ )`,
		`If(_, _, Block(_)?)`,
		`Array( Lit(_){10} )`,
		`Lit(_){5,10}`,
		`Lit(Bool(_)){10,}`

`<a>#<name>`	Named submatch	`Lit(Int(_))#foo  Lit(Int(_#bar))`

The result type

A lot of lints require checks that go beyond what the pattern syntax described above can express. For example, a lint might want to check whether a node was created as part of a macro expansion or whether there's no comment above a node. Another example would be a lint that wants to match two nodes that have the same value (as needed by lints like `almost_swapped`). Instead of allowing users to write these checks into the pattern directly (which might make patterns hard to read), the proposed solution allows users to assign names to parts of a pattern expression. When matching a

pattern against a syntax tree node, the return value will contain references to all nodes that were matched by these named subpatterns. This is similar to capture groups in regular expressions.

For example, given the following pattern

```
pattern!{
    // matches character literals
    my_pattern: Expr =
        Lit(Char(_#val_inner)#val)#val_outer
}
```

one could get references to the nodes that matched the subpatterns in the following way:

```
...
fn check_expr(expr: &syntax::ast::Expr) {
    if let Some(result) = my_pattern(expr) {
        result.val_inner  // type: &char
        result.val        // type: &syntax::ast::Lit
        result.val_outer  // type: &syntax::ast::Expr
    }
}
```

The types in the `result` struct depend on the pattern. For example, the following pattern

```
pattern!{
    // matches arrays of character literals
    my_pattern_seq: Expr =
        Array( Lit(_)*#foo )
}
```

matches arrays that consist of any number of literal expressions. Because those expressions are named `foo`, the result struct contains a `foo` attribute which is a vector of expressions:

```
...
if let Some(result) = my_pattern_seq(expr) {
    result.foo        // type: Vec<&syntax::ast::Expr>
}
```

Another result type occurs when a name is only defined in one branch of an alternation:

```
pattern!{
    // matches if expression is a boolean or integer literal
    my_pattern_alt: Expr =
        Lit( Bool(_#bar) | Int(_) )
}
```

In the pattern above, the `bar` name is only defined if the pattern matches a boolean literal. If it matches an integer literal, the name isn't set. To account for this, the result struct's `bar` attribute is an option type:

```
...
if let Some(result) = my_pattern_alt(expr) {
    result.bar        // type: Option<&bool>
}
```

It's also possible to use a name in multiple alternation branches if they have compatible types:

```
pattern!{
    // matches if expression is a boolean or integer literal
    my_pattern_mult: Expr =
        Lit(_#baz) | Array( Lit(_#baz) )
}
...
if let Some(result) = my_pattern_mult(expr) {
    result.baz        // type: &syntax::ast::Lit
}
```

Named submatches are a **flat** namespace and this is intended. In the example above, two different sub-structures are assigned to a flat name. I expect that for most lints, a flat namespace is sufficient and easier to work with than a hierarchical one.

Two stages

Using named subpatterns, users can write lints in two stages. First, a coarse selection of possible matches is produced by the pattern syntax. In the second stage, the named subpattern references can be used to do additional tests like asserting that a node hasn't been created as part of a macro expansion.

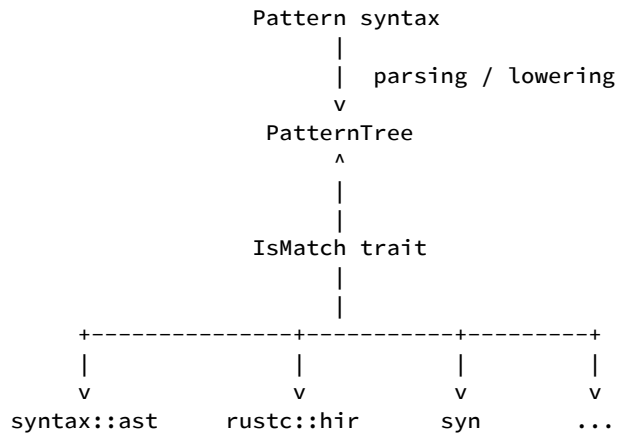
Implementing Clippy lints using patterns

As a "real-world" example, I re-implemented the `collapsible_if` lint using patterns. The code can be found here. The pattern-based version passes all test cases that were written for `collapsible_if`.

Reference-level explanation

Overview

The following diagram shows the dependencies between the main parts of the proposed solution:



The pattern syntax described in the previous section is parsed / lowered into the so-called *PatternTree* data structure that represents a valid syntax tree pattern. Matching a *PatternTree* against

an actual syntax tree (e.g. rust ast / hir or the syn ast, ...) is done using the *IsMatch* trait.

The *PatternTree* and the *IsMatch* trait are introduced in more detail in the following sections.

PatternTree

The core data structure of this RFC is the **PatternTree**.

It's a data structure similar to rust's AST / HIR, but with the following differences:

- The PatternTree doesn't contain parsing information like `Span` s
- The PatternTree can represent alternatives, sequences and optionals

The code below shows a simplified version of the current PatternTree:

Note: The current implementation can be found here.

```
pub enum Expr {
    Lit(Alt<Lit>),
    Array(Seq<Expr>),
    Block_(Alt<BlockType>),
    If(Alt<Expr>, Alt<BlockType>, Opt<Expr>),
    IfLet(
        Alt<BlockType>,
        Opt<Expr>,
    ),
}

pub enum Lit {
    Char(Alt<char>),
    Bool(Alt<bool>),
    Int(Alt<u128>),
}

pub enum Stmt {
    Expr(Alt<Expr>),
    Semi(Alt<Expr>),
}

pub enum BlockType {
    Block(Seq<Stmt>),
}
```

The `Alt`, `Seq` and `Opt` structs look like these:

Note: The current implementation can be found here.

```
pub enum Alt<T> {
    Any,
    Elmt(Box<T>),
    Alt(Box<Self>, Box<Self>),
    Named(Box<Self>, ...)
}

pub enum Opt<T> {
    Any,  // anything, but not None
    Elmt(Box<T>),
    None,
    Alt(Box<Self>, Box<Self>),
    Named(Box<Self>, ...)
}

pub enum Seq<T> {
    Any,
    Empty,
    Elmt(Box<T>),
    Repeat(Box<Self>, RepeatRange),
    Seq(Box<Self>, Box<Self>),
    Alt(Box<Self>, Box<Self>),
    Named(Box<Self>, ...)
}

pub struct RepeatRange {
    pub start: usize,
    pub end: Option<usize>  // exclusive
}
```

Parsing / Lowering

The input of a `pattern!` macro call is parsed into a `ParseTree` first and then lowered to a `PatternTree`.

Valid patterns depend on the *PatternTree* definitions. For example, the pattern `Lit(Bool(_)*)` isn't valid because the parameter type of the `Lit` variant of the `Expr` enum is `Any<Lit>` and therefore doesn't support repetition (`*`). As another example, `Array( Lit(_)* )` is a valid pattern because the parameter of `Array` is of type `Seq<Expr>` which allows sequences and repetitions.

> Note: names in the pattern syntax correspond to *PatternTree* enum **variants**. For example, the `Lit` in the pattern above refers to the `Lit` variant of the `Expr` enum (`Expr::Lit`), not the `Lit` enum.

The IsMatch Trait

The pattern syntax and the *PatternTree* are independent of specific syntax tree implementations (rust ast / hir, syn, ...). When looking at the different pattern examples in the previous sections, it can be seen that the patterns don't contain any information specific to a certain syntax tree implementation. In contrast, Clippy lints currently match against ast / hir syntax tree nodes and therefore directly depend on their implementation.

The connection between the *PatternTree* and specific syntax tree implementations is the `IsMatch` trait. It defines how to match *PatternTree* nodes against specific syntax tree nodes. A simplified implementation of the `IsMatch` trait is shown below:

```
pub trait IsMatch<O> {
    fn is_match(&self, other: &'o O) -> bool;
}
```

This trait needs to be implemented on each enum of the *PatternTree* (for the corresponding syntax tree types). For example, the `IsMatch` implementation for matching `ast::LitKind` against the *PatternTree's* `Lit` enum might look like this:

```
impl IsMatch<ast::LitKind> for Lit {
    fn is_match(&self, other: &ast::LitKind) -> bool {
        match (self, other) {
            (Lit::Char(i), ast::LitKind::Char(j)) => i.is_match(j),
            (Lit::Bool(i), ast::LitKind::Bool(j)) => i.is_match(j),
            (Lit::Int(i), ast::LitKind::Int(j, _)) => i.is_match(j),
            _ => false,
        }
    }
}
```

All `IsMatch` implementations for matching the current *PatternTree* against `syntax::ast` can be found here.

Drawbacks

Performance

The pattern matching code is currently not optimized for performance, so it might be slower than hand-written matching code. Additionally, the two-stage approach (matching against the coarse pattern first and checking for additional properties later) might be slower than the current practice of checking for structure and additional properties in one pass. For example, the following lint

```
pattern!{
    pat_if_without_else: Expr =
        If(
            _,
            Block(
                Expr( If(_, _, ())#inner )
                | Semi( If(_, _, ())#inner )
            )#then,
            ()
        )
}
...
fn check_expr(&mut self, cx: &EarlyContext<'_>, expr: &ast::Expr) {
    if let Some(result) = pat_if_without_else(expr) {
        if !block_starts_with_comment(cx, result.then) {
            ...
        }
}
```

first matches against the pattern and then checks that the `then` block doesn't start with a comment. Using clippy's current approach, it's possible to check for these conditions earlier:

```
fn check_expr(&mut self, cx: &EarlyContext<'_>, expr: &ast::Expr) {
    if_chain! {
        if let ast::ExprKind::If(ref check, ref then, None) = expr.node;
        if !block_starts_with_comment(cx, then);
        if let Some(inner) = expr_block(then);
        if let ast::ExprKind::If(ref check_inner, ref content, None) = inner.node;
        then {
            ...
        }
    }
}
```

Whether or not this causes performance regressions depends on actual patterns. If it turns out to be a problem, the pattern matching algorithms could be extended to allow "early filtering" (see the Early Filtering section in Future Possibilities).

That being said, I don't see any conceptual limitations regarding pattern matching performance.

Applicability

Even though I'd expect that a lot of lints can be written using the proposed pattern syntax, it's unlikely that all lints can be expressed using patterns. I suspect that there will still be lints that need to be implemented by writing custom pattern matching code. This would lead to mix within clippy's codebase where some lints are implemented using patterns and others aren't. This inconsistency might be considered a drawback.

Rationale and alternatives

Specifying lints using syntax tree patterns has a couple of advantages compared to the current approach of manually writing matching code. First, syntax tree patterns allow users to describe patterns in a simple and expressive way. This makes it easier to write new lints for both novices and experts and also makes reading / modifying existing lints simpler.

Another advantage is that lints are independent of specific syntax tree implementations (e.g. AST / HIR, ...). When these syntax tree implementations change, only the `IsMatch` trait implementations need to be adapted and existing lints can remain unchanged. This also means that if the `IsMatch` trait implementations were integrated into the compiler, updating the `IsMatch` implementations would be required for the compiler to compile successfully. This could reduce the number of times Clippy breaks because of changes in the compiler. Another advantage of the pattern's independence is that converting an `EarlyLintPass` lint into a `LatePassLint` wouldn't require rewriting the whole pattern matching code. In fact, the pattern might work just fine without any adaptions.

Alternatives

Rust-like pattern syntax

The proposed pattern syntax requires users to know the structure of the `PatternTree` (which is very similar to the AST's / HIR's structure) and also the pattern syntax. An alternative would be to introduce a pattern syntax that is similar to actual Rust syntax (probably like the `quote!` macro). For

example, a pattern that matches `if` expressions that have `false` in their condition could look like this:

```
if false {
    #[*]
}
```

Problems

Extending Rust syntax (which is quite complex by itself) with additional syntax needed for specifying patterns (alternations, sequences, repetitions, named submatches, ...) might become difficult to read and really hard to parse properly.

For example, a pattern that matches a binary operation that has `0` on both sides might look like this:

```
0 #[*:BinOpKind] 0
```

Now consider this slightly more complex example:

```
1 + 0 #[*:BinOpKind] 0
```

The parser would need to know the precedence of `#[*:BinOpKind]` because it affects the structure of the resulting AST. `1 + 0 + 0` is parsed as `(1 + 0) + 0` while `1 + 0 * 0` is parsed as `1 + (0 * 0)`. Since the pattern could be any `BinOpKind`, the precedence cannot be known in advance.

Another example of a problem would be named submatches. Take a look at this pattern:

```
fn test() {
    1 #foo
}
```

Which node is `#foo` referring to? `int`, `ast::Lit`, `ast::Expr`, `ast::Stmt`? Naming subpatterns in a rust-like syntax is difficult because a lot of AST nodes don't have a syntactic element that can be used to put the name tag on. In these situations, the only sensible option would be to assign the name tag to the outermost node (`ast::Stmt` in the example above), because the information of all child nodes can be retrieved through the outermost node. The problem with this then would be that accessing inner nodes (like `ast::Lit`) would again require manual pattern matching.

In general, Rust syntax contains a lot of code structure implicitly. This structure is reconstructed during parsing (e.g. binary operations are reconstructed using operator precedence and left-to-right) and is one of the reasons why parsing is a complex task. The advantage of this approach is that writing code is simpler for users.

When writing *syntax tree patterns*, each element of the hierarchy might have alternatives, repetitions, etc.. Respecting that while still allowing human-friendly syntax that contains structure implicitly seems to be really complex, if not impossible.

Developing such a syntax would also require to maintain a custom parser that is at least as complex as the Rust parser itself. Additionally, future changes in the Rust syntax might be incompatible with such a syntax.

In summary, I think that developing such a syntax would introduce a lot of complexity to solve a

relatively minor problem.

The issue of users not knowing about the *PatternTree* structure could be solved by a tool that, given a rust program, generates a pattern that matches only this program (similar to the Clippy author lint).

For some simple cases (like the first example above), it might be possible to successfully mix Rust and pattern syntax. This space could be further explored in a future extension.

Prior art

The pattern syntax is heavily inspired by regular expressions (repetitions, alternatives, sequences, ...).

From what I've seen until now, other linters also implement lints that directly work on syntax tree data structures, just like Clippy does currently. I would therefore consider the pattern syntax to be *new*, but please correct me if I'm wrong.

Unresolved questions

How to handle multiple matches?

When matching a syntax tree node against a pattern, there are possibly multiple ways in which the pattern can be matched. A simple example of this would be the following pattern:

```
pattern!{
    my_pattern: Expr =
        Array( _* Lit(_)+#literals)
}
```

This pattern matches arrays that end with at least one literal. Now given the array `[x, 1, 2]`, should `1` be matched as part of the `_*` or the `Lit(_)+` part of the pattern? The difference is important because the named submatch `#literals` would contain 1 or 2 elements depending how the pattern is matched. In regular expressions, this problem is solved by matching "greedy" by default and "non-greedy" optionally.

I haven't looked much into this yet because I don't know how relevant it is for most lints. The current implementation simply returns the first match it finds.

Future possibilities

Implement rest of Rust Syntax

The current project only implements a small part of the Rust syntax. In the future, this should incrementally be extended to more syntax to allow implementing more lints. Implementing more of the Rust syntax requires extending the `PatternTree` and `IsMatch` implementations, but should be relatively straight-forward.

Early filtering

As described in the *Drawbacks/Performance* section, allowing additional checks during the pattern

matching might be beneficial.

The pattern below shows how this could look like:

```
pattern!{
    pat_if_without_else: Expr =
        If(
            _,
            Block(
                Expr( If(_, _, ())#inner )
                | Semi( If(_, _, ())#inner )
            )#then,
            ()
        )
    where
        !in_macro(#then.span);
}
```

The difference compared to the currently proposed two-stage filtering is that using early filtering, the condition (`!in_macro(#then.span)` in this case) would be evaluated as soon as the `Block(_)#then` was matched.

Another idea in this area would be to introduce a syntax for backreferences. They could be used to require that multiple parts of a pattern should match the same value. For example, the `assign_op_pattern` lint that searches for `a = a op b` and recommends changing it to `a op= b` requires that both occurrences of `a` are the same. Using `=#...` as syntax for backreferences, the lint could be implemented like this:

```
pattern!{
    assign_op_pattern: Expr =
        Assign(_#target, Binary(_, =#target, _)
}
```

Match descendant

A lot of lints currently implement custom visitors that check whether any subtree (which might not be a direct descendant) of the current node matches some properties. This cannot be expressed with the proposed pattern syntax. Extending the pattern syntax to allow patterns like "a function that contains at least two return statements" could be a practical addition.

Negation operator for alternatives

For patterns like "a literal that is not a boolean literal" one currently needs to list all alternatives except the boolean case. Introducing a negation operator that allows to write `Lit(!Bool(_))` might be a good idea. This pattern would be equivalent to `Lit( Char(_) | Int(_) )` (given that currently only three literal types are implemented).

Functional composition

Patterns currently don't have any concept of composition. This leads to repetitions within patterns. For example, one of the collapsible-if patterns currently has to be written like this:

```
pattern!{
    pat_if_else: Expr =
        If(
            _,
            _,
            Block_(
                Block(
                    Expr((If(_, _, _?) | IfLet(_, _?))#else_) |
                    Semi((If(_, _, _?) | IfLet(_, _?))#else_)
                )#block_inner
            )#block
        ) |
        IfLet(
            _,
            Block_(
                Block(
                    Expr((If(_, _, _?) | IfLet(_, _?))#else_) |
                    Semi((If(_, _, _?) | IfLet(_, _?))#else_)
                )#block_inner
            )#block
        )
}
```

If patterns supported defining functions of subpatterns, the code could be simplified as follows:

```
pattern!{
    fn expr_or_semi(expr: Expr) -> Stmt {
        Expr(expr) | Semi(expr)
    }
    fn if_or_if_let(then: Block, else: Opt<Expr>) -> Expr {
        If(_, then, else) | IfLet(then, else)
    }
    pat_if_else: Expr =
        if_or_if_let(
            _,
            Block_(
                Block(
                    expr_or_semi( if_or_if_let(_, _?)#else_ )
                )#block_inner
            )#block
        )
}
```

Additionally, common patterns like `expr_or_semi` could be shared between different lints.

Clippy Pattern Author

Another improvement could be to create a tool that, given some valid Rust syntax, generates a pattern that matches this syntax exactly. This would make starting to write a pattern easier. A user could take a look at the patterns generated for a couple of Rust code examples and use that information to write a pattern that matches all of them.

This is similar to clippy's author lint.

Supporting other syntaxes

Most of the proposed system is language-agnostic. For example, the pattern syntax could also be used to describe patterns for other programming languages.

In order to support other languages' syntaxes, one would need to implement another `PatternTree` that sufficiently describes the languages' AST and implement `IsMatch` for this `PatternTree` and the languages' AST.

One aspect of this is that it would even be possible to write lints that work on the pattern syntax itself. For example, when writing the following pattern

```
pattern!{
    my_pattern: Expr =
        Array( Lit(Bool(false)) Lit(Bool(false)) )
}
```

a lint that works on the pattern syntax's AST could suggest using this pattern instead:

```
pattern!{
    my_pattern: Expr =
        Array( Lit(Bool(false)){2} )
}
```

In the future, Clippy could use this system to also provide lints for custom syntaxes like those found in macros.

The team

Everyone who contributes to Clippy makes the project what it is. Collaboration and discussions are the lifeblood of every open-source project. Clippy has a very flat hierarchy. The teams mainly have additional access rights to the repo.

This document outlines the onboarding process, as well as duties, and access rights for members of a group.

All regular events mentioned in this chapter are tracked in the calendar repository. The calendar file is also available for download: clippy.ics

Everyone

Everyone, including you, is welcome to join discussions and contribute in other ways, like PRs.

You also have some triage rights, using `@rustbot` to add labels and claim issues. See labeling with @rustbot.

A rule for everyone should be to keep a healthy work-life balance. Take a break when you need one.

Clippy-Contributors

This is a group of regular contributors to Clippy to help with triaging.

Duties

This team exists to make contributing easier for regular members. It doesn't carry any duties that need to be done. However, we want to encourage members of this group to help with triaging, which can include:

1. **Labeling issues**

   For the `good first issue` label, it can still be good to use `@rustbot` to subscribe to the issue and help interested parties, if they post questions in the comments.

2. **Closing duplicate or resolved issues**

   When you manually close an issue, it's often a good idea, to add a short comment explaining the reason.

3. **Ping people after two weeks of inactivity**

   We try to keep issue assignments and PRs fairly up-to-date. After two weeks, it can be good to send a friendly ping to the delaying party.

   You might close a PR with the `I-inactive-closed` label if the author is busy or wants to abandon it. If the reviewer is busy, the PR can be reassigned to someone else.

Checkout: https://triage.rust-lang.org/triage/rust-lang/rust-clippy to monitor PRs.

While not part of their duties, contributors are encouraged to review PRs and help on Zulip. The team always appreciates help!

Membership

If you have been contributing to Clippy for some time, we'll probably ask you if you want to join this team. Members of this team are also welcome to suggest people who they think would make a great addition to this group.

For this group, there is no direct onboarding process. You're welcome to just continue what you've been doing. If you like, you can ask for someone to mentor you, either in the Clippy stream on Zulip or privately via a PM.

If you have been inactive in Clippy for over three months, we'll probably move you to the alumni group. You're always welcome to come back.

The Clippy Team

The Clippy team is responsible for maintaining Clippy.

Duties

1. **Respond to PRs in a timely manner**

   It's totally fine, if you don't have the time for reviews right now. You can reassign the PR to a random member by commenting `r? clippy`.

2. **Take a break when you need one**

   You are valuable! Clippy wouldn't be what it is without you. So take a break early and recharge some energy when you need to.

3. **Be responsive on Zulip**

   This means in a reasonable time frame, so responding within one or two days is totally fine.

   It's also good, if you answer threads on Zulip and take part in our Clippy meetings, every two weeks. The meeting dates are tracked in the calendar repository.

4. **Sync Clippy with the rust-lang/rust repo**

   This is done every two weeks, usually by @flip1995.

5. **Update the changelog**

   This needs to be done for every release, every six weeks.

Membership

If you have been active for some time, we'll probably reach out and ask if you want to help with reviews and eventually join the Clippy team.

During the onboarding process, you'll be assigned pull requests to review. You'll also have an active team member as a mentor who'll stay in contact via Zulip DMs to provide advice and feedback. If you have questions, you're always welcome to ask, that is the best way to learn. Once you're done with the review, you can ping your mentor for a full review and to r+ the PR in both of your names.

When your mentor is confident that you can handle reviews on your own, they'll start an informal vote among the active team members to officially add you to the team. This vote is usually accepted unanimously. Then you'll be added to the team once you've confirmed that you're still interested in joining. The onboarding phase typically takes a couple of weeks to a few months.

If you have been inactive in Clippy for over three months, we'll probably move you to the alumni group. You're always welcome to come back.