## NAME

clang - the Clang C, C++, and Objective-C compiler

## SYNOPSIS

**clang** [*options*] *filename ...*

## DESCRIPTION

**clang** is a C, C++, and Objective-C compiler which encompasses preprocessing, parsing, optimization, code generation, assembly, and linking. Depending on which high-level mode setting is passed, Clang will stop before doing a full link. While Clang is highly integrated, it is important to understand the stages of compilation, to understand how to invoke it. These stages are:

### Driver

The clang executable is actually a small driver which controls the overall execution of other tools such as the compiler, assembler and linker. Typically you do not need to interact with the driver, but you transparently use it to run the other tools.

### Preprocessing

This stage handles tokenization of the input source file, macro expansion, #include expansion and handling of other preprocessor directives. The output of this stage is typically called a ".i" (for C), ".ii" (for C++), ".mi" (for Objective-C), or ".mii" (for Objective-C++) file.

### Parsing and Semantic Analysis

This stage parses the input file, translating preprocessor tokens into a parse tree. Once in the form of a parse tree, it applies semantic analysis to compute types for expressions as well and determine whether the code is well formed. This stage is responsible for generating most of the compiler warnings as well as parse errors. The output of this stage is an "Abstract Syntax Tree" (AST).

### Code Generation and Optimization

This stage translates an AST into low-level intermediate code (known as "LLVM IR") and ultimately to machine code. This phase is responsible for optimizing the generated code and handling target-specific code generation. The output of this stage is typically called a ".s" file or "assembly" file.

Clang also supports the use of an integrated assembler, in which the code generator produces object files directly. This avoids the overhead of generating the ".s" file and of calling the target assembler.

### Assembler

This stage runs the target assembler to translate the output of the compiler into a target object file.

The output of this stage is typically called a ".o" file or "object" file.

**Linker**
This stage runs the target linker to merge multiple object files into an executable or dynamic library. The output of this stage is typically called an "a.out", ".dylib" or ".so" file.

**Clang Static Analyzer**

The Clang Static Analyzer is a tool that scans source code to try to find bugs through code analysis. This tool uses many parts of Clang and is built into the same driver. Please see <*https://clang-analyzer.llvm.org*> for more details on how to use the static analyzer.

## OPTIONS

### Stage Selection Options

**-E** Run the preprocessor stage.

**-fsyntax-only**
Run the preprocessor, parser and semantic analysis stages.

**-S** Run the previous stages as well as LLVM generation and optimization stages and target-specific code generation, producing an assembly file.

**-c** Run all of the above, plus the assembler, generating a target ".o" object file.

**no stage selection option**
If no stage selection option is specified, all stages above are run, and the linker is run to combine the results into an executable or shared library.

### Language Selection and Mode Options

**-x <language>**
Treat subsequent input files as having type language.

**-std=<standard>**
Specify the language standard to compile for.

Supported values for the C language are:
**c89**
**c90**

**iso9899:1990**

ISO C 1990
**iso9899:199409**

ISO C 1990 with amendment 1
**gnu89**
**gnu90**

ISO C 1990 with GNU extensions
**c99**
**iso9899:1999**

ISO C 1999
**gnu99**

ISO C 1999 with GNU extensions
**c11**
**iso9899:2011**

ISO C 2011
**gnu11**

ISO C 2011 with GNU extensions
**c17**
**iso9899:2017**

ISO C 2017
**gnu17**

ISO C 2017 with GNU extensions

The default C language standard is **gnu17**, except on PS4, where it is **gnu99**.

Supported values for the C++ language are:
**c++98**
**c++03**

ISO C++ 1998 with amendments
**gnu++98**

**gnu++03**

ISO C++ 1998 with amendments and GNU extensions

**c++11**

ISO C++ 2011 with amendments

**gnu++11**

ISO C++ 2011 with amendments and GNU extensions

**c++14**

ISO C++ 2014 with amendments

**gnu++14**

ISO C++ 2014 with amendments and GNU extensions

**c++17**

ISO C++ 2017 with amendments

**gnu++17**

ISO C++ 2017 with amendments and GNU extensions

**c++20**

ISO C++ 2020 with amendments

**gnu++20**

ISO C++ 2020 with amendments and GNU extensions

**c++2b**

Working draft for ISO C++ 2023

**gnu++2b**

Working draft for ISO C++ 2023 with GNU extensions

The default C++ language standard is **gnu++17**.

Supported values for the OpenCL language are:

**cl1.0**

OpenCL 1.0

**cl1.1**

OpenCL 1.1

**cl1.2**

OpenCL 1.2

**cl2.0**

OpenCL 2.0

The default OpenCL language standard is **cl1.0**.

Supported values for the CUDA language are:

**cuda**

NVIDIA CUDA(tm)

**-stdlib=<library>**
Specify the C++ standard library to use; supported options are libstdc++ and libc++. If not specified, platform default will be used.

**-rtlib=<library>**
Specify the compiler runtime library to use; supported options are libgcc and compiler-rt. If not specified, platform default will be used.

**-ansi**
Same as -std=c89.

**-ObjC, -ObjC++**
Treat source input files as Objective-C and Object-C++ inputs respectively.

**-trigraphs**
Enable trigraphs.

**-ffreestanding**
Indicate that the file should be compiled for a freestanding, not a hosted, environment. Note that it is assumed that a freestanding environment will additionally provide *memcpy*, *memmove*, *memset* and *memcmp* implementations, as these are needed for efficient codegen for many programs.

**-fno-builtin**

Disable special handling and optimizations of well-known library functions, like **strlen()** and **malloc()**.

**-fno-builtin-<function>**
Disable special handling and optimizations for the specific library function. For example, **-fno-builtin-strlen** removes any special handling for the **strlen()** library function.

**-fno-builtin-std-<function>**
Disable special handling and optimizations for the specific C++ standard library function in namespace **std**. For example, **-fno-builtin-std-move_if_noexcept** removes any special handling for the **std::move_if_noexcept()** library function.

For C standard library functions that the C++ standard library also provides in namespace **std**, use *-fno-builtin-<function>* instead.

**-fmath-errno**
Indicate that math functions should be treated as updating **errno**.

**-fpascal-strings**
Enable support for Pascal-style strings with "\pfoo".

**-fms-extensions**
Enable support for Microsoft extensions.

**-fmsc-version=**
Set _MSC_VER. Defaults to 1300 on Windows. Not set otherwise.

**-fborland-extensions**
Enable support for Borland extensions.

**-fwritable-strings**
Make all string literals default to writable. This disables uniquing of strings and other optimizations.

**-flax-vector-conversions, -flax-vector-conversions=<kind>, -fno-lax-vector-conversions**
Allow loose type checking rules for implicit vector conversions. Possible values of <kind>:

- **none**: allow no implicit conversions between vectors
- **integer**: allow implicit bitcasts between integer vectors of the same overall bit-width

- **all**: allow implicit bitcasts between any vectors of the same overall bit-width

<kind> defaults to **integer** if unspecified.

**-fblocks**
Enable the "Blocks" language feature.

**-fobjc-abi-version=version**
Select the Objective-C ABI version to use. Available versions are 1 (legacy "fragile" ABI), 2 (non-fragile ABI 1), and 3 (non-fragile ABI 2).

**-fobjc-nonfragile-abi-version=<version>**
Select the Objective-C non-fragile ABI version to use by default. This will only be used as the Objective-C ABI when the non-fragile ABI is enabled (either via *-fobjc-nonfragile-abi*, or because it is the platform default).

**-fobjc-nonfragile-abi, -fno-objc-nonfragile-abi**
Enable use of the Objective-C non-fragile ABI. On platforms for which this is the default ABI, it can be disabled with *-fno-objc-nonfragile-abi*.

**Target Selection Options**
Clang fully supports cross compilation as an inherent part of its design. Depending on how your version of Clang is configured, it may have support for a number of cross compilers, or may only support a native target.

**-arch <architecture>**
Specify the architecture to build for (Mac OS X specific).

**-target <architecture>**
Specify the architecture to build for (all platforms).

**-mmacosx-version-min=<version>**
When building for macOS, specify the minimum version supported by your application.

**-miphoneos-version-min**
When building for iPhone OS, specify the minimum version supported by your application.

**--print-supported-cpus**
Print out a list of supported processors for the given target (specified through **--target=<architecture>** or *-arch* **<architecture>**). If no target is specified, the system default target

will be used.

**-mcpu=?, -mtune=?**
Acts as an alias for *--print-supported-cpus*.

**-march=<cpu>**
Specify that Clang should generate code for a specific processor family member and later. For example, if you specify -march=i486, the compiler is allowed to generate instructions that are valid on i486 and later processors, but which may not exist on earlier ones.

## Code Generation Options

**-O0, -O1, -O2, -O3, -Ofast, -Os, -Oz, -Og, -O, -O4**
Specify which optimization level to use:

-*O0* Means "no optimization": this level compiles the fastest and generates the most debuggable code.

-*O1* Somewhere between -*O0* and -*O2*.

-*O2* Moderate level of optimization which enables most optimizations.

-*O3* Like -*O2*, except that it enables optimizations that take longer to perform or that may generate larger code (in an attempt to make the program run faster).

-*Ofast* Enables all the optimizations from -*O3* along with other aggressive optimizations that may violate strict compliance with language standards.

-*Os* Like -*O2* with extra optimizations to reduce code size.

-*Oz* Like -*Os* (and thus -*O2*), but reduces code size further.

-*Og* Like -*O1*. In future versions, this option might disable different optimizations in order to improve debuggability.

-*O* Equivalent to -*O1*.

-*O4* and higher
Currently equivalent to -*O3*

**-g, -gline-tables-only, -gmodules**

Control debug information output. Note that Clang debug information works best at *-O0*. When more than one option starting with *-g* is specified, the last one wins:

*-g* Generate debug information.

*-gline-tables-only* Generate only line table debug information. This allows for symbolicated backtraces with inlining information, but does not include any information about variables, their locations or types.

*-gmodules* Generate debug information that contains external references to types defined in Clang modules or precompiled headers instead of emitting redundant debug type information into every object file. This option transparently switches the Clang module format to object file containers that hold the Clang module together with the debug information. When compiling a program that uses Clang modules or precompiled headers, this option produces complete debug information with faster compile times and much smaller object files.

This option should not be used when building static libraries for distribution to other machines because the debug info will contain references to the module cache on the machine the object files in the library were built on.

**-fstandalone-debug -fno-standalone-debug**

Clang supports a number of optimizations to reduce the size of debug information in the binary. They work based on the assumption that the debug type information can be spread out over multiple compilation units. For instance, Clang will not emit type definitions for types that are not needed by a module and could be replaced with a forward declaration. Further, Clang will only emit type info for a dynamic C++ class in the module that contains the vtable for the class.

The *-fstandalone-debug* option turns off these optimizations. This is useful when working with 3rd-party libraries that don't come with debug information. This is the default on Darwin. Note that Clang will never emit type information for types that are not referenced at all by the program.

**-feliminate-unused-debug-types**

By default, Clang does not emit type information for types that are defined but not used in a program. To retain the debug info for these unused types, the negation **-fno-eliminate-unused-debug-types** can be used.

**-fexceptions**

Enable generation of unwind information. This allows exceptions to be thrown through Clang compiled stack frames. This is on by default in x86-64.

**-ftrapv**
Generate code to catch integer overflow errors. Signed integer overflow is undefined in C. With this flag, extra code is generated to detect this and abort when it happens.

**-fvisibility**
This flag sets the default visibility level.

**-fcommon, -fno-common**
This flag specifies that variables without initializers get common linkage. It can be disabled with *-fno-common*.

**-ftls-model=<model>**
Set the default thread-local storage (TLS) model to use for thread-local variables. Valid values are: "global-dynamic", "local-dynamic", "initial-exec" and "local-exec". The default is "global-dynamic". The default model can be overridden with the tls_model attribute. The compiler will try to choose a more efficient model if possible.

**-flto, -flto=full, -flto=thin, -emit-llvm**
Generate output files in LLVM formats, suitable for link time optimization. When used with *-S* this generates LLVM intermediate language assembly files, otherwise this generates LLVM bitcode format object files (which may be passed to the linker depending on the stage selection options).

The default for *-flto* is "full", in which the LLVM bitcode is suitable for monolithic Link Time Optimization (LTO), where the linker merges all such modules into a single combined module for optimization. With "thin", *ThinLTO* compilation is invoked instead.

**NOTE:**
On Darwin, when using *-flto* along with *-g* and compiling and linking in separate steps, you also need to pass **-Wl,-object_path_lto,<lto-filename>.o** at the linking step to instruct the ld64 linker not to delete the temporary object file generated during Link Time Optimization (this flag is automatically passed to the linker by Clang if compilation and linking are done in a single step). This allows debugging the executable as well as generating the **.dSYM** bundle using **dsymutil(1)**.

## Driver Options

*-###*
Print (but do not run) the commands to run for this compilation.

**--help**
Display available options.

**-Qunused-arguments**
Do not emit any warnings for unused driver arguments.

**-Wa,<args>**
Pass the comma separated arguments in args to the assembler.

**-Wl,<args>**
Pass the comma separated arguments in args to the linker.

**-Wp,<args>**
Pass the comma separated arguments in args to the preprocessor.

**-Xanalyzer <arg>**
Pass arg to the static analyzer.

**-Xassembler <arg>**
Pass arg to the assembler.

**-Xlinker <arg>**
Pass arg to the linker.

**-Xpreprocessor <arg>**
Pass arg to the preprocessor.

**-o <file>**
Write output to file.

**-print-file-name=<file>**
Print the full library path of file.

**-print-libgcc-file-name**
Print the library path for the currently used compiler runtime library ("libgcc.a" or "libclang_rt.builtins.*.a").

**-print-prog-name=<name>**
Print the full program path of name.

**-print-search-dirs**
Print the paths used for finding libraries and programs.

**-save-temps**
Save intermediate compilation results.

**-save-stats, -save-stats=cwd, -save-stats=obj**
Save internal code generation (LLVM) statistics to a file in the current directory (*-save-stats*/"-save-stats=cwd") or the directory of the output file ("-save-state=obj").

**-integrated-as, -no-integrated-as**
Used to enable and disable, respectively, the use of the integrated assembler. Whether the integrated assembler is on by default is target dependent.

**-time**
Time individual commands.

**-ftime-report**
Print timing summary of each stage of compilation.

**-v** Show commands to run and use verbose output.

## Diagnostics Options

**-fshow-column, -fshow-source-location, -fcaret-diagnostics, -fdiagnostics-fixit-info, -fdiagnostics-parseable-fixits, -fdiagnostics-print-source-range-info, -fprint-source-range-info, -fdiagnostics-show-option, -fmessage-length**
These options control how Clang prints out information about diagnostics (errors and warnings). Please see the Clang User's Manual for more information.

## Preprocessor Options

**-D<macroname>=<value>**
Adds an implicit #define into the predefines buffer which is read before the source file is preprocessed.

**-U<macroname>**
Adds an implicit #undef into the predefines buffer which is read before the source file is preprocessed.

**-include <filename>**
Adds an implicit #include into the predefines buffer which is read before the source file is preprocessed.

**-I<directory>**
Add the specified directory to the search path for include files.

**-F<directory>**
Add the specified directory to the search path for framework include files.

**-nostdinc**
Do not search the standard system directories or compiler builtin directories for include files.

**-nostdlibinc**
Do not search the standard system directories for include files, but do search compiler builtin include directories.

**-nobuiltininc**
Do not search clang's builtin directory for include files.

## ENVIRONMENT

**TMPDIR, TEMP, TMP**
These environment variables are checked, in order, for the location to write temporary files used during the compilation process.

**CPATH**
If this environment variable is present, it is treated as a delimited list of paths to be added to the default system include path list. The delimiter is the platform dependent delimiter, as used in the PATH environment variable.

Empty components in the environment variable are ignored.

**C_INCLUDE_PATH, OBJC_INCLUDE_PATH, CPLUS_INCLUDE_PATH, OBJCPLUS_INCLUDE_PATH**
These environment variables specify additional paths, as for *CPATH*, which are only used when processing the appropriate language.

**MACOSX_DEPLOYMENT_TARGET**
If *-mmacosx-version-min* is unspecified, the default deployment target is read from this

environment variable. This option only affects Darwin targets.

## BUGS

To report bugs, please visit <*https://github.com/llvm/llvm-project/issues/*>. Most bug reports should include preprocessed source files (use the *-E* option) and the full output of the compiler, along with information to reproduce.

## SEE ALSO

**as(1)**, **ld(1)**

## AUTHOR

Maintained by the Clang / LLVM Team (<http://clang.llvm.org>)

## COPYRIGHT