What is rustc?

Welcome to "The rustc book"! `rustc` is the compiler for the Rust programming language, provided by the project itself. Compilers take your source code and produce binary code, either as a library or executable.

Most Rust programmers don't invoke `rustc` directly, but instead do it through Cargo. It's all in service of `rustc` though! If you want to see how Cargo calls `rustc`, you can

```
$ cargo build --verbose
```

And it will print out each `rustc` invocation. This book can help you understand what each of these options does. Additionally, while most Rustaceans use Cargo, not all do: sometimes they integrate `rustc` into other build systems. This book should provide a guide to all of the options you'd need to do so.

Basic usage

Let's say you've got a little hello world program in a file `hello.rs`:

```
fn main() {
    println!("Hello, world!");
}
```

To turn this source code into an executable, you can use `rustc`:

```
$ rustc hello.rs
$ ./hello # on a *NIX
$ .\hello.exe # on Windows
```

Note that we only ever pass `rustc` the *crate root*, not every file we wish to compile. For example, if we had a `main.rs` that looked like this:

```
mod foo;

fn main() {
    foo::hello();
}
```

And a `foo.rs` that had this:

```
pub fn hello() {
    println!("Hello, world!");
}
```

To compile this, we'd run this command:

```
$ rustc main.rs
```

No need to tell `rustc` about `foo.rs`; the `mod` statements give it everything that it needs. This is different than how you would use a C compiler, where you invoke the compiler on each file, and then link everything together. In other words, the *crate* is a translation unit, not a particular module.

Command-line arguments

Here's a list of command-line arguments to `rustc` and what they do.

`-h`/`--help`: get help

This flag will print out help information for `rustc`.

`--cfg`: configure the compilation environment

This flag can turn on or off various `#[cfg]` settings for conditional compilation.

The value can either be a single identifier or two identifiers separated by `=`.

For examples, `--cfg 'verbose'` or `--cfg 'feature="serde"'`. These correspond to `#[cfg(verbose)]` and `#[cfg(feature = "serde")]` respectively.

`-L`: add a directory to the library search path

The `-L` flag adds a path to search for external crates and libraries.

The kind of search path can optionally be specified with the form `-L KIND=PATH` where `KIND` may be one of:

- `dependency` — Only search for transitive dependencies in this directory.
- `crate` — Only search for this crate's direct dependencies in this directory.
- `native` — Only search for native libraries in this directory.
- `framework` — Only search for macOS frameworks in this directory.
- `all` — Search for all library kinds in this directory. This is the default if `KIND` is not specified.

`-l`: link the generated crate to a native library

This flag allows you to specify linking to a specific native library when building a crate.

The kind of library can optionally be specified with the form `-l KIND=lib` where `KIND` may be one of:

- `dylib` — A native dynamic library.
- `static` — A native static library (such as a `.a` archive).
- `framework` — A macOS framework.

The kind of library can be specified in a `#[link]` attribute. If the kind is not specified in the `link` attribute or on the command-line, it will link a dynamic library if available, otherwise it will use a static library. If the kind is specified on the command-line, it will override the kind specified in a `link` attribute.

The name used in a `link` attribute may be overridden using the form `-l ATTR_NAME:LINK_NAME` where `ATTR_NAME` is the name in the `link` attribute, and `LINK_NAME` is the name of the actual library that will be linked.

`--crate-type`: a list of types of crates for the compiler to emit

This instructs `rustc` on which crate type to build. This flag accepts a comma-separated list of values, and may be specified multiple times. The valid crate types are:

- `lib` — Generates a library kind preferred by the compiler, currently defaults to `rlib`.
- `rlib` — A Rust static library.
- `staticlib` — A native static library.
- `dylib` — A Rust dynamic library.
- `cdylib` — A native dynamic library.
- `bin` — A runnable executable program.
- `proc-macro` — Generates a format suitable for a procedural macro library that may be loaded by the compiler.

The crate type may be specified with the `crate_type` attribute. The `--crate-type` command-line value will override the `crate_type` attribute.

More details may be found in the linkage chapter of the reference.

`--crate-name`: specify the name of the crate being built

This informs `rustc` of the name of your crate.

`--edition`: specify the edition to use

This flag takes a value of `2015` or `2018`. The default is `2015`. More information about editions may be found in the edition guide.

`--emit`: specifies the types of output files to generate

This flag controls the types of output files generated by the compiler. It accepts a comma-separated list of values, and may be specified multiple times. The valid emit kinds are:

- `asm` — Generates a file with the crate's assembly code. The default output filename is `CRATE_NAME.s`.

- `dep-info` — Generates a file with Makefile syntax that indicates all the source files that were loaded to generate the crate. The default output filename is `CRATE_NAME.d`.
- `link` — Generates the crates specified by `--crate-type`. The default output filenames depend on the crate type and platform. This is the default if `--emit` is not specified.
- `llvm-bc` — Generates a binary file containing the LLVM bitcode. The default output filename is `CRATE_NAME.bc`.
- `llvm-ir` — Generates a file containing LLVM IR. The default output filename is `CRATE_NAME.ll`.
- `metadata` — Generates a file containing metadata about the crate. The default output filename is `CRATE_NAME.rmeta`.
- `mir` — Generates a file containing rustc's mid-level intermediate representation. The default output filename is `CRATE_NAME.mir`.
- `obj` — Generates a native object file. The default output filename is `CRATE_NAME.o`.

The output filename can be set with the `-o` flag. A suffix may be added to the filename with the `-C extra-filename` flag. The files are written to the current directory unless the `--out-dir` flag is used. Each emission type may also specify the output filename with the form `KIND=PATH`, which takes precedence over the `-o` flag.

`--print`: print compiler information

This flag prints out various information about the compiler. This flag may be specified multiple times, and the information is printed in the order the flags are specified. Specifying a `--print` flag will usually disable the `--emit` step and will only print the requested information. The valid types of print values are:

- `crate-name` — The name of the crate.
- `file-names` — The names of the files created by the `link` emit kind.
- `sysroot` — Path to the sysroot.
- `target-libdir` - Path to the target libdir.
- `cfg` — List of cfg values. See conditional compilation for more information about cfg values.
- `target-list` — List of known targets. The target may be selected with the `--target` flag.

- `target-cpus` — List of available CPU values for the current target. The target CPU may be selected with the `-C target-cpu=val` flag.
- `target-features` — List of available target features for the current target. Target features may be enabled with the `-C target-feature=val` flag. This flag is unsafe. See known issues for more details.
- `relocation-models` — List of relocation models. Relocation models may be selected with the `-C relocation-model=val` flag.
- `code-models` — List of code models. Code models may be selected with the `-C code-model=val` flag.
- `tls-models` — List of Thread Local Storage models supported. The model may be selected with the `-Z tls-model=val` flag.
- `native-static-libs` — This may be used when creating a `staticlib` crate type. If this is the only flag, it will perform a full compilation and include a diagnostic note that indicates the linker flags to use when linking the resulting static library. The note starts with the text `native-static-libs:` to make it easier to fetch the output.

`-g`: include debug information

A synonym for `-C debuginfo=2`.

`-O`: optimize your code

A synonym for `-C opt-level=2`.

`-o`: filename of the output

This flag controls the output filename.

`--out-dir`: directory to write the output in

The outputted crate will be written to this directory. This flag is ignored if the `-o` flag is used.

`--explain`: provide a detailed explanation of an error message

Each error of `rustc` 's comes with an error code; this will print out a longer explanation of a given error.

`--test`: build a test harness

When compiling this crate, `rustc` will ignore your `main` function and instead produce a test harness.

`--target`: select a target triple to build

This controls which target to produce.

`-W`: set lint warnings

This flag will set which lints should be set to the warn level.

Note: The order of these lint level arguments is taken into account, see lint level via compiler flag for more information.

`-A`: set lint allowed

This flag will set which lints should be set to the allow level.

Note: The order of these lint level arguments is taken into account, see lint level via compiler flag for more information.

`-D`: set lint denied

This flag will set which lints should be set to the deny level.

Note: The order of these lint level arguments is taken into account, see lint level via compiler flag for more information.

`-F`: set lint forbidden

This flag will set which lints should be set to the forbid level.

Note: The order of these lint level arguments is taken into account, see lint level via compiler flag for more information.

`-Z`: set unstable options

This flag will allow you to set unstable options of rustc. In order to set multiple options, the -Z flag can be used multiple times. For example: `rustc -Z verbose -Z time`. Specifying options with -Z is only available on nightly. To view all available options run: `rustc -Z help`.

`--cap-lints`: set the most restrictive lint level

This flag lets you 'cap' lints, for more, see here.

`-C`/`--codegen`: code generation options

This flag will allow you to set codegen options.

`-V`/`--version`: print a version

This flag will print out `rustc` 's version.

`-v`/`--verbose`: use verbose output

This flag, when combined with other flags, makes them produce extra output.

`--extern`: specify where an external library is located

This flag allows you to pass the name and location for an external crate of a direct dependency. Indirect dependencies (dependencies of dependencies) are located using the `-L` flag. The given crate name is added to the extern prelude, which is the same as specifying `extern crate` within the root module. The given crate name does not need to match the name the library was built with.

This flag may be specified multiple times. This flag takes an argument with either of the following formats:

- `CRATENAME=PATH` — Indicates the given crate is found at the given path.
- `CRATENAME` — Indicates the given crate may be found in the search path, such as within the sysroot or via the `-L` flag.

The same crate name may be specified multiple times for different crate types. If both an `rlib` and `dylib` are found, an internal algorithm is used to decide which to use for linking. The `-C prefer-dynamic` flag may be used to influence which is used.

If the same crate name is specified with and without a path, the one with the path is used and the pathless flag has no effect.

`--sysroot`: Override the system root

The "sysroot" is where `rustc` looks for the crates that come with the Rust distribution; this flag allows that to be overridden.

`--error-format`: control how errors are produced

This flag lets you control the format of messages. Messages are printed to stderr. The valid options are:

- `human` — Human-readable output. This is the default.
- `json` — Structured JSON output. See the JSON chapter for more detail.
- `short` — Short, one-line messages.

`--color`: configure coloring of output

This flag lets you control color settings of the output. The valid options are:

- `auto` — Use colors if output goes to a tty. This is the default.
- `always` — Always use colors.
- `never` — Never colorize output.

`--remap-path-prefix`: remap source names in output

Remap source path prefixes in all output, including compiler diagnostics, debug information, macro expansions, etc. It takes a value of the form `FROM=TO` where a path prefix equal to `FROM` is rewritten to the value `TO`. The `FROM` may itself contain an `=` symbol, but the `TO` value may not. This flag may be specified multiple times.

This is useful for normalizing build products, for example by removing the current directory out of pathnames emitted into the object files. The replacement is purely textual, with no consideration of the current system's pathname syntax. For example `--remap-path-prefix foo=bar` will match `foo/lib.rs` but not `./foo/lib.rs`.

`--json`: configure json messages printed by the compiler

When the `--error-format=json` option is passed to rustc then all of the compiler's diagnostic output will be emitted in the form of JSON blobs. The `--json` argument can be used in conjunction with `--error-format=json` to configure what the JSON blobs contain as well as which ones are emitted.

With `--error-format=json` the compiler will always emit any compiler errors as a JSON blob, but the following options are also available to the `--json` flag to customize the output:

- `diagnostic-short` - json blobs for diagnostic messages should use the "short" rendering instead of the normal "human" default. This means that the output of `--error-format=short` will be embedded into the JSON diagnostics instead of the default `--error-format=human` .
- `diagnostic-rendered-ansi` - by default JSON blobs in their `rendered` field will contain a plain text rendering of the diagnostic. This option instead indicates that the diagnostic should have embedded ANSI color codes intended to be used to colorize the message in the manner rustc typically already does for terminal outputs. Note that this is usefully combined with crates like `fwdansi` to translate these ANSI codes on Windows to console commands or `strip-ansi-escapes` if you'd like to optionally remove the ansi colors afterwards.
- `artifacts` - this instructs rustc to emit a JSON blob for each artifact that is emitted. An artifact corresponds to a request from the `--emit` CLI argument, and as soon as the artifact is available on the filesystem a notification will be emitted.

Note that it is invalid to combine the `--json` argument with the `--color` argument, and it is required to combine `--json` with `--error-format=json` .

See the JSON chapter for more detail.

`@path`: load command-line flags from a path

If you specify `@path` on the command-line, then it will open `path` and read command line options from it. These options are one per line; a blank line indicates an empty option. The file can use Unix or Windows style line endings, and must be encoded as UTF-8.

Lints

In software, a "lint" is a tool used to help improve your source code. The Rust compiler contains a number of lints, and when it compiles your code, it will also run the lints. These lints may produce a warning, an error, or nothing at all, depending on how you've configured things.

Here's a small example:

```
$ cat main.rs
fn main() {
    let x = 5;
}
$ rustc main.rs
warning: unused variable: `x`
 --> main.rs:2:9
  |
2 |     let x = 5;
  |         ^
  |
  = note: `#[warn(unused_variables)]` on by default
  = note: to avoid this warning, consider using `_x` instead
```

This is the `unused_variables` lint, and it tells you that you've introduced a variable that you don't use in your code. That's not *wrong*, so it's not an error, but it might be a bug, so you get a warning.

Lint levels

In `rustc`, lints are divided into four *levels*:

1. allow
2. warn
3. deny
4. forbid

Each lint has a default level (explained in the lint listing later in this chapter), and the compiler has a default warning level. First, let's explain what these levels mean, and then we'll talk about configuration.

allow

These lints exist, but by default, do nothing. For example, consider this source:

```
pub fn foo() {}
```

Compiling this file produces no warnings:

```
$ rustc lib.rs --crate-type=lib
$
```

But this code violates the `missing_docs` lint.

These lints exist mostly to be manually turned on via configuration, as we'll talk about later in this section.

warn

The 'warn' lint level will produce a warning if you violate the lint. For example, this code runs afoul of the `unused_variables` lint:

```
pub fn foo() {
    let x = 5;
}
```

This will produce this warning:

```
$ rustc lib.rs --crate-type=lib
warning: unused variable: `x`
 --> lib.rs:2:9
  |
2 |     let x = 5;
  |         ^
  |
  = note: `#[warn(unused_variables)]` on by default
  = note: to avoid this warning, consider using `_x` instead
```

deny

A 'deny' lint produces an error if you violate it. For example, this code runs into the `exceeding_bitshifts` lint.

```
fn main() {
    100u8 << 10;
}
```

```
$ rustc main.rs
error: bitshift exceeds the type's number of bits
 --> main.rs:2:13
  |
2 |     100u8 << 10;
  |     ^^^^^^^^^^^
  |
  = note: `#[deny(exceeding_bitshifts)]` on by default
```

What's the difference between an error from a lint and a regular old error? Lints are configurable via levels, so in a similar way to 'allow' lints, warnings that are 'deny' by default let you allow them. Similarly, you may wish to set up a lint that is `warn` by default to produce an error instead. This lint level gives you that.

forbid

'forbid' is a special lint level that's stronger than 'deny'. It's the same as 'deny' in that a lint at this level will produce an error, but unlike the 'deny' level, the 'forbid' level can not be overridden to be anything lower than an error. However, lint levels may still be capped with `--cap-lints` (see below) so `rustc --cap-lints warn` will make lints set to 'forbid' just warn.

Configuring warning levels

Remember our `missing_docs` example from the 'allow' lint level?

```
$ cat lib.rs
pub fn foo() {}
$ rustc lib.rs --crate-type=lib
$
```

We can configure this lint to operate at a higher level, both with compiler flags, as well as with an attribute in the source code.

You can also "cap" lints so that the compiler can choose to ignore certain lint levels. We'll talk about that last.

Via compiler flag

The `-A`, `-W`, `-D`, and `-F` flags let you turn one or more lints into allowed, warning, deny, or forbid levels, like this:

```
$ rustc lib.rs --crate-type=lib -W missing-docs
warning: missing documentation for crate
 --> lib.rs:1:1
  |
1 | pub fn foo() {}
  | ^^^^^^^^^^^^
  |
  = note: requested on the command line with `-W missing-docs`

warning: missing documentation for a function
 --> lib.rs:1:1
  |
1 | pub fn foo() {}
  | ^^^^^^^^^^^^
```

```
$ rustc lib.rs --crate-type=lib -D missing-docs
error: missing documentation for crate
 --> lib.rs:1:1
  |
1 | pub fn foo() {}
  | ^^^^^^^^^^^^
  |
  = note: requested on the command line with `-D missing-docs`

error: missing documentation for a function
 --> lib.rs:1:1
  |
1 | pub fn foo() {}
  | ^^^^^^^^^^^^

error: aborting due to 2 previous errors
```

You can also pass each flag more than once for changing multiple lints:

```
$ rustc lib.rs --crate-type=lib -D missing-docs -D unused-variables
```

And of course, you can mix these four flags together:

```
$ rustc lib.rs --crate-type=lib -D missing-docs -A unused-variables
```

The order of these command line arguments is taken into account. The following allows the `unused-variables` lint, because it is the last argument for that lint:

```
$ rustc lib.rs --crate-type=lib -D unused-variables -A unused-variables
```

You can make use of this behavior by overriding the level of one specific lint out of a group of lints. The following example denies all the lints in the `unused` group, but explicitly allows the `unused-variables` lint in that group (forbid still trumps everything regardless of ordering):

```
$ rustc lib.rs --crate-type=lib -D unused -A unused-variables
```

Via an attribute

You can also modify the lint level with a crate-wide attribute:

```
$ cat lib.rs
#![warn(missing_docs)]

pub fn foo() {}
$ rustc lib.rs --crate-type=lib
warning: missing documentation for crate
 --> lib.rs:1:1
  |
1 | / #![warn(missing_docs)]
2 | |
3 | | pub fn foo() {}
  | |_______________^
  |
note: lint level defined here
 --> lib.rs:1:9
  |
1 | #![warn(missing_docs)]
  |         ^^^^^^^^^^^^

warning: missing documentation for a function
 --> lib.rs:3:1
  |
3 | pub fn foo() {}
  | ^^^^^^^^^^^^
```

All four, `warn`, `allow`, `deny`, and `forbid` all work this way.

You can also pass in multiple lints per attribute:

```
#![warn(missing_docs, unused_variables)]

pub fn foo() {}
```

And use multiple attributes together:

```
#![warn(missing_docs)]
#![deny(unused_variables)]

pub fn foo() {}
```

Capping lints

`rustc` supports a flag, `--cap-lints LEVEL` that sets the "lint cap level." This is the maximum level for all lints. So for example, if we take our code sample from the "deny" lint level above:

```
fn main() {
    100u8 << 10;
}
```

And we compile it, capping lints to warn:

```
$ rustc lib.rs --cap-lints warn
warning: bitshift exceeds the type's number of bits
 --> lib.rs:2:5
  |
2 |     100u8 << 10;
  |     ^^^^^^^^^^^
  |
  = note: `#[warn(exceeding_bitshifts)]` on by default

warning: this expression will panic at run-time
 --> lib.rs:2:5
  |
2 |     100u8 << 10;
  |     ^^^^^^^^^^^ attempt to shift left with overflow
```

It now only warns, rather than errors. We can go further and allow all lints:

```
$ rustc lib.rs --cap-lints allow
$
```

This feature is used heavily by Cargo; it will pass `--cap-lints allow` when compiling your dependencies, so that if they have any warnings, they do not pollute the output of your build.

Lint Groups

`rustc` has the concept of a "lint group", where you can toggle several warnings through one name.

For example, the `nonstandard-style` lint sets `non-camel-case-types`, `non-snake-case`, and `non-upper-case-globals` all at once. So these are equivalent:

```
$ rustc -D nonstandard-style
$ rustc -D non-camel-case-types -D non-snake-case -D non-upper-case-
globals
```

Here's a list of each lint group, and the lints that they are made up of:

group	description	lints
nonstandard-style	Violation of standard naming conventions	non-camel-case-types, non-snake-case, non-upper-case-globals
warnings	all lints that would be issuing warnings	all lints that would be issuing warnings
edition-2018	Lints that will be turned into errors in Rust 2018	tyvar-behind-raw-pointer
rust-2018-idioms	Lints to nudge you toward idiomatic features of Rust 2018	bare-trait-object, unreachable-pub
unused	These lints detect things being declared but not used	unused-imports, unused-variables, unused-assignments, dead-code, unused-mut, unreachable-code, unreachable-patterns, unused-must-use, unused-unsafe, path-statements, unused-attributes, unused-macros, unused-allocation, unused-doc-comment, unused-extern-crates, unused-features, unused-parens
future-incompatible	Lints that detect code that has future-compatibility problems	private-in-public, pub-use-of-private-extern-crate, patterns-in-fns-without-body, safe-extern-statics, invalid-type-param-default, legacy-directory-ownership, legacy-imports, legacy-constructor-visibility, missing-fragment-specifier, illegal-floating-point-literal-pattern, anonymous-parameters, parenthesized-params-

group	description	lints
		in-types-and-modules, late-bound-lifetime-arguments, safe-packed-borrows, tyvar-behind-raw-pointer, unstable-name-collision

Additionally, there's a `bad-style` lint group that's a deprecated alias for `nonstandard-style`.

Finally, you can also see the table above by invoking `rustc -W help`. This will give you the exact values for the specific compiler you have installed.

Lint listing

This section lists out all of the lints, grouped by their default lint levels.

You can also see this list by running `rustc -W help`.

Allowed-by-default lints

These lints are all set to the 'allow' level by default. As such, they won't show up unless you set them to a higher lint level with a flag or attribute.

anonymous-parameters

This lint detects anonymous parameters. Some example code that triggers this lint:

```rust
trait Foo {
    fn foo(usize);
}
```

When set to 'deny', this will produce:

```
error: use of deprecated anonymous parameter
 --> src/lib.rs:5:11
  |
5 |     fn foo(usize);
  |           ^
  |
  = warning: this was previously accepted by the compiler but is being
phased out; it will become a hard error in a future release!
  = note: for more information, see issue #41686 <https://github.com
/rust-lang/rust/issues/41686>
```

This syntax is mostly a historical accident, and can be worked around quite easily:

```
trait Foo {
    fn foo(_: usize);
}
```

bare-trait-object

This lint suggests using `dyn Trait` for trait objects. Some example code that triggers this lint:

```
#![feature(dyn_trait)]

trait Trait { }

fn takes_trait_object(_: Box<Trait>) {
}
```

When set to 'deny', this will produce:

```
error: trait objects without an explicit `dyn` are deprecated
 --> src/lib.rs:7:30
  |
7 | fn takes_trait_object(_: Box<Trait>) {
  |                              ^^^^^ help: use `dyn`: `dyn Trait`
  |
```

To fix it, do as the help message suggests:

```
#![feature(dyn_trait)]
#![deny(bare_trait_objects)]

trait Trait { }

fn takes_trait_object(_: Box<dyn Trait>) {
}
```

box-pointers

This lints use of the Box type. Some example code that triggers this lint:

```
struct Foo {
    x: Box<isize>,
}
```

When set to 'deny', this will produce:

```
error: type uses owned (Box type) pointers: std::boxed::Box<isize>
 --> src/lib.rs:6:5
  |
6 |     x: Box<isize> //~ ERROR type uses owned
  |     ^^^^^^^^^^^^^
  |
```

This lint is mostly historical, and not particularly useful. `Box<T>` used to be built into the language, and the only way to do heap allocation. Today's Rust can call into other allocators, etc.

elided-lifetime-in-path

This lint detects the use of hidden lifetime parameters. Some example code that triggers this lint:

```rust
struct Foo<'a> {
    x: &'a u32
}

fn foo(x: &Foo) {
}
```

When set to 'deny', this will produce:

```
error: hidden lifetime parameters are deprecated, try `Foo<'_>`
 --> src/lib.rs:5:12
  |
5 | fn foo(x: &Foo) {
  |            ^^^
  |
```

Lifetime elision elides this lifetime, but that is being deprecated.

missing-copy-implementations

This lint detects potentially-forgotten implementations of `Copy`. Some example code that triggers this lint:

```rust
pub struct Foo {
    pub field: i32
}
```

When set to 'deny', this will produce:

```
error: type could implement `Copy`; consider adding `impl Copy`
 --> src/main.rs:3:1
  |
3 | / pub struct Foo { //~ ERROR type could implement `Copy`; consider
adding `impl Copy`
4 | |     pub field: i32
5 | | }
  | |_^
  |
```

You can fix the lint by deriving `Copy`.

This lint is set to 'allow' because this code isn't bad; it's common to write newtypes like this specifically so that a `Copy` type is no longer `Copy`.

missing-debug-implementations

This lint detects missing implementations of `fmt::Debug`. Some example code that triggers this lint:

```rust
pub struct Foo;
```

When set to 'deny', this will produce:

```
error: type does not implement `fmt::Debug`; consider adding
`#[derive(Debug)]` or a manual implementation
 --> src/main.rs:3:1
  |
3 | pub struct Foo;
  | ^^^^^^^^^^^^^^^
  |
```

You can fix the lint by deriving `Debug`.

missing-docs

This lint detects missing documentation for public items. Some example code that triggers this lint:

```rust
pub fn foo() {}
```

When set to 'deny', this will produce:

```
error: missing documentation for crate
 --> src/main.rs:1:1
  |
1 | / #![deny(missing_docs)]
2 | |
3 | | pub fn foo() {}
4 | |
5 | | fn main() {}
  | |____________^
  |

error: missing documentation for a function
 --> src/main.rs:3:1
  |
3 | pub fn foo() {}
  | ^^^^^^^^^^^^
```

To fix the lint, add documentation to all items.

single-use-lifetimes

This lint detects lifetimes that are only used once. Some example code that triggers this lint:

```
struct Foo<'x> {
    x: &'x u32
}
```

When set to 'deny', this will produce:

```
error: lifetime name `'x` only used once
 --> src/main.rs:3:12
  |
3 | struct Foo<'x> {
  |            ^^
  |
```

trivial-casts

This lint detects trivial casts which could be removed. Some example code that triggers this lint:

```
let x: &u32 = &42;
let _ = x as *const u32;
```

When set to 'deny', this will produce:

```
error: trivial cast: `&u32` as `*const u32`. Cast can be replaced by
coercion, this might require type ascription or a temporary variable
 --> src/main.rs:5:13
  |
5 |     let _ = x as *const u32;
  |             ^^^^^^^^^^^^^^^
  |
note: lint level defined here
 --> src/main.rs:1:9
  |
1 | #![deny(trivial_casts)]
  |         ^^^^^^^^^^^^^
```

trivial-numeric-casts

This lint detects trivial casts of numeric types which could be removed. Some example code that triggers this lint:

```
let x = 42i32 as i32;
```

When set to 'deny', this will produce:

```
error: trivial numeric cast: `i32` as `i32`. Cast can be replaced by
coercion, this might require type ascription or a temporary variable
 --> src/main.rs:4:13
  |
4 |     let x = 42i32 as i32;
  |             ^^^^^^^^^^^^
  |
```

unreachable-pub

This lint triggers for `pub` items not reachable from the crate root. Some example code that triggers this lint:

```
mod foo {
    pub mod bar {

    }
}
```

When set to 'deny', this will produce:

```
error: unreachable `pub` item
 --> src/main.rs:4:5
  |
4 |     pub mod bar {
  |     ---^^^^^^^^^
  |     |
  |     help: consider restricting its visibility: `pub(crate)`
  |
```

unsafe-code

This lint catches usage of `unsafe` code. Some example code that triggers this lint:

```
fn main() {
    unsafe {

    }
}
```

When set to 'deny', this will produce:

```
error: usage of an `unsafe` block
 --> src/main.rs:4:5
  |
4 | /     unsafe {
5 | |
6 | |     }
  | |_____^
  |
```

unstable-features

This lint is deprecated and no longer used.

unused-extern-crates

This lint guards against `extern crate` items that are never used. Some example code that triggers this lint:

```rust
extern crate semver;
```

When set to 'deny', this will produce:

```
error: unused extern crate
 --> src/main.rs:3:1
  |
3 | extern crate semver;
  | ^^^^^^^^^^^^^^^^^^^^
  |
```

unused-import-braces

This lint catches unnecessary braces around an imported item. Some example code that triggers this lint:

```rust
use test::{A};

pub mod test {
    pub struct A;
}
```

When set to 'deny', this will produce:

```
error: braces around A is unnecessary
 --> src/main.rs:3:1
  |
3 | use test::{A};
  | ^^^^^^^^^^^^^^
  |
```

To fix it, `use test::A;`

unused-qualifications

This lint detects unnecessarily qualified names. Some example code that triggers this lint:

```rust
mod foo {
    pub fn bar() {}
}

fn main() {
    use foo::bar;
    foo::bar();
}
```

When set to 'deny', this will produce:

```
error: unnecessary qualification
 --> src/main.rs:9:5
  |
9 |     foo::bar();
  |     ^^^^^^^^
  |
```

You can call `bar()` directly, without the `foo::`.

unused-results

This lint checks for the unused result of an expression in a statement. Some example code that triggers this lint:

```rust
fn foo<T>() -> T { panic!() }

fn main() {
    foo::<usize>();
}
```

When set to 'deny', this will produce:

```
error: unused result
 --> src/main.rs:6:5
  |
6 |     foo::<usize>();
  |     ^^^^^^^^^^^^^^^
  |
```

variant-size-differences

This lint detects enums with widely varying variant sizes. Some example code that triggers this lint:

```
enum En {
    V0(u8),
    VBig([u8; 1024]),
}
```

When set to 'deny', this will produce:

```
error: enum variant is more than three times larger (1024 bytes) than
the next largest
 --> src/main.rs:5:5
  |
5 |     VBig([u8; 1024]),   //~ ERROR variant is more than three times
larger
  |     ^^^^^^^^^^^^^^^^
  |
```

Warn-by-default lints

These lints are all set to the 'warn' level by default.

const-err

This lint detects an erroneous expression while doing constant evaluation. Some example code that triggers this lint:

```
let b = 200u8 + 200u8;
```

This will produce:

```
warning: attempt to add with overflow
 --> src/main.rs:2:9
  |
2 | let b = 200u8 + 200u8;
  |         ^^^^^^^^^^^^^
  |
```

dead-code

This lint detects unused, unexported items. Some example code that triggers this lint:

```
fn foo() {}
```

This will produce:

```
warning: function is never used: `foo`
 --> src/lib.rs:2:1
  |
2 | fn foo() {}
  | ^^^^^^^^
  |
```

deprecated

This lint detects use of deprecated items. Some example code that triggers this lint:

```
#[deprecated]
fn foo() {}

fn bar() {
    foo();
}
```

This will produce:

```
warning: use of deprecated item 'foo'
 --> src/lib.rs:7:5
  |
7 |     foo();
  |     ^^^
  |
```

illegal-floating-point-literal-pattern

This lint detects floating-point literals used in patterns. Some example code that triggers this lint:

```
let x = 42.0;

match x {
    5.0 => {},
    _ => {},
}
```

This will produce:

```
warning: floating-point literals cannot be used in patterns
 --> src/main.rs:4:9
  |
4 |         5.0 => {},
  |         ^^^
  |
  = note: `#[warn(illegal_floating_point_literal_pattern)]` on by
default
  = warning: this was previously accepted by the compiler but is being
phased out; it will become a hard error in a future release!
  = note: for more information, see issue #41620 <https://github.com
/rust-lang/rust/issues/41620>
```

improper-ctypes

This lint detects proper use of libc types in foreign modules. Some example code that triggers this lint:

```
extern "C" {
    static STATIC: String;
}
```

This will produce:

```
warning: found struct without foreign-function-safe representation
annotation in foreign module, consider adding a `#[repr(C)]` attribute
to the type
 --> src/main.rs:2:20
  |
2 |     static STATIC: String;
  |                    ^^^^^^
  |
```

late-bound-lifetime-arguments

This lint detects generic lifetime arguments in path segments with late bound lifetime parameters. Some example code that triggers this lint:

```
struct S;

impl S {
    fn late<'a, 'b>(self, _: &'a u8, _: &'b u8) {}
}

fn main() {
    S.late::<'static>(&0, &0);
}
```

This will produce:

```
warning: cannot specify lifetime arguments explicitly if late bound
lifetime parameters are present
 --> src/main.rs:8:14
  |
4 |     fn late<'a, 'b>(self, _: &'a u8, _: &'b u8) {}
  |             -- the late bound lifetime parameter is introduced here
...
8 |     S.late::<'static>(&0, &0);
  |              ^^^^^^^
  |
  = note: `#[warn(late_bound_lifetime_arguments)]` on by default
  = warning: this was previously accepted by the compiler but is being
phased out; it will become a hard error in a future release!
  = note: for more information, see issue #42868 <https://github.com
/rust-lang/rust/issues/42868>
```

non-camel-case-types

This lint detects types, variants, traits and type parameters that don't have camel case names. Some example code that triggers this lint:

```
struct s;
```

This will produce:

```
warning: type `s` should have a camel case name such as `S`
 --> src/main.rs:1:1
  |
1 | struct s;
  | ^^^^^^^^^
  |
```

non-shorthand-field-patterns

This lint detects using `Struct { x: x }` instead of `Struct { x }` in a pattern. Some example code that triggers this lint:

```rust
struct Point {
    x: i32,
    y: i32,
}


fn main() {
    let p = Point {
        x: 5,
        y: 5,
    };

    match p {
        Point { x: x, y: y } => (),
    }
}
```

This will produce:

```text
warning: the `x:` in this pattern is redundant
  --> src/main.rs:14:17
   |
14 |         Point { x: x, y: y } => (),
   |                 --^^
   |                 |
   |                 help: remove this
   |

warning: the `y:` in this pattern is redundant
  --> src/main.rs:14:23
   |
14 |         Point { x: x, y: y } => (),
   |                       --^^
   |                       |
   |                       help: remove this
   |
```

non-snake-case

This lint detects variables, methods, functions, lifetime parameters and modules that don't have snake case names. Some example code that triggers this lint:

```rust
let X = 5;
```

This will produce:

```
warning: variable `X` should have a snake case name such as `x`
 --> src/main.rs:2:9
  |
2 |     let X = 5;
  |         ^
  |
```

non-upper-case-globals

This lint detects static constants that don't have uppercase identifiers. Some example code that triggers this lint:

```rust
static x: i32 = 5;
```

This will produce:

```
warning: static variable `x` should have an upper case name such as `X`
 --> src/main.rs:1:1
  |
1 | static x: i32 = 5;
  | ^^^^^^^^^^^^^^^^^^
  |
```

no-mangle-generic-items

This lint detects generic items must be mangled. Some example code that triggers this lint:

```rust
#[no_mangle]
fn foo<T>(t: T) {

}
```

This will produce:

```
warning: functions generic over types must be mangled
 --> src/main.rs:2:1
  |
1 |   #[no_mangle]
  |   ------------ help: remove this attribute
2 | / fn foo<T>(t: T) {
3 | |
4 | | }
  | |_^
  |
```

path-statements

This lint detects path statements with no effect. Some example code that triggers this lint:

```
let x = 42;

x;
```

This will produce:

```
warning: path statement with no effect
 --> src/main.rs:3:5
  |
3 |     x;
  |     ^^
  |
```

private-in-public

This lint detects private items in public interfaces not caught by the old implementation. Some example code that triggers this lint:

```rust
pub trait Trait {
    type A;
}

pub struct S;

mod foo {
    struct Z;

    impl ::Trait for ::S {
        type A = Z;
    }
}
```

This will produce:

```text
error[E0446]: private type `foo::Z` in public interface
  --> src/main.rs:11:9
   |
11 |         type A = Z;
   |         ^^^^^^^^^^^ can't leak private type
```

private-no-mangle-fns

This lint detects functions marked `#[no_mangle]` that are also private. Given that private functions aren't exposed publicly, and `#[no_mangle]` controls the public symbol, this combination is erroneous. Some example code that triggers this lint:

```rust
#[no_mangle]
fn foo() {}
```

This will produce:

```text
warning: function is marked `#[no_mangle]`, but not exported
 --> src/main.rs:2:1
  |
2 | fn foo() {}
  | -^^^^^^^^^^
  | |
  | help: try making it public: `pub`
  |
```

To fix this, either make it public or remove the `#[no_mangle]` .

private-no-mangle-statics

This lint detects any statics marked `#[no_mangle]` that are private. Given that private statics aren't exposed publicly, and `#[no_mangle]` controls the public symbol, this combination is erroneous. Some example code that triggers this lint:

```rust
#[no_mangle]
static X: i32 = 4;
```

This will produce:

```
warning: static is marked `#[no_mangle]`, but not exported
 --> src/main.rs:2:1
  |
2 | static X: i32 = 4;
  | -^^^^^^^^^^^^^^^^^
  | |
  | help: try making it public: `pub`
  |
```

To fix this, either make it public or remove the `#[no_mangle]` .

renamed-and-removed-lints

This lint detects lints that have been renamed or removed. Some example code that triggers this lint:

```rust
#![deny(raw_pointer_derive)]
```

This will produce:

```
warning: lint raw_pointer_derive has been removed: using derive with raw
pointers is ok
 --> src/main.rs:1:9
  |
1 | #![deny(raw_pointer_derive)]
  |         ^^^^^^^^^^^^^^^^^^
  |
```

To fix this, either remove the lint or use the new name.

safe-packed-borrows

This lint detects borrowing a field in the interior of a packed structure with alignment other than 1. Some example code that triggers this lint:

```
#[repr(packed)]
pub struct Unaligned<T>(pub T);

pub struct Foo {
    start: u8,
    data: Unaligned<u32>,
}

fn main() {
    let x = Foo { start: 0, data: Unaligned(1) };
    let y = &x.data.0;
}
```

This will produce:

```
warning: borrow of packed field requires unsafe function or block (error
E0133)
  --> src/main.rs:11:13
   |
11 |     let y = &x.data.0;
   |             ^^^^^^^^^
   |
   = note: `#[warn(safe_packed_borrows)]` on by default
   = warning: this was previously accepted by the compiler but is being
phased out; it will become a hard error in a future release!
   = note: for more information, see issue #46043 <https://github.com
/rust-lang/rust/issues/46043>
```

stable-features

This lint detects a `#[feature]` attribute that's since been made stable. Some example code that triggers this lint:

```
#![feature(test_accepted_feature)]
```

This will produce:

```
warning: this feature has been stable since 1.0.0. Attribute no longer
needed
 --> src/main.rs:1:12
  |
1 | #![feature(test_accepted_feature)]
  |            ^^^^^^^^^^^^^^^^^^^^^
  |
```

To fix, simply remove the `#![feature]` attribute, as it's no longer needed.

type-alias-bounds

This lint detects bounds in type aliases. These are not currently enforced. Some example code that triggers this lint:

```
#[allow(dead_code)]
type SendVec<T: Send> = Vec<T>;
```

This will produce:

```
warning: bounds on generic parameters are not enforced in type aliases
 --> src/lib.rs:2:17
  |
2 | type SendVec<T: Send> = Vec<T>;
  |                 ^^^^
  |
  = note: `#[warn(type_alias_bounds)]` on by default
  = help: the bound will not be checked when the type alias is used, and
should be removed
```

tyvar-behind-raw-pointer

This lint detects raw pointer to an inference variable. Some example code that triggers this lint:

```
let data = std::ptr::null();
let _ = &data as *const *const ();

if data.is_null() {}
```

This will produce:

```
warning: type annotations needed
 --> src/main.rs:4:13
  |
4 |     if data.is_null() {}
  |             ^^^^^^^
  |
  = note: `#[warn(tyvar_behind_raw_pointer)]` on by default
  = warning: this was previously accepted by the compiler but is being
phased out; it will become a hard error in the 2018 edition!
  = note: for more information, see issue #46906 <https://github.com
/rust-lang/rust/issues/46906>
```

unconditional-recursion

This lint detects functions that cannot return without calling themselves. Some example code that triggers this lint:

```
fn foo() {
    foo();
}
```

This will produce:

```
warning: function cannot return without recursing
 --> src/main.rs:1:1
  |
1 | fn foo() {
  | ^^^^^^^^ cannot return without recursing
2 |     foo();
  |     ----- recursive call site
  |
```

unknown-lints

This lint detects unrecognized lint attribute. Some example code that triggers this lint:

```
#[allow(not_a_real_lint)]
```

This will produce:

```
warning: unknown lint: `not_a_real_lint`
 --> src/main.rs:1:10
  |
1 | #![allow(not_a_real_lint)]
  |          ^^^^^^^^^^^^^^^
  |
```

unreachable-code

This lint detects unreachable code paths. Some example code that triggers this lint:

```
panic!("we never go past here!");

let x = 5;
```

This will produce:

```
warning: unreachable statement
 --> src/main.rs:4:5
  |
4 |     let x = 5;
  |     ^^^^^^^^^^
  |
```

unreachable-patterns

This lint detects unreachable patterns. Some example code that triggers this lint:

```rust
let x = 5;
match x {
    y => (),
    5 => (),
}
```

This will produce:

```
warning: unreachable pattern
 --> src/main.rs:5:5
  |
5 |     5 => (),
  |     ^
  |
```

The `y` pattern will always match, so the five is impossible to reach. Remember, match arms match in order, you probably wanted to put the `5` case above the `y` case.

unstable-name-collision

This lint detects that you've used a name that the standard library plans to add in the future, which means that your code may fail to compile without additional type annotations in the future. Either rename, or add those annotations now.

unused-allocation

This lint detects unnecessary allocations that can be eliminated.

unused-assignments

This lint detects assignments that will never be read. Some example code that triggers this lint:

```rust
let mut x = 5;
x = 6;
```

This will produce:

```
warning: value assigned to `x` is never read
 --> src/main.rs:4:5
  |
4 |     x = 6;
  |     ^
  |
```

unused-attributes

This lint detects attributes that were not used by the compiler. Some example code that triggers this lint:

```rust
#![macro_export]
```

This will produce:

```
warning: unused attribute
 --> src/main.rs:1:1
  |
1 | #![macro_export]
  | ^^^^^^^^^^^^^^^^
  |
```

unused-comparisons

This lint detects comparisons made useless by limits of the types involved. Some example code that triggers this lint:

```
fn foo(x: u8) {
    x >= 0;
}
```

This will produce:

```
warning: comparison is useless due to type limits
 --> src/main.rs:6:5
  |
6 |     x >= 0;
  |     ^^^^^^
  |
```

unused-doc-comment

This lint detects doc comments that aren't used by rustdoc. Some example code that triggers this lint:

```
/// docs for x
let x = 12;
```

This will produce:

```
warning: doc comment not used by rustdoc
 --> src/main.rs:2:5
  |
2 |     /// docs for x
  |     ^^^^^^^^^^^^^^
  |
```

unused-features

This lint detects unused or unknown features found in crate-level `#[feature]` directives. To fix this, simply remove the feature flag.

unused-imports

This lint detects imports that are never used. Some example code that triggers this lint:

```
use std::collections::HashMap;
```

This will produce:

```
warning: unused import: `std::collections::HashMap`
 --> src/main.rs:1:5
  |
1 | use std::collections::HashMap;
  |     ^^^^^^^^^^^^^^^^^^^^^^^^^
  |
```

unused-macros

This lint detects macros that were not used. Some example code that triggers this lint:

```
macro_rules! unused {
    () => {};
}

fn main() {
}
```

This will produce:

```
warning: unused macro definition
 --> src/main.rs:1:1
  |
1 | / macro_rules! unused {
2 | |     () => {};
3 | | }
  | |_^
  |
```

unused-must-use

This lint detects unused result of a type flagged as `#[must_use]` . Some example code that triggers this lint:

```
fn returns_result() -> Result<(), ()> {
    Ok(())
}

fn main() {
    returns_result();
}
```

This will produce:

```
warning: unused `std::result::Result` that must be used
 --> src/main.rs:6:5
  |
6 |     returns_result();
  |     ^^^^^^^^^^^^^^^^^
  |
```

unused-mut

This lint detects mut variables which don't need to be mutable. Some example code that triggers this lint:

```
let mut x = 5;
```

This will produce:

```
warning: variable does not need to be mutable
 --> src/main.rs:2:9
  |
2 |     let mut x = 5;
  |         ----^
  |         |
  |         help: remove this `mut`
  |
```

unused-parens

This lint detects `if`, `match`, `while` and `return` with parentheses; they do not need them. Some example code that triggers this lint:

```rust
if(true) {}
```

This will produce:

```
warning: unnecessary parentheses around `if` condition
 --> src/main.rs:2:7
  |
2 |     if(true) {}
  |       ^^^^^^ help: remove these parentheses
  |
```

unused-unsafe

This lint detects unnecessary use of an `unsafe` block. Some example code that triggers this lint:

```rust
unsafe {}
```

This will produce:

```
warning: unnecessary `unsafe` block
 --> src/main.rs:2:5
  |
2 |     unsafe {}
  |     ^^^^^^ unnecessary `unsafe` block
  |
```

unused-variables

This lint detects variables which are not used in any way. Some example code that triggers this lint:

```
let x = 5;
```

This will produce:

```
warning: unused variable: `x`
 --> src/main.rs:2:9
  |
2 |     let x = 5;
  |         ^ help: consider using `_x` instead
  |
```

warnings

This lint is a bit special; by changing its level, you change every other warning that would produce a warning to whatever value you'd like:

```
#![deny(warnings)]
```

As such, you won't ever trigger this lint in your code directly.

while-true

This lint detects `while true { }`. Some example code that triggers this lint:

```
while true {

}
```

This will produce:

```
warning: denote infinite loops with `loop { ... }`
 --> src/main.rs:2:5
  |
2 |     while true {
  |     ^^^^^^^^^^ help: use `loop`
  |
```

Deny-by-default lints

These lints are all set to the 'deny' level by default.

exceeding-bitshifts

This lint detects that a shift exceeds the type's number of bits. Some example code that triggers this lint:

```
1_i32 << 32;
```

This will produce:

```
error: bitshift exceeds the type's number of bits
 --> src/main.rs:2:5
  |
2 |     1_i32 << 32;
  |     ^^^^^^^^^^^
  |
```

invalid-type-param-default

This lint detects type parameter default erroneously allowed in invalid location. Some example code that triggers this lint:

```
fn foo<T=i32>(t: T) {}
```

This will produce:

```
error: defaults for type parameters are only allowed in `struct`,
`enum`, `type`, or `trait` definitions.
 --> src/main.rs:4:8
  |
4 | fn foo<T=i32>(t: T) {}
  |        ^
  |
  = note: `#[deny(invalid_type_param_default)]` on by default
  = warning: this was previously accepted by the compiler but is being
phased out; it will become a hard error in a future release!
  = note: for more information, see issue #36887 <https://github.com
/rust-lang/rust/issues/36887>
```

missing-fragment-specifier

The missing_fragment_specifier warning is issued when an unused pattern in a `macro_rules!` macro definition has a meta-variable (e.g. `$e`) that is not followed by a fragment specifier (e.g. `:expr`).

This warning can always be fixed by removing the unused pattern in the `macro_rules!` macro definition.

mutable-transmutes

This lint catches transmuting from `&T` to `&mut T` because it is undefined behavior. Some example code that triggers this lint:

```
unsafe {
    let y = std::mem::transmute::<&i32, &mut i32>(&5);
}
```

This will produce:

```
error: mutating transmuted &mut T from &T may cause undefined behavior,
consider instead using an UnsafeCell
 --> src/main.rs:3:17
  |
3 |         let y = std::mem::transmute::<&i32, &mut i32>(&5);
  |                 ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  |
```

no-mangle-const-items

This lint detects any `const` items with the `#[no_mangle]` attribute. Constants do not have their symbols exported, and therefore, this probably means you meant to use a `static`, not a `const`. Some example code that triggers this lint:

```rust
#[no_mangle]
const FOO: i32 = 5;
```

This will produce:

```
error: const items should never be `#[no_mangle]`
 --> src/main.rs:3:1
  |
3 | const FOO: i32 = 5;
  | -----^^^^^^^^^^^^^^
  | |
  | help: try a static value: `pub static`
  |
```

overflowing-literals

This lint detects literal out of range for its type. Some example code that triggers this lint:

```rust
let x: u8 = 1000;
```

This will produce:

```
error: literal out of range for u8
 --> src/main.rs:2:17
  |
2 |     let x: u8 = 1000;
  |                 ^^^^
  |
```

patterns-in-fns-without-body

This lint detects patterns in functions without body were that were previously erroneously allowed. Some example code that triggers this lint:

```
trait Trait {
    fn foo(mut arg: u8);
}
```

This will produce:

```
warning: patterns aren't allowed in methods without bodies
 --> src/main.rs:2:12
  |
2 |     fn foo(mut arg: u8);
  |            ^^^^^^^
  |
  = note: `#[warn(patterns_in_fns_without_body)]` on by default
  = warning: this was previously accepted by the compiler but is being
phased out; it will become a hard error in a future release!
  = note: for more information, see issue #35203 <https://github.com
/rust-lang/rust/issues/35203>
```

To fix this, remove the pattern; it can be used in the implementation without being used in the definition. That is:

```
trait Trait {
    fn foo(arg: u8);
}

impl Trait for i32 {
    fn foo(mut arg: u8) {

    }
}
```

pub-use-of-private-extern-crate

This lint detects a specific situation of re-exporting a private `extern crate`;

unknown-crate-types

This lint detects an unknown crate type found in a `#[crate_type]` directive. Some example code that triggers this lint:

```
#![crate_type="lol"]
```

This will produce:

```
error: invalid `crate_type` value
 --> src/lib.rs:1:1
  |
1 | #![crate_type="lol"]
  | ^^^^^^^^^^^^^^^^^^^^
  |
```

const-err

This lint detects expressions that will always panic at runtime and would be an error in a `const` context.

```
let _ = [0; 4][4];
```

This will produce:

```
error: index out of bounds: the len is 4 but the index is 4
 --> src/lib.rs:1:9
  |
1 | let _ = [0; 4][4];
  |         ^^^^^^^^^
  |
```

order-dependent-trait-objects

This lint detects a trait coherency violation that would allow creating two trait impls for the same dynamic trait object involving marker traits.

Codegen options

All of these options are passed to `rustc` via the `-C` flag, short for "codegen." You can see a version of this list for your exact compiler by running `rustc -C help`.

ar

This option is deprecated and does nothing.

code-model

This option lets you choose which code model to use.
Code models put constraints on address ranges that the program and its symbols may use.
With smaller address ranges machine instructions may be able to use use more compact addressing modes.

The specific ranges depend on target architectures and addressing modes available to them.
For x86 more detailed description of its code models can be found in System V Application Binary Interface specification.

Supported values for this option are:

- `tiny` - Tiny code model.
- `small` - Small code model. This is the default model for majority of supported targets.
- `kernel` - Kernel code model.
- `medium` - Medium code model.
- `large` - Large code model.

Supported values can also be discovered by running `rustc --print code-models`.

codegen-units

This flag controls how many code generation units the crate is split into. It takes an integer greater than 0.

When a crate is split into multiple codegen units, LLVM is able to process them in parallel. Increasing parallelism may speed up compile times, but may also produce slower code. Setting this to 1 may improve the performance of generated code, but may be slower to compile.

The default value, if not specified, is 16 for non-incremental builds. For incremental builds the default is 256 which allows caching to be more granular.

debug-assertions

This flag lets you turn `cfg(debug_assertions)` conditional compilation on or off. It takes one of the following values:

- `y`, `yes`, `on`, or no value: enable debug-assertions.
- `n`, `no`, or `off`: disable debug-assertions.

If not specified, debug assertions are automatically enabled only if the opt-level is 0.

debuginfo

This flag controls the generation of debug information. It takes one of the following values:

- `0`: no debug info at all (the default).
- `1`: line tables only.
- `2`: full debug info.

Note: The `-g` flag is an alias for `-C debuginfo=2`.

default-linker-libraries

This flag controls whether or not the linker includes its default libraries. It takes one of the following values:

- `y`, `yes`, `on`, or no value: include default libraries (the default).
- `n`, `no`, or `off`: exclude default libraries.

For example, for gcc flavor linkers, this issues the `-nodefaultlibs` flag to the linker.

embed-bitcode

This flag controls whether or not the compiler embeds LLVM bitcode into object files. It takes one of the following values:

- `y`, `yes`, `on`, or no value: put bitcode in rlibs (the default).
- `n`, `no`, or `off`: omit bitcode from rlibs.

LLVM bitcode is required when rustc is performing link-time optimization (LTO). It is also required on some targets like iOS ones where vendors look for LLVM bitcode. Embedded bitcode will appear in rustc-generated object files inside of a section whose name is defined by the target platform. Most of the time this is `.llvmbc`.

The use of `-C embed-bitcode=no` can significantly improve compile times and reduce generated file sizes if your compilation does not actually need bitcode (e.g. if you're not compiling for iOS or you're not performing LTO). For these reasons, Cargo uses `-C embed-bitcode=no` whenever possible. Likewise, if you are building directly with `rustc` we recommend using `-C embed-bitcode=no` whenever you are not using LTO.

If combined with `-C lto`, `-C embed-bitcode=no` will cause `rustc` to abort at start-up, because the combination is invalid.

> **Note**: if you're building Rust code with LTO then you probably don't even need the `embed-bitcode` option turned on. You'll likely want to use `-Clinker-plugin-lto` instead which skips generating object files entirely and simply replaces object files with LLVM bitcode. The only purpose for

`-Cembed-bitcode` is when you're generating an rlib that is both being used with and without LTO. For example Rust's standard library ships with embedded bitcode since users link to it both with and without LTO.

This also may make you wonder why the default is `yes` for this option. The reason for that is that it's how it was for rustc 1.44 and prior. In 1.45 this option was added to turn off what had always been the default.

extra-filename

This option allows you to put extra data in each output filename. It takes a string to add as a suffix to the filename. See the `--emit` flag for more information.

force-frame-pointers

This flag forces the use of frame pointers. It takes one of the following values:

- `y`, `yes`, `on`, or no value: force-enable frame pointers.
- `n`, `no`, or `off`: do not force-enable frame pointers. This does not necessarily mean frame pointers will be removed.

The default behaviour, if frame pointers are not force-enabled, depends on the target.

force-unwind-tables

This flag forces the generation of unwind tables. It takes one of the following values:

- `y`, `yes`, `on`, or no value: Unwind tables are forced to be generated.
- `n`, `no`, or `off`: Unwind tables are not forced to be generated. If unwind tables are required by the target or `-C panic=unwind`, an error will be emitted.

The default if not specified depends on the target.

incremental

This flag allows you to enable incremental compilation, which allows `rustc` to save information after compiling a crate to be reused when recompiling the crate, improving re-compile times. This takes a path to a directory where incremental files will be stored.

inline-threshold

This option lets you set the default threshold for inlining a function. It takes an unsigned integer as a value. Inlining is based on a cost model, where a higher threshold will allow more inlining.

The default depends on the opt-level:

opt-level	Threshold
0	N/A, only inlines always-inline functions
1	N/A, only inlines always-inline functions and LLVM lifetime intrinsics
2	225
3	275
s	75
z	25

link-arg

This flag lets you append a single extra argument to the linker invocation.

"Append" is significant; you can pass this flag multiple times to add multiple arguments.

link-args

This flag lets you append multiple extra arguments to the linker invocation. The options should be separated by spaces.

link-dead-code

This flag controls whether the linker will keep dead code. It takes one of the following values:

- `y`, `yes`, `on`, or no value: keep dead code.
- `n`, `no`, or `off`: remove dead code (the default).

An example of when this flag might be useful is when trying to construct code coverage metrics.

linker

This flag controls which linker `rustc` invokes to link your code. It takes a path to the linker executable. If this flag is not specified, the linker will be inferred based on the target. See also the linker-flavor flag for another way to specify the linker.

linker-flavor

This flag controls the linker flavor used by `rustc`. If a linker is given with the `-C linker` flag, then the linker flavor is inferred from the value provided. If no linker is given then the linker flavor is used to determine the linker to use. Every `rustc` target defaults to some linker flavor. Valid options are:

- `em`: use Emscripten `emcc`.
- `gcc`: use the `cc` executable, which is typically gcc or clang on many systems.
- `ld`: use the `ld` executable.
- `msvc`: use the `link.exe` executable from Microsoft Visual Studio MSVC.
- `ptx-linker`: use `rust-ptx-linker` for Nvidia NVPTX GPGPU support.
- `wasm-ld`: use the `wasm-ld` executable, a port of LLVM `lld` for WebAssembly.
- `ld64.lld`: use the LLVM `lld` executable with the `-flavor darwin` flag for

Apple's `ld`.
- `ld.lld`: use the LLVM `lld` executable with the `-flavor gnu` flag for GNU binutils' `ld`.
- `lld-link`: use the LLVM `lld` executable with the `-flavor link` flag for Microsoft's `link.exe`.

linker-plugin-lto

This flag defers LTO optimizations to the linker. See linker-plugin-LTO for more details. It takes one of the following values:

- `y`, `yes`, `on`, or no value: enable linker plugin LTO.
- `n`, `no`, or `off`: disable linker plugin LTO (the default).
- A path to the linker plugin.

More specifically this flag will cause the compiler to replace its typical object file output with LLVM bitcode files. For example an rlib produced with `-Clinker-plugin-lto` will still have `*.o` files in it, but they'll all be LLVM bitcode instead of actual machine code. It is expected that the native platform linker is capable of loading these LLVM bitcode files and generating code at link time (typically after performing optimizations).

Note that rustc can also read its own object files produced with `-Clinker-plugin-lto`. If an rlib is only ever going to get used later with a `-Clto` compilation then you can pass `-Clinker-plugin-lto` to speed up compilation and avoid generating object files that aren't used.

llvm-args

This flag can be used to pass a list of arguments directly to LLVM.

The list must be separated by spaces.

Pass `--help` to see a list of options.

lto

This flag controls whether LLVM uses link time optimizations to produce better optimized code, using whole-program analysis, at the cost of longer linking time. It takes one of the following values:

- `y`, `yes`, `on`, `fat`, or no value: perform "fat" LTO which attempts to perform optimizations across all crates within the dependency graph.
- `n`, `no`, `off`: disables LTO.
- `thin`: perform "thin" LTO. This is similar to "fat", but takes substantially less time to run while still achieving performance gains similar to "fat".

If `-C lto` is not specified, then the compiler will attempt to perform "thin local LTO" which performs "thin" LTO on the local crate only across its codegen units. When `-C lto` is not specified, LTO is disabled if codegen units is 1 or optimizations are disabled (`-C opt-level=0`). That is:

- When `-C lto` is not specified:
  - `codegen-units=1`: disable LTO.
  - `opt-level=0`: disable LTO.
- When `-C lto=true`:
  - `lto=true`: 16 codegen units, perform fat LTO across crates.
  - `codegen-units=1 + lto=true`: 1 codegen unit, fat LTO across crates.

See also linker-plugin-lto for cross-language LTO.

metadata

This option allows you to control the metadata used for symbol mangling. This takes a space-separated list of strings. Mangled symbols will incorporate a hash of the metadata. This may be used, for example, to differentiate symbols between two different versions of the same crate being linked.

no-prepopulate-passes

This flag tells the pass manager to use an empty list of passes, instead of the usual pre-populated list of passes.

no-redzone

This flag allows you to disable the red zone. It takes one of the following values:

- `y`, `yes`, `on`, or no value: disable the red zone.
- `n`, `no`, or `off`: enable the red zone.

The default behaviour, if the flag is not specified, depends on the target.

no-stack-check

This option is deprecated and does nothing.

no-vectorize-loops

This flag disables loop vectorization.

no-vectorize-slp

This flag disables vectorization using superword-level parallelism.

opt-level

This flag controls the optimization level.

- `0`: no optimizations, also turns on `cfg(debug_assertions)` (the default).
- `1`: basic optimizations.
- `2`: some optimizations.
- `3`: all optimizations.
- `s`: optimize for binary size.
- `z`: optimize for binary size, but also turn off loop vectorization.

Note: The `-O` flag is an alias for `-C opt-level=2`.

The default is `0`.

overflow-checks

This flag allows you to control the behavior of runtime integer overflow. When overflow-checks are enabled, a panic will occur on overflow. This flag takes one of the following values:

- `y`, `yes`, `on`, or no value: enable overflow checks.
- `n`, `no`, or `off`: disable overflow checks.

If not specified, overflow checks are enabled if debug-assertions are enabled, disabled otherwise.

panic

This option lets you control what happens when the code panics.

- `abort`: terminate the process upon panic
- `unwind`: unwind the stack upon panic

If not specified, the default depends on the target.

passes

This flag can be used to add extra LLVM passes to the compilation.

The list must be separated by spaces.

See also the `no-prepopulate-passes` flag.

prefer-dynamic

By default, `rustc` prefers to statically link dependencies. This option will indicate that dynamic linking should be used if possible if both a static and dynamic versions of a library are available. There is an internal algorithm for determining whether or not it is possible to statically or dynamically link with a dependency. For example, `cdylib` crate types may only use static linkage. This flag takes one of the following values:

- `y`, `yes`, `on`, or no value: use dynamic linking.
- `n`, `no`, or `off`: use static linking (the default).

profile-generate

This flag allows for creating instrumented binaries that will collect profiling data for use with profile-guided optimization (PGO). The flag takes an optional argument which is the path to a directory into which the instrumented binary will emit the collected data. See the chapter on profile-guided optimization for more information.

profile-use

This flag specifies the profiling data file to be used for profile-guided optimization (PGO). The flag takes a mandatory argument which is the path to a valid `.profdata` file. See the chapter on profile-guided optimization for more information.

relocation-model

This option controls generation of position-independent code (PIC).

Supported values for this option are:

Primary relocation models

- `static` - non-relocatable code, machine instructions may use absolute addressing modes.

- `pic` - fully relocatable position independent code, machine instructions need to use relative addressing modes.
  Equivalent to the "uppercase" `-fPIC` or `-fPIE` options in other compilers, depending on the produced crate types.
  This is the default model for majority of supported targets.

Special relocation models

- `dynamic-no-pic` - relocatable external references, non-relocatable code.
  Only makes sense on Darwin and is rarely used.
  If StackOverflow tells you to use this as an opt-out of PIC or PIE, don't believe it, use `-C relocation-model=static` instead.
- `ropi`, `rwpi` and `ropi-rwpi` - relocatable code and read-only data, relocatable read-write data, and combination of both, respectively.
  Only makes sense for certain embedded ARM targets.
- `default` - relocation model default to the current target.
  Only makes sense as an override for some other explicitly specified relocation model previously set on the command line.

Supported values can also be discovered by running `rustc --print relocation-models`.

Linking effects

In addition to codegen effects, `relocation-model` has effects during linking.

If the relocation model is `pic` and the current target supports position-independent executables (PIE), the linker will be instructed (`-pie`) to produce one.
If the target doesn't support both position-independent and statically linked executables, then `-C target-feature=+crt-static` "wins" over `-C relocation-model=pic`, and the linker is instructed (`-static`) to produce a statically linked but not position-independent executable.

remark

This flag lets you print remarks for optimization passes.

The list of passes should be separated by spaces.

`all` will remark on every pass.

rpath

This flag controls whether `rpath` is enabled. It takes one of the following values:

- `y`, `yes`, `on`, or no value: enable rpath.
- `n`, `no`, or `off`: disable rpath (the default).

save-temps

This flag controls whether temporary files generated during compilation are deleted once compilation finishes. It takes one of the following values:

- `y`, `yes`, `on`, or no value: save temporary files.
- `n`, `no`, or `off`: delete temporary files (the default).

soft-float

This option controls whether `rustc` generates code that emulates floating point instructions in software. It takes one of the following values:

- `y`, `yes`, `on`, or no value: use soft floats.
- `n`, `no`, or `off`: use hardware floats (the default).

target-cpu

This instructs `rustc` to generate code specifically for a particular processor.

You can run `rustc --print target-cpus` to see the valid options to pass here. Additionally, `native` can be passed to use the processor of the host machine. Each target has a default base CPU.

target-feature

Individual targets will support different features; this flag lets you control enabling or disabling a feature. Each feature should be prefixed with a `+` to enable it or `-` to disable it.

Features from multiple `-C target-feature` options are combined.
Multiple features can be specified in a single option by separating them with commas - `-C target-feature=+x,-y`.
If some feature is specified more than once with both `+` and `-`, then values passed later override values passed earlier.
For example, `-C target-feature=+x,-y,+z -Ctarget-feature=-x,+y` is equivalent to `-C target-feature=-x,+y,+z`.

To see the valid options and an example of use, run `rustc --print target-features`.

Using this flag is unsafe and might result in undefined runtime behavior.

See also the `target_feature` attribute for controlling features per-function.

This also supports the feature `+crt-static` and `-crt-static` to control static C runtime linkage.

Each target and `target-cpu` has a default set of enabled features.

JSON Output

This chapter documents the JSON structures emitted by `rustc`. JSON may be enabled with the `--error-format=json` flag. Additional options may be specified with the `--json` flag which can change which messages are generated, and the format of the messages.

JSON messages are emitted one per line to stderr.

If parsing the output with Rust, the `cargo_metadata` crate provides some support for parsing the messages.

When parsing, care should be taken to be forwards-compatible with future changes to the format. Optional values may be `null`. New fields may be added. Enumerated fields like "level" or "suggestion_applicability" may add new values.

Diagnostics

Diagnostic messages provide errors or possible concerns generated during compilation. `rustc` provides detailed information about where the diagnostic originates, along with hints and suggestions.

Diagnostics are arranged in a parent/child relationship where the parent diagnostic value is the core of the diagnostic, and the attached children provide additional context, help, and information.

Diagnostics have the following format:

```
{
    /* The primary message. */
    "message": "unused variable: `x`",
    /* The diagnostic code.
       Some messages may set this value to null.
    */
    "code": {
        /* A unique string identifying which diagnostic triggered. */
        "code": "unused_variables",
        /* An optional string explaining more detail about the
diagnostic code. */
        "explanation": null
    },
    /* The severity of the diagnostic.
       Values may be:
       - "error": A fatal error that prevents compilation.
       - "warning": A possible error or concern.
       - "note": Additional information or context about the diagnostic.
       - "help": A suggestion on how to resolve the diagnostic.
       - "failure-note": A note attached to the message for further
information.
       - "error: internal compiler error": Indicates a bug within the
compiler.
    */
    "level": "warning",
    /* An array of source code locations to point out specific details
about
       where the diagnostic originates from. This may be empty, for
example
       for some global messages, or child messages attached to a parent.

       Character offsets are offsets of Unicode Scalar Values.
    */
    "spans": [
        {
            /* The file where the span is located.
               Note that this path may not exist. For example, if the
path
               points to the standard library, and the rust src is not
               available in the sysroot, then it may point to a non-
existent
               file. Beware that this may also point to the source of an
               external crate.
            */
            "file_name": "lib.rs",
            /* The byte offset where the span starts (0-based,
inclusive). */
            "byte_start": 21,
            /* The byte offset where the span ends (0-based, exclusive).
*/
```

```
            "byte_end": 22,
            /* The first line number of the span (1-based, inclusive).
*/
            "line_start": 2,
            /* The last line number of the span (1-based, inclusive). */
            "line_end": 2,
            /* The first character offset of the line_start (1-based,
inclusive). */
            "column_start": 9,
            /* The last character offset of the line_end (1-based,
exclusive). */
            "column_end": 10,
            /* Whether or not this is the "primary" span.

               This indicates that this span is the focal point of the
               diagnostic.

               There are rare cases where multiple spans may be marked
as
               primary. For example, "immutable borrow occurs here" and
               "mutable borrow ends here" can be two separate primary
spans.

               The top (parent) message should always have at least one
               primary span, unless it has zero spans. Child messages
may have
               zero or more primary spans.
            */
            "is_primary": true,
            /* An array of objects showing the original source code for
this
               span. This shows the entire lines of text where the span
is
               located. A span across multiple lines will have a
separate
               value for each line.
            */
            "text": [
                {
                    /* The entire line of the original source code. */
                    "text": "    let x = 123;",
                    /* The first character offset of the line of
                       where the span covers this line (1-based,
inclusive). */
                    "highlight_start": 9,
                    /* The last character offset of the line of
                       where the span covers this line (1-based,
exclusive). */
                    "highlight_end": 10
                }
            ],
            /* An optional message to display at this span location.
               This is typically null for primary spans.
            */
            "label": null,
            /* An optional string of a suggested replacement for this
span to
               solve the issue. Tools may try to replace the contents of
the
               span with this text.
            */
            "suggested_replacement": null,
            /* An optional string that indicates the confidence of the
               "suggested_replacement". Tools may use this value to
determine
               whether or not suggestions should be automatically
applied.

               Possible values may be:
               - "MachineApplicable": The suggestion is definitely what
the
                 user intended. This suggestion should be automatically
                 applied.
               - "MaybeIncorrect": The suggestion may be what the user
                 intended, but it is uncertain. The suggestion should
result
                 in valid Rust code if it is applied.
               - "HasPlaceholders": The suggestion contains placeholders
like
                 `(...)`. The suggestion cannot be applied automatically
                 because it will not result in valid Rust code. The user
will
                 need to fill in the placeholders.
               - "Unspecified": The applicability of the suggestion is
unknown.
            */
            "suggestion_applicability": null,
            /* An optional object indicating the expansion of a macro
within
               this span.

               If a message occurs within a macro invocation, this
object will
               provide details of where within the macro expansion the
message
               is located.
            */
            "expansion": {
                /* The span of the macro invocation.
                   Uses the same span definition as the "spans" array.
```

```
                    */
                    "span": {/*...*/}
                    /* Name of the macro, such as "foo!" or "#[derive(Eq)]".
*/
                    "macro_decl_name": "some_macro!",
                    /* Optional span where the relevant part of the macro is
                      defined. */
                    "def_site_span": {/*...*/},
                }
            }
        ],
        /* Array of attached diagnostic messages.
           This is an array of objects using the same format as the parent
           message. Children are not nested (children do not themselves
           contain "children" definitions).
        */
        "children": [
            {
                "message": "`#[warn(unused_variables)]` on by default",
                "code": null,
                "level": "note",
                "spans": [],
                "children": [],
                "rendered": null
            },
            {
                "message": "if this is intentional, prefix it with an
underscore",
                "code": null,
                "level": "help",
                "spans": [
                    {
                        "file_name": "lib.rs",
                        "byte_start": 21,
                        "byte_end": 22,
                        "line_start": 2,
                        "line_end": 2,
                        "column_start": 9,
                        "column_end": 10,
                        "is_primary": true,
                        "text": [
                            {
                                "text": "    let x = 123;",
                                "highlight_start": 9,
                                "highlight_end": 10
                            }
                        ],
                        "label": null,
                        "suggested_replacement": "_x",
                        "suggestion_applicability": "MachineApplicable",
                        "expansion": null
                    }
                ],
                "children": [],
                "rendered": null
            }
        ],
        /* Optional string of the rendered version of the diagnostic as
    displayed
           by rustc. Note that this may be influenced by the `--json` flag.
        */
        "rendered": "warning: unused variable: `x`\n --> lib.rs:2:9\n  |\n2
    |     let x = 123;\n  |         ^ help: if this is intentional, prefix
    it with an underscore: `_x`\n  |\n  = note: `#[warn(unused_variables)]`
    on by default\n\n"
    }
```

Artifact notifications

Artifact notifications are emitted when the `--json=artifacts` flag is used. They indicate that a file artifact has been saved to disk. More information about emit kinds may be found in the `--emit` flag documentation.

```
{
    /* The filename that was generated. */
    "artifact": "libfoo.rlib",
    /* The kind of artifact that was generated. Possible values:
       - "link": The generated crate as specified by the crate-type.
       - "dep-info": The `.d` file with dependency information in a
Makefile-like syntax.
       - "metadata": The Rust `.rmeta` file containing metadata about
the crate.
       - "save-analysis": A JSON file emitted by the `-Zsave-analysis`
feature.
    */
    "emit": "link"
}
```

Targets

`rustc` is a cross-compiler by default. This means that you can use any compiler to build for any architecture. The list of *targets* are the possible architectures that you can build for.

To see all the options that you can set with a target, see the docs here.

To compile to a particular target, use the `--target` flag:

```
$ rustc src/main.rs --target=wasm32-unknown-unknown
```

Target Features

`x86`, and `ARMv8` are two popular CPU architectures. Their instruction sets form a common baseline across most CPUs. However, some CPUs extend these with custom instruction sets, e.g. vector (`AVX`), bitwise manipulation (`BMI`) or cryptographic (`AES`).

Developers, who know on which CPUs their compiled code is going to run can choose to add (or remove) CPU specific instruction sets via the `-C target-feature=val` flag.

Please note, that this flag is generally considered as unsafe. More details can be found in this section.

Built-in Targets

`rustc` ships with the ability to compile to many targets automatically, we call these "built-in" targets, and they generally correspond to targets that the team is supporting directly.

To see the list of built-in targets, you can run `rustc --print target-list`, or look at the API docs. Each module there defines a builder for a particular target.

Custom Targets

If you'd like to build for a target that is not yet supported by `rustc`, you can use a "custom target specification" to define a target. These target specification files are JSON. To see the JSON for the host target, you can run:

```
$ rustc +nightly -Z unstable-options --print target-spec-json
```

To see it for a different target, add the `--target` flag:

```
$ rustc +nightly -Z unstable-options --target=wasm32-unknown-unknown
--print target-spec-json
```

To use a custom target, see `xargo` .

Known Issues

This section informs you about known "gotchas". Keep in mind, that this section is (and always will be) incomplete. For suggestions and amendments, feel free to contribute to this guide.

Target Features

Most target-feature problems arise, when mixing code that have the target-feature *enabled* with code that have it *disabled*. If you want to avoid undefined behavior, it is recommended to build *all code* (including the standard library and imported crates) with a common set of target-features.

By default, compiling your code with the `-C target-feature` flag will not recompile the entire standard library and/or imported crates with matching target features. Therefore, target features are generally considered as unsafe. Using `#[target_feature]` on individual functions makes the function unsafe.

Examples:

Target-Feature	Issue	Seen on	Description	Details
`+soft-float` and `-sse`	Segfaults and ABI mismatches	`x86` and `x86-64`	The `x86` and `x86_64` architecture uses SSE registers (aka `xmm`) for floating point operations. Using software emulated floats ("soft-floats") disables usage of `xmm` registers, but	#63466

Target-Feature	Issue	Seen on	Description	Details
			parts of Rust's core libraries (e.g. `std::f32` or `std::f64`) are compiled without soft-floats and expect parameters to be passed in `xmm` registers. This leads to ABI mismatches. Attempting to compile with disabled SSE causes the same error, too.	

Profile Guided Optimization

`rustc` supports doing profile-guided optimization (PGO). This chapter describes what PGO is, what it is good for, and how it can be used.

What Is Profiled-Guided Optimization?

The basic concept of PGO is to collect data about the typical execution of a program (e.g. which branches it is likely to take) and then use this data to inform optimizations such as inlining, machine-code layout, register allocation, etc.

There are different ways of collecting data about a program's execution. One is to run the program inside a profiler (such as `perf`) and another is to create an instrumented binary, that is, a binary that has data collection built into it, and run that. The latter usually provides more accurate data and it is also what is supported by `rustc` .

Usage

Generating a PGO-optimized program involves following a workflow with four steps:

1. Compile the program with instrumentation enabled (e.g. `rustc -Cprofile-generate=/tmp/pgo-data main.rs`)
2. Run the instrumented program (e.g. `./main`) which generates a `default_<id>.profraw` file
3. Convert the `.profraw` file into a `.profdata` file using LLVM's `llvm-profdata` tool
4. Compile the program again, this time making use of the profiling data (for example `rustc -Cprofile-use=merged.profdata main.rs`)

An instrumented program will create one or more `.profraw` files, one for each instrumented binary. E.g. an instrumented executable that loads two instrumented dynamic libraries at runtime will generate three `.profraw` files. Running an instrumented binary multiple times, on the other hand, will re-use the respective `.profraw` files, updating them in place.

These `.profraw` files have to be post-processed before they can be fed back into the compiler. This is done by the `llvm-profdata` tool. This tool is most easily installed via

```
rustup component add llvm-tools-preview
```

Note that installing the `llvm-tools-preview` component won't add `llvm-profdata` to the `PATH`. Rather, the tool can be found in:

```
~/.rustup/toolchains/<toolchain>/lib/rustlib/<target-triple>/bin/
```

Alternatively, an `llvm-profdata` coming with a recent LLVM or Clang version usually works too.

The `llvm-profdata` tool merges multiple `.profraw` files into a single `.profdata` file that can then be fed back into the compiler via `-Cprofile-use`:

```
# STEP 1: Compile the binary with instrumentation
rustc -Cprofile-generate=/tmp/pgo-data -O ./main.rs

# STEP 2: Run the binary a few times, maybe with common sets of args.
#         Each run will create or update `.profraw` files in /tmp/pgo-
data
./main mydata1.csv
./main mydata2.csv
./main mydata3.csv

# STEP 3: Merge and post-process all the `.profraw` files in /tmp/pgo-
data
llvm-profdata merge -o ./merged.profdata /tmp/pgo-data

# STEP 4: Use the merged `.profdata` file during optimization. All
`rustc`
#         flags have to be the same.
rustc -Cprofile-use=./merged.profdata -O ./main.rs
```

A Complete Cargo Workflow

Using this feature with Cargo works very similar to using it with `rustc` directly. Again, we generate an instrumented binary, run it to produce data, merge the data, and feed it back into the compiler. Some things of note:

- We use the `RUSTFLAGS` environment variable in order to pass the PGO compiler flags to the compilation of all crates in the program.
- We pass the `--target` flag to Cargo, which prevents the `RUSTFLAGS` arguments to be passed to Cargo build scripts. We don't want the build scripts to generate a bunch of `.profraw` files.
- We pass `--release` to Cargo because that's where PGO makes the most sense. In theory, PGO can also be done on debug builds but there is little reason to do so.
- It is recommended to use *absolute paths* for the argument of `-Cprofile-generate` and `-Cprofile-use`. Cargo can invoke `rustc` with varying working directories, meaning that `rustc` will not be able to find the supplied `.profdata` file. With absolute paths this is not an issue.
- It is good practice to make sure that there is no left-over profiling data from previous compilation sessions. Just deleting the directory is a simple way of doing so (see `STEP 0` below).

This is what the entire workflow looks like:

```
# STEP 0: Make sure there is no left-over profiling data from previous
runs
rm -rf /tmp/pgo-data

# STEP 1: Build the instrumented binaries
RUSTFLAGS="-Cprofile-generate=/tmp/pgo-data" \
    cargo build --release --target=x86_64-unknown-linux-gnu

# STEP 2: Run the instrumented binaries with some typical data
./target/x86_64-unknown-linux-gnu/release/myprogram mydata1.csv
./target/x86_64-unknown-linux-gnu/release/myprogram mydata2.csv
./target/x86_64-unknown-linux-gnu/release/myprogram mydata3.csv

# STEP 3: Merge the `.profraw` files into a `.profdata` file
llvm-profdata merge -o /tmp/pgo-data/merged.profdata /tmp/pgo-data

# STEP 4: Use the `.profdata` file for guiding optimizations
RUSTFLAGS="-Cprofile-use=/tmp/pgo-data/merged.profdata" \
    cargo build --release --target=x86_64-unknown-linux-gnu
```

Troubleshooting

- It is recommended to pass `-Cllvm-args=-pgo-warn-missing-function` during the `-Cprofile-use` phase. LLVM by default does not warn if it cannot find profiling data for a given function. Enabling this warning will make it easier to spot errors in your setup.

- There is a known issue in Cargo prior to version 1.39 that will prevent PGO from working correctly. Be sure to use Cargo 1.39 or newer when doing PGO.

Further Reading

`rustc`'s PGO support relies entirely on LLVM's implementation of the feature and is equivalent to what Clang offers via the `-fprofile-generate` / `-fprofile-use` flags. The Profile Guided Optimization section in Clang's documentation is therefore an interesting read for anyone who wants to use PGO with Rust.

Linker-plugin-LTO

The `-C linker-plugin-lto` flag allows for deferring the LTO optimization to the actual linking step, which in turn allows for performing interprocedural optimizations across programming language boundaries if all the object files being linked were created by LLVM based toolchains. The prime example here would be linking Rust code together with Clang-compiled C/C++ code.

Usage

There are two main cases how linker plugin based LTO can be used:

- compiling a Rust `staticlib` that is used as a C ABI dependency
- compiling a Rust binary where `rustc` invokes the linker

In both cases the Rust code has to be compiled with `-C linker-plugin-lto` and the C/C++ code with `-flto` or `-flto=thin` so that object files are emitted as LLVM bitcode.

Rust `staticlib` as dependency in C/C++ program

In this case the Rust compiler just has to make sure that the object files in the `staticlib` are in the right format. For linking, a linker with the LLVM plugin must be used (e.g. LLD).

Using `rustc` directly:

```
# Compile the Rust staticlib
rustc --crate-type=staticlib -Clinker-plugin-lto -Copt-level=2 ./lib.rs
# Compile the C code with `-flto=thin`
clang -c -O2 -flto=thin -o main.o ./main.c
# Link everything, making sure that we use an appropriate linker
clang -flto=thin -fuse-ld=lld -L . -l"name-of-your-rust-lib" -o main -O2
./cmain.o
```

Using `cargo`:

```
# Compile the Rust staticlib
RUSTFLAGS="-Clinker-plugin-lto" cargo build --release
# Compile the C code with `-flto=thin`
clang -c -O2 -flto=thin -o main.o ./main.c
# Link everything, making sure that we use an appropriate linker
clang -flto=thin -fuse-ld=lld -L . -l"name-of-your-rust-lib" -o main -O2
./cmain.o
```

C/C++ code as a dependency in Rust

In this case the linker will be invoked by `rustc`. We again have to make sure that an appropriate linker is used.

Using `rustc` directly:

```
# Compile C code with `-flto`
clang ./clib.c -flto=thin -c -o ./clib.o -O2
# Create a static library from the C code
ar crus ./libxyz.a ./clib.o

# Invoke `rustc` with the additional arguments
rustc -Clinker-plugin-lto -L. -Copt-level=2 -Clinker=clang -Clink-arg=-
fuse-ld=lld ./main.rs
```

Using `cargo` directly:

```
# Compile C code with `-flto`
clang ./clib.c -flto=thin -c -o ./clib.o -O2
# Create a static library from the C code
ar crus ./libxyz.a ./clib.o

# Set the linking arguments via RUSTFLAGS
RUSTFLAGS="-Clinker-plugin-lto -Clinker=clang -Clink-arg=-fuse-ld=lld"
cargo build --release
```

Explicitly specifying the linker plugin to be used by `rustc`

If one wants to use a linker other than LLD, the LLVM linker plugin has to be specified explicitly. Otherwise the linker cannot read the object files. The path to the plugin is passed as an argument to the `-Clinker-plugin-lto` option:

```
rustc -Clinker-plugin-lto="/path/to/LLVMgold.so" -L. -Copt-level=2
./main.rs
```

Toolchain Compatibility

In order for this kind of LTO to work, the LLVM linker plugin must be able to handle the LLVM bitcode produced by both `rustc` and `clang`.

Best results are achieved by using a `rustc` and `clang` that are based on the exact same version of LLVM. One can use `rustc -vV` in order to view the LLVM used by a given `rustc` version. Note that the version number given here is only an approximation as Rust sometimes uses unstable revisions of LLVM. However, the approximation is usually reliable.

The following table shows known good combinations of toolchain versions.

	Clang 7	Clang 8	Clang 9
Rust 1.34	✗	✓	✗
Rust 1.35	✗	✓	✗
Rust 1.36	✗	✓	✗
Rust 1.37	✗	✓	✗
Rust 1.38	✗	✗	✓
Rust 1.39	✗	✗	✓
Rust 1.40	✗	✗	✓
Rust 1.41	✗	✗	✓
Rust 1.42	✗	✗	✓
Rust 1.43	✗	✗	✓

Note that the compatibility policy for this feature might change in the future.

Contributing to rustc

We'd love to have your help improving `rustc`! To that end, we've written a whole book on its internals, how it works, and how to get started working on it. To learn more, you'll want to check that out.

If you would like to contribute to *this* book, you can find its source in the rustc source at src/doc/rustc.