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 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.

control-flow-guard

This flag controls whether LLVM enables the Windows Control Flow Guard platform security feature. This flag is currently ignored for non-Windows targets. It takes one of the following values:

- `y`, `yes`, `on`, `checks`, or no value: enable Control Flow Guard.
- `nochecks`: emit Control Flow Guard metadata without runtime enforcement checks (this should only be used for testing purposes as it does not provide security enforcement).
- `n`, `no`, `off`: do not enable Control Flow Guard (the default).

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 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

instrument-coverage

This option enables instrumentation-based code coverage support. See the chapter on instrumentation-based code coverage for more information.

Note that while the `-C instrument-coverage` option is stable, the profile data format produced by the resulting instrumentation may change, and may not work with coverage tools other than those built and shipped with the compiler.

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.

link-self-contained

On targets that support it this flag controls whether the linker will use libraries and objects shipped with Rust instead or those in the system. It takes one of the following values:

- no value: rustc will use heuristic to disable self-contained mode if system has necessary tools.
- `y`, `yes`, `on`: use only libraries/objects shipped with Rust.
- `n`, `no`, or `off`: rely on the user or the linker to provide non-Rust libraries/objects.

This allows overriding cases when detection fails or user wants to use shipped libraries.

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.
- `bpf-linker` : use `bpf-linker` for eBPF 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` is specified:
  - `lto` : 16 codegen units, perform fat LTO across crates.
  - `codegen-units=1` + `lto` : 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.
- `pie` - position independent executable, relocatable code but without support for symbol interpositioning (replacing symbols by name using `LD_PRELOAD` and similar). Equivalent to the "uppercase" `-fPIE` option in other compilers. `pie` code cannot be linked into shared libraries (you'll get a linking error on attempt to do this).

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).

split-debuginfo

This option controls the emission of "split debuginfo" for debug information that `rustc` generates. The default behavior of this option is platform-specific, and not all possible values for this option work on all platforms. Possible values are:

- `off` - This is the default for platforms with ELF binaries and windows-gnu (not Windows MSVC and not macOS). This typically means that DWARF debug information can be found in the final artifact in sections of the executable. This option is not supported on Windows MSVC. On macOS this options prevents the final execution of `dsymutil` to generate debuginfo.
- `packed` - This is the default for Windows MSVC and macOS. The term "packed" here means that all the debug information is packed into a separate file from the main executable. On Windows MSVC this is a `*.pdb` file, on macOS this is a `*.dSYM` folder, and on other platforms this is a `*.dwp` file.
- `unpacked` - This means that debug information will be found in separate files for each compilation unit (object file). This is not supported on Windows MSVC. On macOS this means the original object files will contain debug information. On other Unix platforms this means that `*.dwo` files will contain debug information.

Note that `packed` and `unpacked` are gated behind `-Z unstable-options` on non-macOS platforms at this time.

strip

The option `-C strip=val` controls stripping of debuginfo and similar auxiliary data from binaries during linking.

Supported values for this option are:

- `none` - debuginfo and symbols (if they exist) are copied to the produced binary or separate files depending on the target (e.g. `.pdb` files in case of MSVC).
- `debuginfo` - debuginfo sections and debuginfo symbols from the symbol table section are stripped at link time and are not copied to the produced binary or separate files.
- `symbols` - same as `debuginfo`, but the rest of the symbol table section is stripped as well if the linker supports it.

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. Each target has a default base CPU. Special values include:

- `native` can be passed to use the processor of the host machine.
- `generic` refers to an LLVM target with minimal features but modern tuning.

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.

tune-cpu

This instructs `rustc` to schedule code specifically for a particular processor. This does not affect the compatibility (instruction sets or ABI), but should make your code slightly more efficient on the selected CPU.

The valid options are the same as those for `target-cpu`. The default is `None`, which LLVM translates as the `target-cpu`.

This is an unstable option. Use `-Z tune-cpu=machine` to specify a value.

Due to limitations in LLVM (12.0.0-git9218f92), this option is currently effective only for x86 targets.