Clang Compiler User's Manual

Introduction

The Clang Compiler is an open-source compiler for the C family of programming languages, aiming to be the best in class implementation of these languages. Clang builds on the LLVM optimizer and code generator, allowing it to provide high-quality optimization and code generation support for many targets. For more general information, please see the **Clang Web Site** or the **LLVM Web Site**.

This document describes important notes about using Clang as a compiler for an end-user, documenting the supported features, command line options, etc. If you are interested in using Clang to build a tool that processes code, please see **"Clang" CFE Internals Manual**. If you are interested in the **Clang Static Analyzer**, please see its web page.

Clang is one component in a complete toolchain for C family languages. A separate document describes the other pieces necessary to **assemble a complete toolchain**.

Clang is designed to support the C family of programming languages, which includes **C**, **Objective-C**, **C++**, and **Objective-C++** as well as many dialects of those. For language-specific information, please see the corresponding language specific section:

C Language: K&R C, ANSI C89, ISO C90, ISO C94 (C89+AMD1), ISO C99 (+TC1, TC2, TC3).
Objective-C Language: ObjC 1, ObjC 2, ObjC 2.1, plus variants depending on base language.
C++ Language
Objective C++ Language
OpenCL C Language: v1.0, v1.1, v1.2, v2.0.

In addition to these base languages and their dialects, Clang supports a broad variety of language extensions, which are documented in the corresponding language section. These extensions are provided to be compatible with the GCC, Microsoft, and other popular compilers as well as to improve functionality through Clang-specific features. The Clang driver and language features are intentionally designed to be as compatible with the GNU GCC compiler as reasonably possible, easing migration from GCC to Clang. In most cases, code "just works". Clang also provides an alternative driver, **clang-cl**, that is designed to be compatible with the Visual C++ compiler, cl.exe.

In addition to language specific features, Clang has a variety of features that depend on what CPU architecture or operating system is being compiled for. Please see the **Target-Specific Features and Limitations** section for more details.

The rest of the introduction introduces some basic **compiler terminology** that is used throughout this manual and contains a basic **introduction to using Clang** as a command line compiler.

Terminology

Front end, parser, backend, preprocessor, undefined behavior, diagnostic, optimizer

Basic Usage

Intro to how to use a C compiler for newbies.

compile + link compile then link debug info enabling optimizations picking a language to use, defaults to C11 by default. Autosenses based on extension. using a makefile

Command Line Options

This section is generally an index into other sections. It does not go into depth on the ones that are covered by other sections. However, the first part introduces the language selection and other high level options like -c, -g, etc.

Options to Control Error and Warning Messages

-Werror
: Turn warnings into errors.

-Werror=foo
: Turn warning "foo" into an error.

-Wno-error=foo
: Turn warning "foo" into a warning even if -Werror is specified.

-Wfoo
: Enable warning "foo". See the **diagnostics reference** for a complete list of the warning flags that can be specified in this way.

-Wno-foo
: Disable warning "foo".

-w
: Disable all diagnostics.

-Weverything
: **Enable all diagnostics.**

-pedantic
: Warn on language extensions.

-pedantic-errors
: Error on language extensions.

-Wsystem-headers
: Enable warnings from system headers.

-ferror-limit=123

Stop emitting diagnostics after 123 errors have been produced. The default is 20, and the error limit can be disabled with *-ferror-limit=0*.

-ftemplate-backtrace-limit=123

Only emit up to 123 template instantiation notes within the template instantiation backtrace for a single warning or error. The default is 10, and the limit can be disabled with *-ftemplate-backtrace-limit=0*.

Formatting of Diagnostics

Clang aims to produce beautiful diagnostics by default, particularly for new users that first come to Clang. However, different people have different preferences, and sometimes Clang is driven not by a human, but by a program that wants consistent and easily parsable output. For these cases, Clang provides a wide range of options to control the exact output format of the diagnostics that it generates.

-f[no-]show-column

Print column number in diagnostic.

This option, which defaults to on, controls whether or not Clang prints the column number of a diagnostic. For example, when this is enabled, Clang will print something like:

```
test.c:28:8: warning: extra tokens at end of #endif directive [-Wextra-tokens]
#endif bad
       ^
       //
```

When this is disabled, Clang will print "test.c:28: warning..." with no column number.

The printed column numbers count bytes from the beginning of the line; take care if your source contains multibyte characters.

-f[no-]show-source-location

Print source file/line/column information in diagnostic.

This option, which defaults to on, controls whether or not Clang prints the filename, line number and column number of a diagnostic. For example, when this is enabled, Clang will print something like:

```
test.c:28:8: warning: extra tokens at end of #endif directive [-Wextra-tokens]
#endif bad
       ^
       //
```

When this is disabled, Clang will not print the "test.c:28:8: " part.

-f[no-]caret-diagnostics

Print source line and ranges from source code in diagnostic. This option, which defaults to on, controls whether or not Clang prints the source line, source ranges, and caret when emitting a diagnostic. For example, when this is enabled, Clang will print something like:

```
test.c:28:8: warning: extra tokens at end of #endif directive [-Wextra-tokens]
#endif bad
       ^
       //
```

-f[no-]color-diagnostics

This option, which defaults to on when a color-capable terminal is detected, controls whether or not Clang prints diagnostics in color.

When this option is enabled, Clang will use colors to highlight specific parts of the diagnostic, e.g.,

```
test.c:28:8: warning: extra tokens at end of #endif directive [-Wextra-tokens]
#endif bad
       ^
       //
```

When this is disabled, Clang will just print:

```
test.c:2:8: warning: extra tokens at end of #endif directive [-Wextra-tokens]
#endif bad
       ^
       //
```

-fansi-escape-codes

Controls whether ANSI escape codes are used instead of the Windows Console API to output colored diagnostics. This option is only used on Windows and defaults to off.

-fdiagnostics-format=clang/msvc/vi

Changes diagnostic output format to better match IDEs and command line tools.

This option controls the output format of the filename, line number, and column printed in diagnostic messages. The options, and their affect on formatting a simple conversion diagnostic, follow:

clang (default)

```
t.c:3:11: warning: conversion specifies type 'char *' but the argument has type 'int'
```

msvc

```
t.c(3,11) : warning: conversion specifies type 'char *' but the argument has type 'int'
```

vi

```
t.c +3:11: warning: conversion specifies type 'char *' but the argument has type 'int'
```

-f[no-]diagnostics-show-option

Enable [-Woption] information in diagnostic line.

This option, which defaults to on, controls whether or not Clang prints the associated **warning group** option name when outputting a warning diagnostic. For example, in this output:

```
test.c:28:8: warning: extra tokens at end of #endif directive [-Wextra-tokens]
#endif bad
       ^
       //
```

Passing **-fno-diagnostics-show-option** will prevent Clang from printing the [**-Wextra-tokens**] information in the diagnostic. This information tells you the flag needed to enable or disable the diagnostic, either from the command line or through **#pragma GCC diagnostic**.

-fdiagnostics-show-category=none/id/name

Enable printing category information in diagnostic line.

This option, which defaults to "none", controls whether or not Clang prints the category associated with a diagnostic when emitting it. Each diagnostic may or many not have an associated category, if it has one, it is listed in the diagnostic categorization field of the diagnostic line (in the []'s).

For example, a format string warning will produce these three renditions based on the setting of this option:

```
t.c:3:11: warning: conversion specifies type 'char *' but the argument has type 'int' [-Wformat]
t.c:3:11: warning: conversion specifies type 'char *' but the argument has type 'int' [-Wformat,1]
t.c:3:11: warning: conversion specifies type 'char *' but the argument has type 'int' [-Wformat,Format String]
```

This category can be used by clients that want to group diagnostics by category, so it should be a high level category. We want dozens of these, not hundreds or thousands of them.

-fsave-optimization-record

Write optimization remarks to a YAML file.

This option, which defaults to off, controls whether Clang writes optimization reports to a YAML file. By recording diagnostics in a file, using a structured YAML format, users can parse or sort the remarks in a convenient way.

-foptimization-record-file

Control the file to which optimization reports are written.

When optimization reports are being output (see **-fsave-optimization-record**), this option controls the file to which those reports are written.

If this option is not used, optimization records are output to a file named after the primary file being compiled. If that's "foo.c", for example, optimization records are output to "foo.opt.yaml".

-f[no-]diagnostics-show-hotness

Enable profile hotness information in diagnostic line.

This option controls whether Clang prints the profile hotness associated with diagnostics in the presence of profile-guided optimization information. This is currently supported with optimization remarks (see **Options to Emit Optimization Reports**). The hotness information allows users to focus on the hot optimization remarks that are likely to be more relevant for run-time performance.

For example, in this output, the block containing the callsite of *foo* was executed 3000 times according to the profile data:

```
s.c:7:10: remark: foo inlined into bar (hotness: 3000) [-Rpass-analysis=inline]
  sum += foo(x, x - 2);
         ^
```

This option is implied when **-fsave-optimization-record** is used. Otherwise, it defaults to off.

-fdiagnostics-hotness-threshold

Prevent optimization remarks from being output if they do not have at least this hotness value.

This option, which defaults to zero, controls the minimum hotness an optimization remark would need in order to be output by Clang. This is currently supported with optimization remarks (see **Options to Emit Optimization Reports**) when profile hotness information in diagnostics is enabled (see **-fdiagnostics-show-hotness**).

-f[no-]diagnostics-fixit-info

Enable "FixIt" information in the diagnostics output.

This option, which defaults to on, controls whether or not Clang prints the information on how to fix a specific diagnostic underneath it when it knows. For example, in this output:

```
test.c:28:8: warning: extra tokens at end of #endif directive [-Wextra-tokens]
#endif bad
       ^
       //
```

Passing **-fno-diagnostics-fixit-info** will prevent Clang from printing the "//" line at the end of the message. This information

is useful for users who may not understand what is wrong, but can be confusing for machine parsing.

-fdiagnostics-print-source-range-info

Print machine parsable information about source ranges. This option makes Clang print information about source ranges in a machine parsable format after the file/line/column number information. The information is a simple sequence of brace enclosed ranges, where each range lists the start and end line/column locations. For example, in this output:

```
exprs.c:47:15:{47:8-47:14}{47:17-47:24}: error: invalid operands to binary expression ('int *' and '_Complex float')
  P = (P-42) + Gamma*4;
      ~~~~~~ ^ ~~~~~~~
```

The {}'s are generated by -fdiagnostics-print-source-range-info.

The printed column numbers count bytes from the beginning of the line; take care if your source contains multibyte characters.

-fdiagnostics-parseable-fixits

Print Fix-Its in a machine parseable form.

This option makes Clang print available Fix-Its in a machine parseable format at the end of diagnostics. The following example illustrates the format:

```
fix-it:"t.cpp":{7:25-7:29}:"Gamma"
```

The range printed is a half-open range, so in this example the characters at column 25 up to but not including column 29 on line 7 in t.cpp should be replaced with the string "Gamma". Either the range or the replacement string may be empty (representing strict insertions and strict erasures, respectively). Both the file name and the insertion string escape backslash (as "\\"), tabs (as "\t"), newlines (as "\n"), double quotes(as "\"") and non-printable characters (as octal "\xxx").

The printed column numbers count bytes from the beginning of the line; take care if your source contains multibyte characters.

-fno-elide-type

Turns off elision in template type printing.

The default for template type printing is to elide as many template arguments as possible, removing those which are the same in both template types, leaving only the differences. Adding this flag will print all the template arguments. If supported by the terminal, highlighting will still appear on differing arguments.

Default:

```
t.cc:4:5: note: candidate function not viable: no known conversion from 'vector<map<[...], map<float, [...]>>>' to 'vector<map<[...], map<doub
```

-fno-elide-type:

```
t.cc:4:5: note: candidate function not viable: no known conversion from 'vector<map<int, map<float, int>>>' to 'vector<map<int, map<double
```

-fdiagnostics-show-template-tree

Template type diffing prints a text tree.

For diffing large templated types, this option will cause Clang to display the templates as an indented text tree, one argument per line, with differences marked inline. This is compatible with -fno-elide-type.

Default:

```
t.cc:4:5: note: candidate function not viable: no known conversion from 'vector<map<[...], map<float, [...]>>>' to 'vector<map<[...], map<doub
```

With -fdiagnostics-show-template-tree:

```
t.cc:4:5: note: candidate function not viable: no known conversion for 1st argument;
 vector<
   map<
     [...],
     map<
       [float != double],
       [...]>>>
```

Individual Warning Groups

TODO: Generate this from tblgen. Define one anchor per warning group.

-Wextra-tokens

Warn about excess tokens at the end of a preprocessor directive.

This option, which defaults to on, enables warnings about extra tokens at the end of preprocessor directives. For example:

```
test.c:28:8: warning: extra tokens at end of #endif directive [-Wextra-tokens]
#endif bad
       ^
```

These extra tokens are not strictly conforming, and are usually best handled by commenting them out.

-Wambiguous-member-template

Warn about unqualified uses of a member template whose name resolves to another template at the location of the use.

This option, which defaults to on, enables a warning in the following code:

```
template<typename T> struct set{};
template<typename T> struct trait { typedef const T& type; };
struct Value {
  template<typename T> void set(typename trait<T>::type value) {}
};
void foo() {
  Value v;
  v.set<double>(3.2);
}
```

C++ [basic.lookup.classref] requires this to be an error, but, because it's hard to work around, Clang downgrades it to a warning as an extension.

-Wbind-to-temporary-copy

Warn about an unusable copy constructor when binding a reference to a temporary.

This option enables warnings about binding a reference to a temporary when the temporary doesn't have a usable copy constructor. For example:

```
struct NonCopyable {
  NonCopyable();
private:
  NonCopyable(const NonCopyable&);
};
void foo(const NonCopyable&);
```

```
void bar() {
  foo(NonCopyable());  // Disallowed in C++98; allowed in C++11.
}
```

```
struct NonCopyable2 {
  NonCopyable2();
  NonCopyable2(NonCopyable2&);
};
void foo(const NonCopyable2&);
void bar() {
  foo(NonCopyable2());  // Disallowed in C++98; allowed in C++11.
}
```

Note that if NonCopyable2::NonCopyable2() has a default argument whose instantiation produces a compile error, that error will still be a hard error in C++98 mode even if this warning is turned off.

Options to Control Clang Crash Diagnostics

As unbelievable as it may sound, Clang does crash from time to time. Generally, this only occurs to those living on the **bleeding edge**. Clang goes to great lengths to assist you in filing a bug report. Specifically, Clang generates preprocessed source file(s) and associated run script(s) upon a crash. These files should be attached to a bug report to ease reproducibility of the failure. Below are the command line options to control the crash diagnostics.

-fno-crash-diagnostics
: Disable auto-generation of preprocessed source files during a clang crash.

The -fno-crash-diagnostics flag can be helpful for speeding the process of generating a delta reduced test case.

Clang is also capable of generating preprocessed source file(s) and associated run script(s) even without a crash. This is specially useful when trying to generate a reproducer for warnings or errors while using modules.

-gen-reproducer
: Generates preprocessed source files, a reproducer script and if relevant, a cache containing: built module pcm's and all headers needed to rebuilt the same modules.

Options to Emit Optimization Reports

Optimization reports trace, at a high-level, all the major decisions done by compiler transformations. For instance, when the inliner decides to inline function foo() into bar(), or the loop unroller decides to unroll a loop N times, or the vectorizer decides to vectorize a loop body.

Clang offers a family of flags which the optimizers can use to emit a diagnostic in three cases:

1. When the pass makes a transformation (*-Rpass*).
2. When the pass fails to make a transformation (*-Rpass-missed*).
3. When the pass determines whether or not to make a transformation (*-Rpass-analysis*).

NOTE: Although the discussion below focuses on *-Rpass*, the exact same options apply to *-Rpass-missed* and *-Rpass-analysis*.

Since there are dozens of passes inside the compiler, each of these flags take a regular expression that identifies the name of the pass which should emit the associated diagnostic. For example, to get a report from the inliner, compile the code with:

```
$ clang -O2 -Rpass=inline code.cc -o code
code.cc:4:25: remark: foo inlined into bar [-Rpass=inline]
int bar(int j) { return foo(j, j - 2); }
                        ^
```

Note that remarks from the inliner are identified with *[-Rpass=inline]*. To request a report from every optimization pass, you should

use *-Rpass=.** (in fact, you can use any valid POSIX regular expression). However, do not expect a report from every transformation made by the compiler. Optimization remarks do not really make sense outside of the major transformations (e.g., inlining, vectorization, loop optimizations) and not every optimization pass supports this feature.

Note that when using profile-guided optimization information, profile hotness information can be included in the remarks (see **-fdiagnostics-show-hotness**).

Current limitations

1. Optimization remarks that refer to function names will display the mangled name of the function. Since these remarks are emitted by the back end of the compiler, it does not know anything about the input language, nor its mangling rules.
2. Some source locations are not displayed correctly. The front end has a more detailed source location tracking than the locations included in the debug info (e.g., the front end can locate code inside macro expansions). However, the locations used by *-Rpass* are translated from debug annotations. That translation can be lossy, which results in some remarks having no location information.

Other Options

Clang options that don't fit neatly into other categories.

-MV

When emitting a dependency file, use formatting conventions appropriate for NMake or Jom. Ignored unless another option causes Clang to emit a dependency file.

When Clang emits a dependency file (e.g., you supplied the -M option) most filenames can be written to the file without any special formatting. Different Make tools will treat different sets of characters as "special" and use different conventions for telling the Make tool that the character is actually part of the filename. Normally Clang uses backslash to "escape" a special character, which is the convention used by GNU Make. The -MV option tells Clang to put double-quotes around the entire filename, which is the convention used by NMake and Jom.

Language and Target-Independent Features

Controlling Errors and Warnings

Clang provides a number of ways to control which code constructs cause it to emit errors and warning messages, and how they are displayed to the console.

Controlling How Clang Displays Diagnostics

When Clang emits a diagnostic, it includes rich information in the output, and gives you fine-grain control over which information is printed. Clang has the ability to print this information, and these are the options that control it:

1. A file/line/column indicator that shows exactly where the diagnostic occurs in your code [**-fshow-column**, **-fshow-source-location**].
2. A categorization of the diagnostic as a note, warning, error, or fatal error.
3. A text string that describes what the problem is.
4. An option that indicates how to control the diagnostic (for diagnostics that support it) [**-fdiagnostics-show-option**].
5. A **high-level category** for the diagnostic for clients that want to group diagnostics by class (for diagnostics that support it) [**-fdiagnostics-show-category**].
6. The line of source code that the issue occurs on, along with a caret and ranges that indicate the important locations [**-fcaret-diagnostics**].
7. "FixIt" information, which is a concise explanation of how to fix the problem (when Clang is certain it knows) [**-fdiagnostics-fixit-info**].
8. A machine-parsable representation of the ranges involved (off by default) [**-fdiagnostics-print-source-range-info**].

For more information please see **Formatting of Diagnostics**.

Diagnostic Mappings

All diagnostics are mapped into one of these 6 classes:

- Ignored
- Note
- Remark
- Warning
- Error
- Fatal

Diagnostic Categories

Though not shown by default, diagnostics may each be associated with a high-level category. This category is intended to make it possible to triage builds that produce a large number of errors or warnings in a grouped way.

Categories are not shown by default, but they can be turned on with the **-fdiagnostics-show-category** option. When set to "name", the category is printed textually in the diagnostic output. When it is set to "id", a category number is printed. The mapping of category names to category id's can be obtained by running 'clang --print-diagnostic-categories'.

Controlling Diagnostics via Command Line Flags

TODO: -W flags, -pedantic, etc

Controlling Diagnostics via Pragmas

Clang can also control what diagnostics are enabled through the use of pragmas in the source code. This is useful for turning off specific warnings in a section of source code. Clang supports GCC's pragma for compatibility with existing source code, as well as several extensions.

The pragma may control any warning that can be used from the command line. Warnings may be set to ignored, warning, error, or fatal. The following example code will tell Clang or GCC to ignore the -Wall warnings:

```
#pragma GCC diagnostic ignored "-Wall"
```

In addition to all of the functionality provided by GCC's pragma, Clang also allows you to push and pop the current warning state. This is particularly useful when writing a header file that will be compiled by other people, because you don't know what warning flags they build with.

In the below example -Wextra-tokens is ignored for only a single line of code, after which the diagnostics return to whatever state had previously existed.

```
#if foo
#endif foo // warning: extra tokens at end of #endif directive

#pragma clang diagnostic push
#pragma clang diagnostic ignored "-Wextra-tokens"

#if foo
#endif foo // no warning

#pragma clang diagnostic pop
```

The push and pop pragmas will save and restore the full diagnostic state of the compiler, regardless of how it was set. That means that it is possible to use push and pop around GCC compatible diagnostics and Clang will push and pop them appropriately, while GCC will ignore the pushes and pops as unknown pragmas. It should be noted that while Clang supports the GCC pragma, Clang and GCC do not support the exact same set of warnings, so even when using GCC compatible #pragmas there is no guarantee that they will have identical behaviour on both compilers.

In addition to controlling warnings and errors generated by the compiler, it is possible to generate custom warning and error messages through the following pragmas:

```
// The following will produce warning messages
#pragma message "some diagnostic message"
#pragma GCC warning "TODO: replace deprecated feature"

// The following will produce an error message
#pragma GCC error "Not supported"
```

These pragmas operate similarly to the #warning and #error preprocessor directives, except that they may also be embedded into preprocessor macros via the C99 _Pragma operator, for example:

```
#define STR(X) #X
#define DEFER(M,...) M(__VA_ARGS__)
#define CUSTOM_ERROR(X) _Pragma(STR(GCC error(X " at line " DEFER(STR,__LINE__))))

CUSTOM_ERROR("Feature not available");
```

Controlling Diagnostics in System Headers

Warnings are suppressed when they occur in system headers. By default, an included file is treated as a system header if it is found in an include path specified by -isystem, but this can be overridden in several ways.

The system_header pragma can be used to mark the current file as being a system header. No warnings will be produced from the location of the pragma onwards within the same file.

```
#if foo
#endif foo // warning: extra tokens at end of #endif directive

#pragma clang system_header

#if foo
#endif foo // no warning
```

The *-system-header-prefix=* and *-no-system-header-prefix=* command-line arguments can be used to override whether subsets of an include path are treated as system headers. When the name in a #include directive is found within a header search path and starts with a system prefix, the header is treated as a system header. The last prefix on the command-line which matches the specified header name takes precedence. For instance:

```
$ clang -Ifoo -isystem bar --system-header-prefix=x/ \
    --no-system-header-prefix=x/y/
```

Here, #include "x/a.h" is treated as including a system header, even if the header is found in foo, and #include "x/y/b.h" is treated as not including a system header, even if the header is found in bar.

A #include directive which finds a file relative to the current directory is treated as including a system header if the including file is treated as a system header.

Enabling All Diagnostics

In addition to the traditional -W flags, one can enable **all** diagnostics by passing -Weverything. This works as expected with -Werror, and also includes the warnings from -pedantic.

Note that when combined with -w (which disables all warnings), that flag wins.

Controlling Static Analyzer Diagnostics

While not strictly part of the compiler, the diagnostics from Clang's **static analyzer** can also be influenced by the user via changes to the source code. See the available **annotations** and the analyzer's **FAQ page** for more information.

Precompiled Headers

Precompiled headers are a general approach employed by many compilers to reduce compilation time. The underlying motivation of the approach is that it is common for the same (and often large) header files to be included by multiple source files. Consequently, compile times can often be greatly improved by caching some of the (redundant) work done by a compiler to process headers. Precompiled header files, which represent one of many ways to implement this optimization, are literally files that represent an on-disk cache that contains the vital information necessary to reduce some of the work needed to process a corresponding header file. While details of precompiled headers vary between compilers, precompiled headers have been shown to be highly effective at speeding up program compilation on systems with very large system headers (e.g., Mac OS X).

Generating a PCH File

To generate a PCH file using Clang, one invokes Clang with the *-x <language>-header* option. This mirrors the interface in GCC for generating PCH files:

```
$ gcc -x c-header test.h -o test.h.gch
$ clang -x c-header test.h -o test.h.pch
```

Using a PCH File

A PCH file can then be used as a prefix header when a -include option is passed to clang:

```
$ clang -include test.h test.c -o test
```

The clang driver will first check if a PCH file for test.h is available; if so, the contents of test.h (and the files it includes) will be processed from the PCH file. Otherwise, Clang falls back to directly processing the content of test.h. This mirrors the behavior of GCC.

Note

Clang does *not* automatically use PCH files for headers that are directly included within a source file. For example:

```
$ clang -x c-header test.h -o test.h.pch
$ cat test.c
#include "test.h"
$ clang test.c -o test
```

In this example, clang will not automatically use the PCH file for test.h since test.h was included directly in the source file and not specified on the command line using -include.

Relocatable PCH Files

It is sometimes necessary to build a precompiled header from headers that are not yet in their final, installed locations. For example, one might build a precompiled header within the build tree that is then meant to be installed alongside the headers. Clang permits the creation of "relocatable" precompiled headers, which are built with a given path (into the build directory) and can later be used from an installed location.

To build a relocatable precompiled header, place your headers into a subdirectory whose structure mimics the installed location. For example, if you want to build a precompiled header for the header mylib.h that will be installed into /usr/include, create a

subdirectory build/usr/include and place the header mylib.h into that subdirectory. If mylib.h depends on other headers, then they can be stored within build/usr/include in a way that mimics the installed location.

Building a relocatable precompiled header requires two additional arguments. First, pass the --relocatable-pch flag to indicate that the resulting PCH file should be relocatable. Second, pass *-isysroot /path/to/build*, which makes all includes for your library relative to the build directory. For example:

```
# clang -x c-header --relocatable-pch -isysroot /path/to/build /path/to/build/mylib.h mylib.h.pch
```

When loading the relocatable PCH file, the various headers used in the PCH file are found from the system header root. For example, mylib.h can be found in /usr/include/mylib.h. If the headers are installed in some other system root, the *-isysroot* option can be used provide a different system root from which the headers will be based. For example, *-isysroot /Developer /SDKs/MacOSX10.4u.sdk* will look for mylib.h in /Developer/SDKs/MacOSX10.4u.sdk/usr/include/mylib.h.

Relocatable precompiled headers are intended to be used in a limited number of cases where the compilation environment is tightly controlled and the precompiled header cannot be generated after headers have been installed.

Controlling Code Generation

Clang provides a number of ways to control code generation. The options are listed below.

-f[no-]sanitize=check1,check2,...

Turn on runtime checks for various forms of undefined or suspicious behavior.

This option controls whether Clang adds runtime checks for various forms of undefined or suspicious behavior, and is disabled by default. If a check fails, a diagnostic message is produced at runtime explaining the problem. The main checks are:

-fsanitize=address: **AddressSanitizer**, a memory error detector.

-fsanitize=thread: **ThreadSanitizer**, a data race detector.

-fsanitize=memory: **MemorySanitizer**, a detector of uninitialized reads. Requires instrumentation of all program code.

-fsanitize=undefined: **UndefinedBehaviorSanitizer**, a fast and compatible undefined behavior checker.

-fsanitize=dataflow: **DataFlowSanitizer**, a general data flow analysis.
-fsanitize=cfi: **control flow integrity** checks. Requires -flto.
-fsanitize=safe-stack: **safe stack** protection against stack-based memory corruption errors.

There are more fine-grained checks available: see the **list** of specific kinds of undefined behavior that can be detected and the **list** of control flow integrity schemes.

The -fsanitize= argument must also be provided when linking, in order to link to the appropriate runtime library.

It is not possible to combine more than one of the -fsanitize=address, -fsanitize=thread, and -fsanitize=memory checkers in the same program.

-f[no-]sanitize-recover=check1,check2,...

-f[no-]sanitize-recover=all

Controls which checks enabled by -fsanitize= flag are non-fatal. If the check is fatal, program will halt after the first error of this kind is detected and error report is printed.

By default, non-fatal checks are those enabled by **UndefinedBehaviorSanitizer**, except for -fsanitize=return and -fsanitize=unreachable. Some sanitizers may not support recovery (or not support it by default e.g. **AddressSanitizer**), and always crash the program after the issue is detected.

Note that the -fsanitize-trap flag has precedence over this flag. This means that if a check has been configured to trap elsewhere on the command line, or if the check traps by default, this flag will not have any effect unless that sanitizer's trapping behavior is disabled with -fno-sanitize-trap.

For example, if a command line contains the flags -fsanitize=undefined -fsanitize-trap=undefined, the flag -fsanitize-

recover=alignment will have no effect on its own; it will need to be accompanied by -fno-sanitize-trap=alignment.

-f[no-]sanitize-trap=check1,check2,...

Controls which checks enabled by the -fsanitize= flag trap. This option is intended for use in cases where the sanitizer runtime cannot be used (for instance, when building libc or a kernel module), or where the binary size increase caused by the sanitizer runtime is a concern.

This flag is only compatible with **control flow integrity** schemes and **UndefinedBehaviorSanitizer** checks other than vptr. If this flag is supplied together with -fsanitize=undefined, the vptr sanitizer will be implicitly disabled.

This flag is enabled by default for sanitizers in the cfi group.

-fsanitize-blacklist=/path/to/blacklist/file

Disable or modify sanitizer checks for objects (source files, functions, variables, types) listed in the file. See **Sanitizer special case list** for file format description.

-fno-sanitize-blacklist

Don't use blacklist file, if it was specified earlier in the command line.

-f[no-]sanitize-coverage=[type,features,...]

Enable simple code coverage in addition to certain sanitizers. See **SanitizerCoverage** for more details.

-f[no-]sanitize-stats

Enable simple statistics gathering for the enabled sanitizers. See **SanitizerStats** for more details.

-fsanitize-undefined-trap-on-error

Deprecated alias for -fsanitize-trap=undefined.

-fsanitize-cfi-cross-dso

Enable cross-DSO control flow integrity checks. This flag modifies the behavior of sanitizers in the cfi group to allow checking of cross-DSO virtual and indirect calls.

-fstrict-vtable-pointers

Enable optimizations based on the strict rules for overwriting polymorphic C++ objects, i.e. the vptr is invariant during an object's lifetime. This enables better devirtualization. Turned off by default, because it is still experimental.

-ffast-math

Enable fast-math mode. This defines the __FAST_MATH__ preprocessor macro, and lets the compiler make aggressive, potentially-lossy assumptions about floating-point math. These include:

Floating-point math obeys regular algebraic rules for real numbers (e.g. + and * are associative, x/y == x * (1/y), and (a + b) * c == a * c + b * c),
operands to floating-point operations are not equal to NaN and Inf, and
+0 and -0 are interchangeable.

-fdenormal-fp-math=[values]

Select which denormal numbers the code is permitted to require.

Valid values are: ieee, preserve-sign, and positive-zero, which correspond to IEEE 754 denormal numbers, the sign of a flushed-to-zero number is preserved in the sign of 0, denormals are flushed to positive zero, respectively.

-fwhole-program-vtables

Enable whole-program vtable optimizations, such as single-implementation devirtualization and virtual constant propagation, for classes with **hidden LTO visibility**. Requires -flto.

-fno-assume-sane-operator-new

Don't assume that the C++'s new operator is sane.

This option tells the compiler to do not assume that C++'s global new operator will always return a pointer that does not alias any other pointer when the function returns.

-ftrap-function=[name]

Instruct code generator to emit a function call to the specified function name for __builtin_trap().

LLVM code generator translates __builtin_trap() to a trap instruction if it is supported by the target ISA. Otherwise, the builtin is translated into a call to abort. If this option is set, then the code generator will always lower the builtin to a call to the specified function regardless of whether the target ISA has a trap instruction. This option is useful for environments (e.g. deeply embedded) where a trap cannot be properly handled, or when some custom behavior is desired.

-ftls-model=[model]

Select which TLS model to use.

Valid values are: global-dynamic, local-dynamic, initial-exec and local-exec. The default value is global-dynamic. The compiler may use a different model if the selected model is not supported by the target, or if a more efficient model can be used. The TLS model can be overridden per variable using the tls_model attribute.

-femulated-tls

Select emulated TLS model, which overrides all -ftls-model choices.

In emulated TLS mode, all access to TLS variables are converted to calls to __emutls_get_address in the runtime library.

-mhwdiv=[values]

Select the ARM modes (arm or thumb) that support hardware division instructions.

Valid values are: arm, thumb and arm,thumb. This option is used to indicate which mode (arm or thumb) supports hardware division instructions. This only applies to the ARM architecture.

-m[no-]crc

Enable or disable CRC instructions.

This option is used to indicate whether CRC instructions are to be generated. This only applies to the ARM architecture.

CRC instructions are enabled by default on ARMv8.

-mgeneral-regs-only

Generate code which only uses the general purpose registers.

This option restricts the generated code to use general registers only. This only applies to the AArch64 architecture.

-mcompact-branches=[values]

Control the usage of compact branches for MIPSR6.

Valid values are: never, optimal and always. The default value is optimal which generates compact branches when a delay slot cannot be filled. never disables the usage of compact branches and always generates compact branches whenever possible.

-f[no-]max-type-align=[number]

Instruct the code generator to not enforce a higher alignment than the given number (of bytes) when accessing memory via an opaque pointer or reference. This cap is ignored when directly accessing a variable or when the pointee type has an explicit "aligned" attribute.

The value should usually be determined by the properties of the system allocator. Some builtin types, especially vector types, have very high natural alignments; when working with values of those types, Clang usually wants to use instructions that take advantage of that alignment. However, many system allocators do not promise to return memory that is more than 8-byte or 16-byte-aligned. Use this option to limit the alignment that the compiler can assume for an arbitrary pointer, which may point onto the heap.

This option does not affect the ABI alignment of types; the layout of structs and unions and the value returned by the alignof operator remain the same.

This option can be overridden on a case-by-case basis by putting an explicit "aligned" alignment on a struct, union, or typedef. For example:

```
#include <immintrin.h>
// Make an aligned typedef of the AVX-512 16-int vector type.
typedef __v16si __aligned_v16si __attribute__((aligned(64)));

void initialize_vector(__aligned_v16si *v) {
  // The compiler may assume that 'v' is 64-byte aligned, regardless of the
  // value of -fmax-type-align.
}
```

Profile Guided Optimization

Profile information enables better optimization. For example, knowing that a branch is taken very frequently helps the compiler make better decisions when ordering basic blocks. Knowing that a function foo is called more frequently than another function bar helps the inliner.

Clang supports profile guided optimization with two different kinds of profiling. A sampling profiler can generate a profile with very low runtime overhead, or you can build an instrumented version of the code that collects more detailed profile information. Both kinds of profiles can provide execution counts for instructions in the code and information on branches taken and function invocation.

Regardless of which kind of profiling you use, be careful to collect profiles by running your code with inputs that are representative of the typical behavior. Code that is not exercised in the profile will be optimized as if it is unimportant, and the compiler may make poor optimization choices for code that is disproportionately used while profiling.

Differences Between Sampling and Instrumentation

Although both techniques are used for similar purposes, there are important differences between the two:

1. Profile data generated with one cannot be used by the other, and there is no conversion tool that can convert one to the other. So, a profile generated via -fprofile-instr-generate must be used with -fprofile-instr-use. Similarly, sampling profiles generated by external profilers must be converted and used with -fprofile-sample-use.
2. Instrumentation profile data can be used for code coverage analysis and optimization.
3. Sampling profiles can only be used for optimization. They cannot be used for code coverage analysis. Although it would be technically possible to use sampling profiles for code coverage, sample-based profiles are too coarse-grained for code coverage purposes; it would yield poor results.
4. Sampling profiles must be generated by an external tool. The profile generated by that tool must then be converted into a format that can be read by LLVM. The section on sampling profilers describes one of the supported sampling profile formats.

Using Sampling Profilers

Sampling profilers are used to collect runtime information, such as hardware counters, while your application executes. They are typically very efficient and do not incur a large runtime overhead. The sample data collected by the profiler can be used during compilation to determine what the most executed areas of the code are.

Using the data from a sample profiler requires some changes in the way a program is built. Before the compiler can use profiling information, the code needs to execute under the profiler. The following is the usual build cycle when using sample profilers for optimization:

1. Build the code with source line table information. You can use all the usual build flags that you always build your application with. The only requirement is that you add -gline-tables-only or -g to the command line. This is important for the profiler to be able to map instructions back to source line locations.

   ```
   $ clang++ -O2 -gline-tables-only code.cc -o code
   ```

2. Run the executable under a sampling profiler. The specific profiler you use does not really matter, as long as its output can be converted into the format that the LLVM optimizer understands. Currently, there exists a conversion tool for the Linux Perf

profiler (**https://perf.wiki.kernel.org/**), so these examples assume that you are using Linux Perf to profile your code.

```
$ perf record -b ./code
```

Note the use of the -b flag. This tells Perf to use the Last Branch Record (LBR) to record call chains. While this is not strictly required, it provides better call information, which improves the accuracy of the profile data.

3. Convert the collected profile data to LLVM's sample profile format. This is currently supported via the AutoFDO converter create_llvm_prof. It is available at **http://github.com/google/autofdo**. Once built and installed, you can convert the perf.data file to LLVM using the command:

```
$ create_llvm_prof --binary=./code --out=code.prof
```

This will read perf.data and the binary file ./code and emit the profile data in code.prof. Note that if you ran perf without the -b flag, you need to use --use_lbr=false when calling create_llvm_prof.

4. Build the code again using the collected profile. This step feeds the profile back to the optimizers. This should result in a binary that executes faster than the original one. Note that you are not required to build the code with the exact same arguments that you used in the first step. The only requirement is that you build the code with -gline-tables-only and -fprofile-sample-use.

```
$ clang++ -O2 -gline-tables-only -fprofile-sample-use=code.prof code.cc -o code
```

Sample Profile Formats

Since external profilers generate profile data in a variety of custom formats, the data generated by the profiler must be converted into a format that can be read by the backend. LLVM supports three different sample profile formats:

1. ASCII text. This is the easiest one to generate. The file is divided into sections, which correspond to each of the functions with profile information. The format is described below. It can also be generated from the binary or gcov formats using the llvm-profdata tool.
2. Binary encoding. This uses a more efficient encoding that yields smaller profile files. This is the format generated by the create_llvm_prof tool in **http://github.com/google/autofdo**.
3. GCC encoding. This is based on the gcov format, which is accepted by GCC. It is only interesting in environments where GCC and Clang co-exist. This encoding is only generated by the create_gcov tool in **http://github.com/google/autofdo**. It can be read by LLVM and llvm-profdata, but it cannot be generated by either.

If you are using Linux Perf to generate sampling profiles, you can use the conversion tool create_llvm_prof described in the previous section. Otherwise, you will need to write a conversion tool that converts your profiler's native format into one of these three.

Sample Profile Text Format

This section describes the ASCII text format for sampling profiles. It is, arguably, the easiest one to generate. If you are interested in generating any of the other two, consult the ProfileData library in LLVM's source tree (specifically, include/llvm/ProfileData/SampleProfReader.h).

```
function1:total_samples:total_head_samples
 offset1[.discriminator]: number_of_samples [fn1:num fn2:num ... ]
 offset2[.discriminator]: number_of_samples [fn3:num fn4:num ... ]
 ...
 offsetN[.discriminator]: number_of_samples [fn5:num fn6:num ... ]
 offsetA[.discriminator]: fnA:num_of_total_samples
  offsetA1[.discriminator]: number_of_samples [fn7:num fn8:num ... ]
  offsetA1[.discriminator]: number_of_samples [fn9:num fn10:num ... ]
  offsetB[.discriminator]: fnB:num_of_total_samples
   offsetB1[.discriminator]: number_of_samples [fn11:num fn12:num ... ]
```

This is a nested tree in which the indentation represents the nesting level of the inline stack. There are no blank lines in the file. And the spacing within a single line is fixed. Additional spaces will result in an error while reading the file.

Any line starting with the '#' character is completely ignored.

Inlined calls are represented with indentation. The Inline stack is a stack of source locations in which the top of the stack represents the leaf function, and the bottom of the stack represents the actual symbol to which the instruction belongs.

Function names must be mangled in order for the profile loader to match them in the current translation unit. The two numbers in the function header specify how many total samples were accumulated in the function (first number), and the total number of samples accumulated in the prologue of the function (second number). This head sample count provides an indicator of how frequently the function is invoked.

There are two types of lines in the function body.

Sampled line represents the profile information of a source location. offsetN[.discriminator]: number_of_samples [fn5:num fn6:num ...]
Callsite line represents the profile information of an inlined callsite. offsetA[.discriminator]: fnA:num_of_total_samples

Each sampled line may contain several items. Some are optional (marked below):

a. Source line offset. This number represents the line number in the function where the sample was collected. The line number is always relative to the line where symbol of the function is defined. So, if the function has its header at line 280, the offset 13 is at line 293 in the file.

   Note that this offset should never be a negative number. This could happen in cases like macros. The debug machinery will register the line number at the point of macro expansion. So, if the macro was expanded in a line before the start of the function, the profile converter should emit a 0 as the offset (this means that the optimizers will not be able to associate a meaningful weight to the instructions in the macro).

b. [OPTIONAL] Discriminator. This is used if the sampled program was compiled with DWARF discriminator support (**http://wiki.dwarfstd.org/index.php?title=Path_Discriminators**). DWARF discriminators are unsigned integer values that allow the compiler to distinguish between multiple execution paths on the same source line location.

   For example, consider the line of code if (cond) foo(); else bar();. If the predicate cond is true 80% of the time, then the edge into function foo should be considered to be taken most of the time. But both calls to foo and bar are at the same source line, so a sample count at that line is not sufficient. The compiler needs to know which part of that line is taken more frequently.

   This is what discriminators provide. In this case, the calls to foo and bar will be at the same line, but will have different discriminator values. This allows the compiler to correctly set edge weights into foo and bar.

c. Number of samples. This is an integer quantity representing the number of samples collected by the profiler at this source location.

d. [OPTIONAL] Potential call targets and samples. If present, this line contains a call instruction. This models both direct and number of samples. For example,

```
130: 7  foo:3  bar:2  baz:7
```

   The above means that at relative line offset 130 there is a call instruction that calls one of foo(), bar() and baz(), with baz() being the relatively more frequently called target.

As an example, consider a program with the call chain main -> foo -> bar. When built with optimizations enabled, the compiler may inline the calls to bar and foo inside main. The generated profile could then be something like this:

```
main:35504:0
1: _Z3foov:35504
 2: _Z32bari:31977
 1.1: 31977
2: 0
```

This profile indicates that there were a total of 35,504 samples collected in main. All of those were at line 1 (the call to foo). Of those, 31,977 were spent inside the body of bar. The last line of the profile (2: 0) corresponds to line 2 inside main. No samples were collected there.

Profiling with Instrumentation

Clang also supports profiling via instrumentation. This requires building a special instrumented version of the code and has some

runtime overhead during the profiling, but it provides more detailed results than a sampling profiler. It also provides reproducible results, at least to the extent that the code behaves consistently across runs.

Here are the steps for using profile guided optimization with instrumentation:

1. Build an instrumented version of the code by compiling and linking with the -fprofile-instr-generate option.

```
$ clang++ -O2 -fprofile-instr-generate code.cc -o code
```

2. Run the instrumented executable with inputs that reflect the typical usage. By default, the profile data will be written to a default.profraw file in the current directory. You can override that default by using option -fprofile-instr-generate= or by setting the LLVM_PROFILE_FILE environment variable to specify an alternate file. If non-default file name is specified by both the environment variable and the command line option, the environment variable takes precedence. The file name pattern specified can include different modifiers: %p, %h, and %m.

   Any instance of %p in that file name will be replaced by the process ID, so that you can easily distinguish the profile output from multiple runs.

```
$ LLVM_PROFILE_FILE="code-%p.profraw" ./code
```

   The modifier %h can be used in scenarios where the same instrumented binary is run in multiple different host machines dumping profile data to a shared network based storage. The %h specifier will be substituted with the hostname so that profiles collected from different hosts do not clobber each other.

   While the use of %p specifier can reduce the likelihood for the profiles dumped from different processes to clobber each other, such clobbering can still happen because of the pid re-use by the OS. Another side-effect of using %p is that the storage requirement for raw profile data files is greatly increased. To avoid issues like this, the %m specifier can used in the profile name. When this specifier is used, the profiler runtime will substitute %m with a unique integer identifier associated with the instrumented binary. Additionally, multiple raw profiles dumped from different processes that share a file system (can be on different hosts) will be automatically merged by the profiler runtime during the dumping. If the program links in multiple instrumented shared libraries, each library will dump the profile data into its own profile data file (with its unique integer id embedded in the profile name). Note that the merging enabled by %m is for raw profile data generated by profiler runtime. The resulting merged "raw" profile data file still needs to be converted to a different format expected by the compiler (see step 3 below).

```
$ LLVM_PROFILE_FILE="code-%m.profraw" ./code
```

3. Combine profiles from multiple runs and convert the "raw" profile format to the input expected by clang. Use the merge command of the llvm-profdata tool to do this.

```
$ llvm-profdata merge -output=code.profdata code-*.profraw
```

   Note that this step is necessary even when there is only one "raw" profile, since the merge operation also changes the file format.

4. Build the code again using the -fprofile-instr-use option to specify the collected profile data.

```
$ clang++ -O2 -fprofile-instr-use=code.profdata code.cc -o code
```

   You can repeat step 4 as often as you like without regenerating the profile. As you make changes to your code, clang may no longer be able to use the profile data. It will warn you when this happens.

Profile generation using an alternative instrumentation method can be controlled by the GCC-compatible flags -fprofile-generate and -fprofile-use. Although these flags are semantically equivalent to their GCC counterparts, they *do not* handle GCC-compatible profiles. They are only meant to implement GCC's semantics with respect to profile creation and use.

-fprofile-generate[=<dirname>]

> The -fprofile-generate and -fprofile-generate= flags will use an alterantive instrumentation method for profile generation. When given a directory name, it generates the profile file default_%m.profraw in the directory named dirname if specified. If dirname does not exist, it will be created at runtime. %m specifier will be substibuted with a

unique id documented in step 2 above. In other words, with -fprofile-generate[=<dirname>] option, the "raw" profile data automatic merging is turned on by default, so there will no longer any risk of profile clobbering from different running processes. For example,

```
$ clang++ -O2 -fprofile-generate=yyy/zzz code.cc -o code
```

When code is executed, the profile will be written to the file yyy/zzz/default_xxxx.profraw.

To generate the profile data file with the compiler readable format, the llvm-profdata tool can be used with the profile directory as the input:

```
$ llvm-profdata merge -output=code.profdata yyy/zzz/
```

If the user wants to turn off the auto-merging feature, or simply override the the profile dumping path specified at command line, the environment variable LLVM_PROFILE_FILE can still be used to override the directory and filename for the profile file at runtime.

-fprofile-use[=<pathname>]

Without any other arguments, -fprofile-use behaves identically to -fprofile-instr-use. Otherwise, if pathname is the full path to a profile file, it reads from that file. If pathname is a directory name, it reads from pathname/default.profdata.

Disabling Instrumentation

In certain situations, it may be useful to disable profile generation or use for specific files in a build, without affecting the main compilation flags used for the other files in the project.

In these cases, you can use the flag -fno-profile-instr-generate (or -fno-profile-generate) to disable profile generation, and -fno-profile-instr-use (or -fno-profile-use) to disable profile use.

Note that these flags should appear after the corresponding profile flags to have an effect.

Controlling Debug Information

Controlling Size of Debug Information

Debug info kind generated by Clang can be set by one of the flags listed below. If multiple flags are present, the last one is used.

-g0

Don't generate any debug info (default).

-gline-tables-only

Generate line number tables only.

This kind of debug info allows to obtain stack traces with function names, file names and line numbers (by such tools as gdb or addr2line). It doesn't contain any other data (e.g. description of local variables or function parameters).

-fstandalone-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. Note that Clang will never emit type information for types that are not referenced at all by the program.

-fno-standalone-debug

On Darwin **-fstandalone-debug** is enabled by default. The **-fno-standalone-debug** option can be used to get to turn on the vtable-based optimization described above.

-g
Generate complete debug info.

Controlling Macro Debug Info Generation

Debug info for C preprocessor macros increases the size of debug information in the binary. Macro debug info generated by Clang can be controlled by the flags listed below.

-fdebug-macro
Generate debug info for preprocessor macros. This flag is discarded when **-g0** is enabled.

-fno-debug-macro
Do not generate debug info for preprocessor macros (default).

Controlling Debugger "Tuning"

While Clang generally emits standard DWARF debug info (**http://dwarfstd.org**), different debuggers may know how to take advantage of different specific DWARF features. You can "tune" the debug info for one of several different debuggers.

-ggdb, -glldb, -gsce
Tune the debug info for the gdb, lldb, or Sony PlayStation® debugger, respectively. Each of these options implies **-g**. (Therefore, if you want both **-gline-tables-only** and debugger tuning, the tuning option must come first.)

Comment Parsing Options

Clang parses Doxygen and non-Doxygen style documentation comments and attaches them to the appropriate declaration nodes. By default, it only parses Doxygen-style comments and ignores ordinary comments starting with // and /*.

-Wdocumentation
Emit warnings about use of documentation comments. This warning group is off by default.

This includes checking that \param commands name parameters that actually present in the function signature, checking that \returns is used only on functions that actually return a value etc.

-Wno-documentation-unknown-command
Don't warn when encountering an unknown Doxygen command.

-fparse-all-comments
Parse all comments as documentation comments (including ordinary comments starting with // and /*).

-fcomment-block-commands=[commands]
Define custom documentation commands as block commands. This allows Clang to construct the correct AST for these custom commands, and silences warnings about unknown commands. Several commands must be separated by a comma *without trailing space*; e.g. -fcomment-block-commands=foo,bar defines custom commands \foo and \bar.

It is also possible to use -fcomment-block-commands several times; e.g. -fcomment-block-commands=foo -fcomment-block-commands=bar does the same as above.

C Language Features

The support for standard C in clang is feature-complete except for the C99 floating-point pragmas.

Extensions supported by clang

See **Clang Language Extensions**.

Differences between various standard modes

clang supports the -std option, which changes what language mode clang uses. The supported modes for C are c89, gnu89, c94, c99, gnu99, c11, gnu11, and various aliases for those modes. If no -std option is specified, clang defaults to gnu11 mode. Many C99 and C11 features are supported in earlier modes as a conforming extension, with a warning. Use -pedantic-errors to request an error if a feature from a later standard revision is used in an earlier mode.

Differences between all c* and gnu* modes:

- c* modes define "__STRICT_ANSI__".
- Target-specific defines not prefixed by underscores, like "linux", are defined in gnu* modes.
- Trigraphs default to being off in gnu* modes; they can be enabled by the -trigraphs option.
- The parser recognizes "asm" and "typeof" as keywords in gnu* modes; the variants "__asm__" and "__typeof__" are recognized in all modes.
- The Apple "blocks" extension is recognized by default in gnu* modes on some platforms; it can be enabled in any mode with the "-fblocks" option.
- Arrays that are VLA's according to the standard, but which can be constant folded by the frontend are treated as fixed size arrays. This occurs for things like "int X[(1, 2)];", which is technically a VLA. c* modes are strictly compliant and treat these as VLAs.

Differences between *89 and *99 modes:

- The *99 modes default to implementing "inline" as specified in C99, while the *89 modes implement the GNU version. This can be overridden for individual functions with the __gnu_inline__ attribute.
- Digraphs are not recognized in c89 mode.
- The scope of names defined inside a "for", "if", "switch", "while", or "do" statement is different. (example: "if ((struct x {int x;}*)0) {}".)
- __STDC_VERSION__ is not defined in *89 modes.
- "inline" is not recognized as a keyword in c89 mode.
- "restrict" is not recognized as a keyword in *89 modes.
- Commas are allowed in integer constant expressions in *99 modes.
- Arrays which are not lvalues are not implicitly promoted to pointers in *89 modes.
- Some warnings are different.

Differences between *99 and *11 modes:

- Warnings for use of C11 features are disabled.
- __STDC_VERSION__ is defined to 201112L rather than 199901L.

c94 mode is identical to c89 mode except that digraphs are enabled in c94 mode (FIXME: And __STDC_VERSION__ should be defined!).

GCC extensions not implemented yet

clang tries to be compatible with gcc as much as possible, but some gcc extensions are not implemented yet:

- clang does not support decimal floating point types (_Decimal32 and friends) or fixed-point types (_Fract and friends); nobody has expressed interest in these features yet, so it's hard to say when they will be implemented.
- clang does not support nested functions; this is a complex feature which is infrequently used, so it is unlikely to be implemented anytime soon. In C++11 it can be emulated by assigning lambda functions to local variables, e.g:

```
auto const local_function = [&](int parameter) {
  // Do something
};
...
local_function(1);
```

- clang only supports global register variables when the register specified is non-allocatable (e.g. the stack pointer). Support for

general global register variables is unlikely to be implemented soon because it requires additional LLVM backend support.

clang does not support static initialization of flexible array members. This appears to be a rarely used extension, but could be implemented pending user demand.

clang does not support __builtin_va_arg_pack/__builtin_va_arg_pack_len. This is used rarely, but in some potentially interesting places, like the glibc headers, so it may be implemented pending user demand. Note that because clang pretends to be like GCC 4.2, and this extension was introduced in 4.3, the glibc headers will not try to use this extension with clang at the moment.

clang does not support the gcc extension for forward-declaring function parameters; this has not shown up in any real-world code yet, though, so it might never be implemented.

This is not a complete list; if you find an unsupported extension missing from this list, please send an e-mail to cfe-dev. This list currently excludes C++; see **C++ Language Features**. Also, this list does not include bugs in mostly-implemented features; please see the **bug tracker** for known existing bugs (FIXME: Is there a section for bug-reporting guidelines somewhere?).

Intentionally unsupported GCC extensions

clang does not support the gcc extension that allows variable-length arrays in structures. This is for a few reasons: one, it is tricky to implement, two, the extension is completely undocumented, and three, the extension appears to be rarely used. Note that clang *does* support flexible array members (arrays with a zero or unspecified size at the end of a structure).

clang does not have an equivalent to gcc's "fold"; this means that clang doesn't accept some constructs gcc might accept in contexts where a constant expression is required, like "x-x" where x is a variable.

clang does not support __builtin_apply and friends; this extension is extremely obscure and difficult to implement reliably.

Microsoft extensions

clang has support for many extensions from Microsoft Visual C++. To enable these extensions, use the -fms-extensions command-line option. This is the default for Windows targets. Clang does not implement every pragma or declspec provided by MSVC, but the popular ones, such as __declspec(dllexport) and #pragma comment(lib) are well supported.

clang has a -fms-compatibility flag that makes clang accept enough invalid C++ to be able to parse most Microsoft headers. For example, it allows **unqualified lookup of dependent base class members**, which is a common compatibility issue with clang. This flag is enabled by default for Windows targets.

-fdelayed-template-parsing lets clang delay parsing of function template definitions until the end of a translation unit. This flag is enabled by default for Windows targets.

For compatibility with existing code that compiles with MSVC, clang defines the _MSC_VER and _MSC_FULL_VER macros. These default to the values of 1800 and 180000000 respectively, making clang look like an early release of Visual C++ 2013. The -fms-compatibility-version= flag overrides these values. It accepts a dotted version tuple, such as 19.00.23506. Changing the MSVC compatibility version makes clang behave more like that version of MSVC. For example, -fms-compatibility-version=19 will enable C++14 features and define char16_t and char32_t as builtin types.

C++ Language Features

clang fully implements all of standard C++98 except for exported templates (which were removed in C++11), and all of standard C++11 and the current draft standard for C++1y.

Controlling implementation limits

-fbracket-depth=N

Sets the limit for nested parentheses, brackets, and braces to N. The default is 256.

-fconstexpr-depth=N

Sets the limit for recursive constexpr function invocations to N. The default is 512.

-ftemplate-depth=N

Sets the limit for recursively nested template instantiations to N. The default is 256.

-foperator-arrow-depth=N

Sets the limit for iterative calls to 'operator->' functions to N. The default is 256.

Objective-C Language Features

Objective-C++ Language Features

OpenMP Features

Clang supports all OpenMP 3.1 directives and clauses. In addition, some features of OpenMP 4.0 are supported. For example, #pragma omp simd, #pragma omp for simd, #pragma omp parallel for simd directives, extended set of atomic constructs, proc_bind clause for all parallel-based directives, depend clause for #pragma omp task directive (except for array sections), #pragma omp cancel and #pragma omp cancellation point directives, and #pragma omp taskgroup directive.

Use *-fopenmp* to enable OpenMP. Support for OpenMP can be disabled with *-fno-openmp*.

Controlling implementation limits

-fopenmp-use-tls

Controls code generation for OpenMP threadprivate variables. In presence of this option all threadprivate variables are generated the same way as thread local variables, using TLS support. If *-fno-openmp-use-tls* is provided or target does not support TLS, code generation for threadprivate variables relies on OpenMP runtime library.

OpenCL Features

Clang can be used to compile OpenCL kernels for execution on a device (e.g. GPU). It is possible to compile the kernel into a binary (e.g. for AMD or Nvidia targets) that can be uploaded to run directly on a device (e.g. using **clCreateProgramWithBinary**) or into generic bitcode files loadable into other toolchains.

Compiling to a binary using the default target from the installation can be done as follows:

```
$ echo "kernel void k(){}" > test.cl
$ clang test.cl
```

Compiling for a specific target can be done by specifying the triple corresponding to the target, for example:

```
$ clang -target nvptx64-unknown-unknown test.cl
$ clang -target amdgcn-amd-amdhsa-opencl test.cl
```

Compiling to bitcode can be done as follows:

```
$ clang -c -emit-llvm test.cl
```

This will produce a generic test.bc file that can be used in vendor toolchains to perform machine code generation.

Clang currently supports OpenCL C language standards up to v2.0.

OpenCL Specific Options

Most of the OpenCL build options from **the specification v2.0 section 5.8.4** are available.

Examples:

```
$ clang -cl-std=CL2.0 -cl-single-precision-constant test.cl
```

Some extra options are available to support special OpenCL features.

-finclude-default-header

Loads standard includes during compilations. By default OpenCL headers are not loaded and therefore standard library includes are not available. To load them automatically a flag has been added to the frontend (see also **the section on the OpenCL Header**):

```
$ clang -Xclang -finclude-default-header test.cl
```

Alternatively -include or -I followed by the path to the header location can be given manually.

```
$ clang -I<path to clang>/lib/Headers/opencl-c.h test.cl
```

In this case the kernel code should contain #include <opencl-c.h> just as a regular C include.

-cl-ext

Disables support of OpenCL extensions. All OpenCL targets provide a list of extensions that they support. Clang allows to amend this using the -cl-ext flag with a comma-separated list of extensions prefixed with '+' or '-'. The syntax: -cl-ext=<(['-'|'+']<extension>[,])+>, where extensions can be either one of **the OpenCL specification extensions** or any known vendor extension. Alternatively, 'all' can be used to enable or disable all known extensions. Example disabling double support for the 64-bit SPIR target:

```
$ clang -cc1 -triple spir64-unknown-unknown -cl-ext=-cl_khr_fp64 test.cl
```

Enabling all extensions except double support in R600 AMD GPU can be done using:

```
$ clang -cc1 -triple r600-unknown-unknown -cl-ext=-all,+cl_khr_fp16 test.cl
```

-ffake-address-space-map

Overrides the target address space map with a fake map. This allows adding explicit address space IDs to the bitcode for non-segmented memory architectures that don't have separate IDs for each of the OpenCL logical address spaces by default. Passing -ffake-address-space-map will add/override address spaces of the target compiled for with the following values: 1-global, 2-constant, 3-local, 4-generic. The private address space is represented by the absence of an address space attribute in the IR (see also **the section on the address space attribute**).

```
$ clang -ffake-address-space-map test.cl
```

Some other flags used for the compilation for C can also be passed while compiling for OpenCL, examples: -c, -O<1-4|s>, -o, -emit-llvm, etc.

OpenCL Targets

OpenCL targets are derived from the regular Clang target classes. The OpenCL specific parts of the target representation provide address space mapping as well as a set of supported extensions.

Specific Targets

There is a set of concrete HW architectures that OpenCL can be compiled for.

For AMD target:

```
$ clang -target amdgcn-amd-amdhsa-opencl test.cl
```

For Nvidia architectures:

```
$ clang -target nvptx64-unknown-unknown test.cl
```

Generic Targets

SPIR is available as a generic target to allow portable bitcode to be produced that can be used across GPU toolchains. The implementation follows **the SPIR specification**. There are two flavors available for 32 and 64 bits.

```
$ clang -target spir-unknown-unknown test.cl
$ clang -target spir64-unknown-unknown test.cl
```

All known OpenCL extensions are supported in the SPIR targets. Clang will generate SPIR v1.2 compatible IR for OpenCL versions up to 2.0 and SPIR v2.0 for OpenCL v2.0.

x86 is used by some implementations that are x86 compatible and currently remains for backwards compatibility (with older implementations prior to SPIR target support). For "non-SPMD" targets which cannot spawn multiple work-items on the fly using hardware, which covers practically all non-GPU devices such as CPUs and DSPs, additional processing is needed for the kernels to support multiple work-item execution. For this, a 3rd party toolchain, such as for example **POCL**, can be used.

This target does not support multiple memory segments and, therefore, the fake address space map can be added using the **-ffake-address-space-map** flag.

OpenCL Header

By default Clang will not include standard headers and therefore OpenCL builtin functions and some types (i.e. vectors) are unknown. The default CL header is, however, provided in the Clang installation and can be enabled by passing the -finclude-default-header flag to the Clang frontend.

```
$ echo "bool is_wg_uniform(int i){return get_enqueued_local_size(i)==get_local_size(i);}" > test.cl
$ clang -Xclang -finclude-default-header -cl-std=CL2.0 test.cl
```

Because the header is very large and long to parse, PCH (**Precompiled Header and Modules Internals**) and modules (**Modules**) are used internally to improve the compilation speed.

To enable modules for OpenCL:

```
$ clang -target spir-unknown-unknown -c -emit-llvm -Xclang -finclude-default-header -fmodules -fimplicit-module-maps -fmodules-ca
```

OpenCL Extensions

All of the cl_khr_* extensions from **the official OpenCL specification** up to and including version 2.0 are available and set per target depending on the support available in the specific architecture.

It is possible to alter the default extensions setting per target using -cl-ext flag. (See **flags description** for more details).

Vendor extensions can be added flexibly by declaring the list of types and functions associated with each extensions enclosed within the following compiler pragma directives:

```
#pragma OPENCL EXTENSION the_new_extension_name : begin
// declare types and functions associated with the extension here
#pragma OPENCL EXTENSION the_new_extension_name : end
```

For example, parsing the following code adds my_t type and my_func function to the custom my_ext extension.

```
#pragma OPENCL EXTENSION my_ext : begin
typedef struct{
  int a;
}my_t;
void my_func(my_t);
#pragma OPENCL EXTENSION my_ext : end
```

Declaring the same types in different vendor extensions is disallowed.

OpenCL Metadata

Clang uses metadata to provide additional OpenCL semantics in IR needed for backends and OpenCL runtime.

Each kernel will have function metadata attached to it, specifying the arguments. Kernel argument metadata is used to provide source level information for querying at runtime, for example using the **clGetKernelArgInfo** call.

Note that -cl-kernel-arg-info enables more information about the original CL code to be added e.g. kernel parameter names will appear in the OpenCL metadata along with other information.

The IDs used to encode the OpenCL's logical address spaces in the argument info metadata follows the SPIR address space mapping as defined in the SPIR specification **section 2.2**

OpenCL-Specific Attributes

OpenCL support in Clang contains a set of attribute taken directly from the specification as well as additional attributes.

See also **Attributes in Clang**.

nosvm

Clang supports this attribute to comply to OpenCL v2.0 conformance, but it does not have any effect on the IR. For more details reffer to the specification **section 6.7.2**

opencl_unroll_hint

The implementation of this feature mirrors the unroll hint for C. More details on the syntax can be found in the specification **section 6.11.5**

convergent

To make sure no invalid optimizations occur for single program multiple data (SPMD) / single instruction multiple thread (SIMT) Clang provides attributes that can be used for special functions that have cross work item semantics. An example is the subgroup operations such as **intel_sub_group_shuffle**

```
// Define custom my_sub_group_shuffle(data, c)
// that makes use of intel_sub_group_shuffle
r1 = ...
if (r0) r1 = computeA();
// Shuffle data from r1 into r3
// of threads id r2.
r3 = my_sub_group_shuffle(r1, r2);
if (r0) r3 = computeB();
```

with non-SPMD semantics this is optimized to the following equivalent code:

```
r1 = ...
if (!r0)
  // Incorrect functionality! The data in r1
  // have not been computed by all threads yet.
  r3 = my_sub_group_shuffle(r1, r2);
else {
  r1 = computeA();
  r3 = my_sub_group_shuffle(r1, r2);
  r3 = computeB();
}
```

Declaring the function my_sub_group_shuffle with the convergent attribute would prevent this:

```
my_sub_group_shuffle() __attribute__((convergent));
```

Using convergent guarantees correct execution by keeping CFG equivalence wrt operations marked as convergent. CFG G´ is equivalent to G wrt node Ni : iff ∀ Nj (i≠j) domination and post-domination relations with respect to Ni remain the same in both G and G´.

noduplicate

noduplicate is more restrictive with respect to optimizations than convergent because a convergent function only preserves CFG equivalence. This allows some optimizations to happen as long as the control flow remains unmodified.

```
for (int i=0; i<4; i++)
  my_sub_group_shuffle()
```

can be modified to:

```
my_sub_group_shuffle();
my_sub_group_shuffle();
my_sub_group_shuffle();
my_sub_group_shuffle();
```

while using noduplicate would disallow this. Also noduplicate doesn't have the same safe semantics of CFG as convergent and can cause changes in CFG that modify semantics of the original program.

noduplicate is kept for backwards compatibility only and it considered to be deprecated for future uses.

address_space

Clang has arbitrary address space support using the address_space(N) attribute, where N is an integer number in the range 0 to 16777215 (0xffffffu).

An OpenCL implementation provides a list of standard address spaces using keywords: private, local, global, and generic. In the AST and in the IR local, global, or generic will be represented by the address space attribute with the corresponding unique number. Note that private does not have any corresponding attribute added and, therefore, is represented by the absence of an address space number. The specific IDs for an address space do not have to match between the AST and the IR. Typically in the AST address space numbers represent logical segments while in the IR they represent physical segments. Therefore, machines with flat memory segments can map all AST address space numbers to the same physical segment ID or skip address space attribute completely while generating the IR. However, if the address space information is needed by the IR passes e.g. to improve alias analysis, it is recommended to keep it and only lower to reflect physical memory segments in the late machine passes.

OpenCL builtins

There are some standard OpenCL functions that are implemented as Clang builtins:

All pipe functions from **section 6.13.16.2/6.13.16.3** of the OpenCL v2.0 kernel language specification. `
Address space qualifier conversion functions to_global/to_local/to_private from **section 6.13.9**.
All the enqueue_kernel functions from **section 6.13.17.1** and enqueue query functions from **section 6.13.17.5**.

Target-Specific Features and Limitations

CPU Architectures Features and Limitations

X86

The support for X86 (both 32-bit and 64-bit) is considered stable on Darwin (Mac OS X), Linux, FreeBSD, and Dragonfly BSD: it has been tested to correctly compile many large C, C++, Objective-C, and Objective-C++ codebases.

On x86_64-mingw32, passing i128(by value) is incompatible with the Microsoft x64 calling convention. You might need to tweak WinX86_64ABIInfo::classify() in lib/CodeGen/TargetInfo.cpp.

For the X86 target, clang supports the *-m16* command line argument which enables 16-bit code output. This is broadly similar to using asm(".code16gcc") with the GNU toolchain. The generated code and the ABI remains 32-bit but the assembler emits instructions appropriate for a CPU running in 16-bit mode, with address-size and operand-size prefixes to enable 32-bit addressing and operations.

ARM

The support for ARM (specifically ARMv6 and ARMv7) is considered stable on Darwin (iOS): it has been tested to correctly compile many large C, C++, Objective-C, and Objective-C++ codebases. Clang only supports a limited number of ARM architectures. It does not yet fully support ARMv5, for example.

PowerPC

The support for PowerPC (especially PowerPC64) is considered stable on Linux and FreeBSD: it has been tested to correctly compile many large C and C++ codebases. PowerPC (32bit) is still missing certain features (e.g. PIC code on ELF platforms).

Other platforms

clang currently contains some support for other architectures (e.g. Sparc); however, significant pieces of code generation are still missing, and they haven't undergone significant testing.

clang contains limited support for the MSP430 embedded processor, but both the clang support and the LLVM backend support are highly experimental.

Other platforms are completely unsupported at the moment. Adding the minimal support needed for parsing and semantic analysis on a new platform is quite easy; see lib/Basic/Targets.cpp in the clang source tree. This level of support is also sufficient for conversion to LLVM IR for simple programs. Proper support for conversion to LLVM IR requires adding code to lib/CodeGen/CGCall.cpp at the moment; this is likely to change soon, though. Generating assembly requires a suitable LLVM backend.

Operating System Features and Limitations

Darwin (Mac OS X)

Thread Sanitizer is not supported.

Windows

Clang has experimental support for targeting "Cygming" (Cygwin / MinGW) platforms.

See also **Microsoft Extensions**.

Cygwin

Clang works on Cygwin-1.7.

MinGW32

Clang works on some mingw32 distributions. Clang assumes directories as below;

C:/mingw/include
C:/mingw/lib
C:/mingw/lib/gcc/mingw32/4.[3-5].0/include/c++

On MSYS, a few tests might fail.

MinGW-w64

For 32-bit (i686-w64-mingw32), and 64-bit (x86_64-w64-mingw32), Clang assumes as below;

GCC versions 4.5.0 to 4.5.3, 4.6.0 to 4.6.2, or 4.7.0 (for the C++ header search path)
some_directory/bin/gcc.exe
some_directory/bin/clang.exe
some_directory/bin/clang++.exe
some_directory/bin/../include/c++/GCC_version
some_directory/bin/../include/c++/GCC_version/x86_64-w64-mingw32
some_directory/bin/../include/c++/GCC_version/i686-w64-mingw32
some_directory/bin/../include/c++/GCC_version/backward
some_directory/bin/../x86_64-w64-mingw32/include
some_directory/bin/../i686-w64-mingw32/include
some_directory/bin/../include

This directory layout is standard for any toolchain you will find on the official **MinGW-w64 website**.

Clang expects the GCC executable "gcc.exe" compiled for i686-w64-mingw32 (or x86_64-w64-mingw32) to be present on PATH.

Some tests might fail on x86_64-w64-mingw32.

clang-cl

clang-cl is an alternative command-line interface to Clang, designed for compatibility with the Visual C++ compiler, cl.exe.

To enable clang-cl to find system headers, libraries, and the linker when run from the command-line, it should be executed inside a Visual Studio Native Tools Command Prompt or a regular Command Prompt where the environment has been set up using e.g. **vcvars32.bat**.

clang-cl can also be used from inside Visual Studio by using an LLVM Platform Toolset.

Command-Line Options

To be compatible with cl.exe, clang-cl supports most of the same command-line options. Those options can start with either / or -. It also supports some of Clang's core options, such as the -W options.

Options that are known to clang-cl, but not currently supported, are ignored with a warning. For example:

```
clang-cl.exe: warning: argument unused during compilation: '/AI'
```

To suppress warnings about unused arguments, use the -Qunused-arguments option.

Options that are not known to clang-cl will be ignored by default. Use the -Werror=unknown-argument option in order to treat them

as errors. If these options are spelled with a leading /, they will be mistaken for a filename:

```
clang-cl.exe: error: no such file or directory: '/foobar'
```

Please **file a bug** for any valid cl.exe flags that clang-cl does not understand.

Execute clang-cl /? to see a list of supported options:

```
CL.EXE COMPATIBILITY OPTIONS:
  /?                      Display available options
  /arch:<value>           Set architecture for code generation
  /Brepro-                Emit an object file which cannot be reproduced over time
  /Brepro                 Emit an object file which can be reproduced over time
  /C                      Don't discard comments when preprocessing
  /c                      Compile only
  /d1reportAllClassLayout Dump record layout information
  /diagnostics:caret      Enable caret and column diagnostics (on by default)
  /diagnostics:classic    Disable column and caret diagnostics
  /diagnostics:column     Disable caret diagnostics but keep column info
  /D <macro[=value]>      Define macro
  /EH<value>              Exception handling model
  /EP                     Disable linemarker output and preprocess to stdout
  /execution-charset:<value>
                          Runtime encoding, supports only UTF-8
  /E                      Preprocess to stdout
  /fallback               Fall back to cl.exe if clang-cl fails to compile
  /FA                     Output assembly code file during compilation
  /Fa<file or directory>  Output assembly code to this file during compilation (with /FA)
  /Fe<file or directory>  Set output executable file or directory (ends in / or \)
  /FI <value>             Include file before parsing
  /Fi<file>               Set preprocess output file name (with /P)
  /Fo<file or directory>  Set output object file, or directory (ends in / or \) (with /c)
  /fp:except-
  /fp:except
  /fp:fast
  /fp:precise
  /fp:strict
  /Fp<filename>           Set pch filename (with /Yc and /Yu)
  /GA                     Assume thread-local variables are defined in the executable
  /Gd                     Set __cdecl as a default calling convention
  /GF-                    Disable string pooling
  /GR-                    Disable emission of RTTI data
  /GR                     Enable emission of RTTI data
  /Gr                     Set __fastcall as a default calling convention
  /GS-                    Disable buffer security check
  /GS                     Enable buffer security check
  /Gs<value>              Set stack probe size
  /Gv                     Set __vectorcall as a default calling convention
  /Gw-                    Don't put each data item in its own section
  /Gw                     Put each data item in its own section
  /GX-                    Enable exception handling
  /GX                     Enable exception handling
  /Gy-                    Don't put each function in its own section
  /Gy                     Put each function in its own section
  /Gz                     Set __stdcall as a default calling convention
  /help                   Display available options
```

```
/imsvc <dir>           Add directory to system include search path, as if part of %INCLUDE%
/I <dir>               Add directory to include search path
/J                     Make char type unsigned
/LDd                   Create debug DLL
/LD                    Create DLL
/link <options>        Forward options to the linker
/MDd                   Use DLL debug run-time
/MD                    Use DLL run-time
/MTd                   Use static debug run-time
/MT                    Use static run-time
/Od                    Disable optimization
/Oi-                   Disable use of builtin functions
/Oi                    Enable use of builtin functions
/Os                    Optimize for size
/Ot                    Optimize for speed
/O<value>              Optimization level
/o <file or directory>  Set output file or directory (ends in / or \)
/P                     Preprocess to file
/Qvec-                 Disable the loop vectorization passes
/Qvec                  Enable the loop vectorization passes
/showIncludes          Print info about included files to stderr
/source-charset:<value> Source encoding, supports only UTF-8
/std:<value>           Language standard to compile for
/TC                    Treat all source files as C
/Tc <filename>         Specify a C source file
/TP                    Treat all source files as C++
/Tp <filename>         Specify a C++ source file
/utf-8                 Set source and runtime encoding to UTF-8 (default)
/U <macro>             Undefine macro
/vd<value>             Control vtordisp placement
/vmb                   Use a best-case representation method for member pointers
/vmg                   Use a most-general representation for member pointers
/vmm                   Set the default most-general representation to multiple inheritance
/vms                   Set the default most-general representation to single inheritance
/vmv                   Set the default most-general representation to virtual inheritance
/volatile:iso          Volatile loads and stores have standard semantics
/volatile:ms           Volatile loads and stores have acquire and release semantics
/W0                    Disable all warnings
/W1                    Enable -Wall
/W2                    Enable -Wall
/W3                    Enable -Wall
/W4                    Enable -Wall and -Wextra
/Wall                  Enable -Wall and -Wextra
/WX-                   Do not treat warnings as errors
/WX                    Treat warnings as errors
/w                     Disable all warnings
/Y-                    Disable precompiled headers, overrides /Yc and /Yu
/Yc<filename>          Generate a pch file for all code up to and including <filename>
/Yu<filename>          Load a pch file and use it instead of all code up to and including <filename>
/Z7                    Enable CodeView debug information in object files
/Zc:sizedDealloc-      Disable C++14 sized global deallocation functions
/Zc:sizedDealloc       Enable C++14 sized global deallocation functions
/Zc:strictStrings      Treat string literals as const
/Zc:threadSafeInit-    Disable thread-safe initialization of static variables
/Zc:threadSafeInit     Enable thread-safe initialization of static variables
/Zc:trigraphs-         Disable trigraphs (default)
/Zc:trigraphs          Enable trigraphs
```

```
  /Zc:twoPhase-          Disable two-phase name lookup in templates
  /Zc:twoPhase           Enable two-phase name lookup in templates
  /Zd                    Emit debug line number tables only
  /Zi                    Alias for /Z7. Does not produce PDBs.
  /Zl                    Don't mention any default libraries in the object file
  /Zp                    Set the default maximum struct packing alignment to 1
  /Zp<value>             Specify the default maximum struct packing alignment
  /Zs                    Syntax-check only

OPTIONS:
  -###                   Print (but do not run) the commands to run for this compilation
  --analyze              Run the static analyzer
  -fansi-escape-codes    Use ANSI escape codes for diagnostics
  -fcolor-diagnostics    Use colors in diagnostics
  -fdebug-macro          Emit macro debug information
  -fdelayed-template-parsing
                         Parse templated function definitions at the end of the translation unit
  -fdiagnostics-absolute-paths
                         Print absolute paths in diagnostics
  -fdiagnostics-parseable-fixits
                         Print fix-its in machine parseable form
  -flto=<value>          Set LTO mode to either 'full' or 'thin'
  -flto                  Enable LTO in 'full' mode
  -fms-compatibility-version=<value>
                         Dot-separated value representing the Microsoft compiler version
                         number to report in _MSC_VER (0 = don't define it (default))
  -fms-compatibility     Enable full Microsoft Visual C++ compatibility
  -fms-extensions        Accept some non-standard constructs supported by the Microsoft compiler
  -fmsc-version=<value>  Microsoft compiler version number to report in _MSC_VER
                         (0 = don't define it (default))
  -fno-debug-macro       Do not emit macro debug information
  -fno-delayed-template-parsing
                         Disable delayed template parsing
  -fno-sanitize-address-use-after-scope
                         Disable use-after-scope detection in AddressSanitizer
  -fno-sanitize-blacklist Don't use blacklist file for sanitizers
  -fno-sanitize-cfi-cross-dso
                         Disable control flow integrity (CFI) checks for cross-DSO calls.
  -fno-sanitize-coverage=<value>
                         Disable specified features of coverage instrumentation for Sanitizers
  -fno-sanitize-memory-track-origins
                         Disable origins tracking in MemorySanitizer
  -fno-sanitize-recover=<value>
                         Disable recovery for specified sanitizers
  -fno-sanitize-stats    Disable sanitizer statistics gathering.
  -fno-sanitize-thread-atomics
                         Disable atomic operations instrumentation in ThreadSanitizer
  -fno-sanitize-thread-func-entry-exit
                         Disable function entry/exit instrumentation in ThreadSanitizer
  -fno-sanitize-thread-memory-access
                         Disable memory access instrumentation in ThreadSanitizer
  -fno-sanitize-trap=<value>
                         Disable trapping for specified sanitizers
  -fno-standalone-debug  Limit debug information produced to reduce size of debug binary
  -fprofile-instr-generate=<file>
                         Generate instrumented code to collect execution counts into <file>
                         (overridden by LLVM_PROFILE_FILE env var)
```

```
-fprofile-instr-generate
                Generate instrumented code to collect execution counts into default.profraw file
                (overridden by '=' form of option or LLVM_PROFILE_FILE env var)
-fprofile-instr-use=<value>
                Use instrumentation data for profile-guided optimization
-fsanitize-address-field-padding=<value>
                Level of field padding for AddressSanitizer
-fsanitize-address-globals-dead-stripping
                Enable linker dead stripping of globals in AddressSanitizer
-fsanitize-address-use-after-scope
                Enable use-after-scope detection in AddressSanitizer
-fsanitize-blacklist=<value>
                Path to blacklist file for sanitizers
-fsanitize-cfi-cross-dso
                Enable control flow integrity (CFI) checks for cross-DSO calls.
-fsanitize-coverage=<value>
                Specify the type of coverage instrumentation for Sanitizers
-fsanitize-memory-track-origins=<value>
                Enable origins tracking in MemorySanitizer
-fsanitize-memory-track-origins
                Enable origins tracking in MemorySanitizer
-fsanitize-memory-use-after-dtor
                Enable use-after-destroy detection in MemorySanitizer
-fsanitize-recover=<value>
                Enable recovery for specified sanitizers
-fsanitize-stats        Enable sanitizer statistics gathering.
-fsanitize-thread-atomics
                Enable atomic operations instrumentation in ThreadSanitizer (default)
-fsanitize-thread-func-entry-exit
                Enable function entry/exit instrumentation in ThreadSanitizer (default)
-fsanitize-thread-memory-access
                Enable memory access instrumentation in ThreadSanitizer (default)
-fsanitize-trap=<value> Enable trapping for specified sanitizers
-fsanitize-undefined-strip-path-components=<number>
                Strip (or keep only, if negative) a given number of path components when emitting check metadata.
-fsanitize=<check>      Turn on runtime checks for various forms of undefined or suspicious
                behavior. See user manual for available checks
-fstandalone-debug      Emit full debug info for all types used by the program
-gcodeview              Generate CodeView debug information
-gline-tables-only      Emit debug line number tables only
-miamcu                 Use Intel MCU ABI
-mllvm <value>          Additional arguments to forward to LLVM's option processing
-nobuiltininc           Disable builtin #include directories
-Qunused-arguments      Don't emit warning for unused driver arguments
-R<remark>              Enable the specified remark
--target=<value>        Generate code for the given target
-v                      Show commands to run and use verbose output
-W<warning>             Enable the specified warning
-Xclang <arg>           Pass <arg> to the clang compiler
```

The /fallback Option

When clang-cl is run with the /fallback option, it will first try to compile files itself. For any file that it fails to compile, it will fall back and try to compile the file by invoking cl.exe.

This option is intended to be used as a temporary means to build projects where clang-cl cannot successfully compile all the files. clang-cl may fail to compile a file either because it cannot generate code for some C++ feature, or because it cannot parse some

Microsoft language extension.