NAME

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

SYNOPSIS

clang [**–c**|**–S**|**–E**] **–std=***standard* **–g**
[**–O0**|**–O1**|**–O2**|**–Os**|**–Oz**|**–O3**|**–O4**]
–W*warnings...* **–pedantic**
–I*dir...* **–L***dir...*
–D*macro[=defn]*
–f*feature–option...*
–m*machine–option...*
–o *output-file*
–stdlib=*library*
input-filenames

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 parser 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.

The Clang compiler supports a large number of options to control each of these stages. In addition to compilation of code, Clang also supports other tools:

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.

OPTIONS

Stage Selection Options

−E Run the preprocessor stage.

−fsyntax−only
Run the preprocessor, parser and type checking 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.

−−analyze
Run the Clang Static Analyzer.

Language Selection and Mode Options

−x *language*
Treat subsequent input files as having type *language*.

−std=*language*
Specify the language standard to compile for.

−stdlib=*language*
Specify the C++ standard library to use; supported options are libstdc++ and libc++.

−ansi
Same as **−std=c89**.

−ObjC++
Treat source input files as Objective−C++ inputs.

−ObjC
Treat source input files as Objective-C inputs.

−trigraphs
Enable trigraphs.

−ffreestanding
Indicate that the file should be compiled for a freestanding, not a hosted, environment.

−fno−builtin
Disable special handling and optimizations of builtin functions like strlen and malloc.

−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
Allow loose type checking rules for implicit vector conversions.

−fblocks
Enable the ''Blocks'' language feature.

−fobjc−gc−only
Indicate that Objective-C code should be compiled in GC-only mode, which only works when Objective-C Garbage Collection is enabled.

−fobjc−gc
Indicate that Objective-C code should be compiled in hybrid-GC mode, which works with both GC and non-GC mode.

−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
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.

−mmacosx−version−min=*version*
When building for Mac OS/X, specify the minimum version supported by your application.

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

−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 −Os −Oz −O3 −O4
Specify which optimization level to use. **−O0** means ''no optimization'': this level compiles the fastest and generates the most debuggable code. **−O2** is a moderate level of optimization which enables most optimizations. **−Os** is like **−O2** with extra optimizations to reduce code size. **−Oz** is like **−Os** (and thus **−O2**), but reduces code size further. **−O3** is 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). On supported platforms, **−O4** enables link-time optimization; object files are stored in the LLVM bitcode file format and whole program optimization is done at link time. **−O1** is somewhere between **−O0** and **−O2**.

−g Generate debug information. Note that Clang debug information works best at **−O0**. At higher optimization levels, only line number information is currently available.

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

−flto −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).

Driver Options

−###
Print the commands to run for this compilation.

−−help
Display available options.

−Qunused−arguments
Don't emit warning 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 ''libgcc.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.

−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 delimitor, 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 <http://llvm.org/bugs/>. 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>).