
Clang Documentation

Release 3.7

The Clang Team

Aug 19, 2017

Contents

1	Introduction	3
2	What's New in Clang 3.7?	5
3	Additional Information	9
4	Using Clang as a Compiler	11
5	Using Clang as a Library	237
6	Using Clang Tools	263
7	Design Documents	277
8	Indices and tables	325

- *Introduction*
- *What's New in Clang 3.7?*
 - *Major New Features*
 - *Improvements to Clang's diagnostics*
 - *New Compiler Flags*
 - *Objective-C Language Changes in Clang*
 - *Profile Guided Optimization*
 - *OpenMP Support*
 - *Internal API Changes*
 - *Static Analyzer*
 - *clang-tidy*
 - *SystemZ*
 - *Last release which will run on Windows XP and Windows Vista*
- *Additional Information*

Written by the LLVM Team

CHAPTER 1

Introduction

This document contains the release notes for the Clang C/C++/Objective-C frontend, part of the LLVM Compiler Infrastructure, release 3.7. Here we describe the status of Clang in some detail, including major improvements from the previous release and new feature work. For the general LLVM release notes, see [the LLVM documentation](#). All LLVM releases may be downloaded from the [LLVM releases web site](#).

For more information about Clang or LLVM, including information about the latest release, please check out the main please see the [Clang Web Site](#) or the [LLVM Web Site](#).

What's New in Clang 3.7?

Some of the major new features and improvements to Clang are listed here. Generic improvements to Clang as a whole or to its underlying infrastructure are described first, followed by language-specific sections with improvements to Clang's support for those languages.

Major New Features

- Use of the `__declspec` language extension for declaration attributes now requires passing the `-fms-extensions` or `-fborland` compiler flag. This language extension is also enabled when compiling CUDA code, but its use should be viewed as an implementation detail that is subject to change.
- On Windows targets, some uses of the `__try`, `__except`, and `__finally` language constructs are supported in Clang 3.7. MSVC-compatible C++ exceptions are not yet supported, however.
- Clang 3.7 fully supports OpenMP 3.1 and reported to work on many platforms, including x86, x86-64 and Power. Also, `pragma omp simd` from OpenMP 4.0 is supported as well. See below for details.
- Clang 3.7 includes an implementation of *control flow integrity*, a security hardening mechanism.

Improvements to Clang's diagnostics

Clang's diagnostics are constantly being improved to catch more issues, explain them more clearly, and provide more accurate source information about them. The improvements since the 3.6 release include:

- `-Wrange-loop-analysis` analyzes the loop variable type and the container type to determine whether copies are made of the container elements. If possible, suggest a `const` reference type to prevent copies, or a non-reference type to indicate a copy is made.
- `-Wredundant-move` warns when a parameter variable is moved on return and the return type is the same as the variable. Returning the variable directly will already make a move, so the call is not needed.
- `-Wpessimizing-move` warns when a local variable is moved on return and the return type is the same as the variable. Copy elision cannot take place with a move, but can take place if the variable is returned directly.

- `-Wmove` is a new warning group which has the previous two warnings, `-Wredundant-move` and `-Wpessimizing-move`, as well as previous warning `-Wself-move`. In addition, this group is part of `-Wmost` and `-Wall` now.
- `-Winfinite-recursion`, a warning for functions that only call themselves, is now part of `-Wmost` and `-Wall`.
- `-Wobjc-circular-container` prevents creation of circular containers, it covers `NSMutableArray`, `NSMutableSet`, `NSMutableDictionary`, `NSMutableOrderedSet` and all their subclasses.

New Compiler Flags

The sized deallocation feature of C++14 is now controlled by the `-fsized-deallocation` flag. This feature relies on library support that isn't yet widely deployed, so the user must supply an extra flag to get the extra functionality.

Objective-C Language Changes in Clang

- `objc_boxable` attribute was added. Structs and unions marked with this attribute can be used with boxed expressions (`@(. . .)`) to create `NSValue`.

Profile Guided Optimization

Clang now accepts GCC-compatible flags for profile guided optimization (PGO). You can now use `-fprofile-generate=<dir>`, `-fprofile-use=<dir>`, `-fno-profile-generate` and `-fno-profile-use`. These flags have the same semantics as their GCC counterparts. However, the generated profile is still LLVM-specific. PGO profiles generated with Clang cannot be used by GCC and vice-versa.

Clang now emits function entry counts in profile-instrumented binaries. This has improved the computation of weights and frequencies in profile analysis.

OpenMP Support

OpenMP 3.1 is fully supported, but disabled by default. To enable it, please use the `-fopenmp=libomp` command line option. Your feedback (positive or negative) on using OpenMP-enabled clang would be much appreciated; please share it either on [cfe-dev](#) or [openmp-dev](#) mailing lists.

In addition to OpenMP 3.1, several important elements of the 4.0 version of the standard are supported as well:

- `omp simd`, `omp for simd` and `omp parallel for simd` pragmas
- atomic constructs
- `proc_bind` clause of `omp parallel` pragma
- `depend` clause of `omp task` pragma (except for array sections)
- `omp cancel` and `omp cancellation point` pragmas
- `omp taskgroup` pragma

Internal API Changes

These are major API changes that have happened since the 3.6 release of Clang. If upgrading an external codebase that uses Clang as a library, this section should help get you past the largest hurdles of upgrading.

- Some of the `PPCallbacks` interface now deals in `MacroDefinition` objects instead of `MacroDirective` objects. This allows preserving full information on macros imported from modules.
- `clang-c/Index.h` no longer `#includes clang-c/Documentation.h`. You now need to explicitly `#include "clang-c/Documentation.h"` if you use the `libclang` documentation API.

Static Analyzer

- The generated plists now contain the name of the check that generated it.
- Configuration options can now be passed to the checkers (not just the static analyzer core).
- New check for dereferencing object that the result of a zero-length allocation.
- Also check functions in precompiled headers.
- Properly handle `alloca()` in some checkers.
- Various improvements to the retain count checker.

clang-tidy

Added new checks:

- `google-global-names-in-headers`: flag global namespace pollution in header files.
- `misc-assert-side-effect`: detects `assert()` conditions with side effects which can cause different behavior in debug / release builds.
- `misc-assign-operator-signature`: finds declarations of assign operators with the wrong return and/or argument types.
- `misc-inaccurate-erase`: warns when some elements of a container are not removed due to using the `erase()` algorithm incorrectly.
- `misc-inefficient-algorithm`: warns on inefficient use of STL algorithms on associative containers.
- `misc-macro-parentheses`: finds macros that can have unexpected behavior due to missing parentheses.
- `misc-macro-repeated-side-effects`: checks for repeated argument with side effects in macros.
- `misc-noexcept-move-constructor`: flags user-defined move constructors and assignment operators not marked with `noexcept` or marked with `noexcept(expr)` where `expr` evaluates to `false` (but is not a `false` literal itself).
- `misc-static-assert`: replaces `assert()` with `static_assert()` if the condition is evaluable at compile time.
- `readability-container-size-empty`: checks whether a call to the `size()` method can be replaced with a call to `empty()`.
- `readability-else-after-return`: flags conditional statements having the `else` branch, when the `true` branch has a `return` as the last statement.

- `readability-redundant-string-cstr`: finds unnecessary calls to `std::string::c_str()`.
- `readability-shrink-to-fit`: replaces copy and swap tricks on shrinkable containers with the `shrink_to_fit()` method call.
- `readability-simplify-boolean-expr`: looks for boolean expressions involving boolean constants and simplifies them to use the appropriate boolean expression directly (`if (x == true) ... -> if (x)`, etc.)

SystemZ

- Clang will now always default to the z10 processor when compiling without any `-march=` option. Previous releases used to automatically detect the current host CPU when compiling natively. If you wish to still have clang detect the current host CPU, you now need to use the `-march=native` option.
- Clang now provides the `<s390intrin.h>` header file.
- Clang now supports the transactional-execution facility and provides associated builtins and the `<htmintrin.h>` and `<htmxlintrin.h>` header files. Support is enabled by default on zEC12 and above, and can additionally be enabled or disabled via the `-mhtm / -mno-htm` command line options.
- Clang now supports the vector facility. This includes a change in the ABI to pass arguments and return values of vector types in vector registers, as well as a change in the default alignment of vector types. Support is enabled by default on z13 and above, and can additionally be enabled or disabled via the `-mvx / -mno-vx` command line options.
- Clang now supports the System z vector language extension, providing a “vector” keyword to define vector types, and a set of builtins defined in the `<vecintrin.h>` header file. This can be enabled via the `-fzvector` command line option. For compatibility with GCC, Clang also supports the `-mzvector` option as an alias.
- Several cases of ABI incompatibility with GCC have been fixed.

Last release which will run on Windows XP and Windows Vista

This is expected to be the last major release of Clang that will support running on Windows XP and Windows Vista. For the next major release the minimum Windows version requirement will be Windows 7.

CHAPTER 3

Additional Information

A wide variety of additional information is available on the [Clang web page](#). The web page contains versions of the API documentation which are up-to-date with the Subversion version of the source code. You can access versions of these documents specific to this release by going into the “clang/docs/” directory in the Clang tree.

If you have any questions or comments about Clang, please feel free to contact us via the [mailing list](#).

Clang Compiler User's Manual

- *Introduction*
 - *Terminology*
 - *Basic Usage*
- *Command Line Options*
 - *Options to Control Error and Warning Messages*
 - * *Formatting of Diagnostics*
 - * *Individual Warning Groups*
 - *Options to Control Clang Crash Diagnostics*
 - *Options to Emit Optimization Reports*
 - * *Current limitations*
 - *Other Options*
- *Language and Target-Independent Features*
 - *Controlling Errors and Warnings*
 - * *Controlling How Clang Displays Diagnostics*
 - * *Diagnostic Mappings*
 - * *Diagnostic Categories*
 - * *Controlling Diagnostics via Command Line Flags*
 - * *Controlling Diagnostics via Pragmas*

- * *Controlling Diagnostics in System Headers*
 - * *Enabling All Diagnostics*
 - * *Controlling Static Analyzer Diagnostics*
- *Precompiled Headers*
 - * *Generating a PCH File*
 - * *Using a PCH File*
 - * *Relocatable PCH Files*
- *Controlling Code Generation*
- *Profile Guided Optimization*
 - * *Differences Between Sampling and Instrumentation*
 - * *Using Sampling Profilers*
 - *Sample Profile Formats*
 - *Sample Profile Text Format*
 - * *Profiling with Instrumentation*
- *Controlling Size of Debug Information*
- *Comment Parsing Options*
- *C Language Features*
 - *Extensions supported by clang*
 - *Differences between various standard modes*
 - *GCC extensions not implemented yet*
 - *Intentionally unsupported GCC extensions*
 - *Microsoft extensions*
- *C++ Language Features*
 - *Controlling implementation limits*
- *Objective-C Language Features*
- *Objective-C++ Language Features*
- *Target-Specific Features and Limitations*
 - *CPU Architectures Features and Limitations*
 - * *X86*
 - * *ARM*
 - * *PowerPC*
 - * *Other platforms*
 - *Operating System Features and Limitations*
 - * *Darwin (Mac OS X)*
 - * *Windows*

- *Cygwin*
- *MinGW32*
- *MinGW-w64*
- *clang-cl*
 - *Command-Line Options*
 - * *The /fallback Option*

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 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*

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”.

-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 by another program that wants to parse simple and consistent output, not a person. 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.,

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.

-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<double, [...]>>>'
↪for 1st argument;
```

`-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, int>>>' for 1st
↪argument;
```

`-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<double, [...]>>>'
↪for 1st argument;
```

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.

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.

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 `-Wmultichar` is ignored for only a single line of code, after which the diagnostics return to whatever state had previously existed.

```
#pragma clang diagnostic push
#pragma clang diagnostic ignored "-Wmultichar"

char b = 'df'; // 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.

```
char a = 'xy'; // warning

#pragma clang system_header

char b = 'ab'; // 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=integer`: Enables checks for undefined or suspicious integer behavior.
- `-fsanitize=thread`: *ThreadSanitizer*, a data race detector.
- `-fsanitize=memory`: *MemorySanitizer*, an *experimental* detector of uninitialized reads. Not ready for widespread use.

- `-fsanitize=undefined`: Fast and compatible undefined behavior checker. Enables the undefined behavior checks that have small runtime cost and no impact on address space layout or ABI. This includes all of the checks listed below other than `unsigned-integer-overflow`.
- `-fsanitize=undefined-trap`: This is a deprecated alias for `-fsanitize=undefined`.
- `-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.

The following more fine-grained checks are also available:

- `-fsanitize=alignment`: Use of a misaligned pointer or creation of a misaligned reference.
- `-fsanitize=bool`: Load of a `bool` value which is neither `true` nor `false`.
- `-fsanitize=bounds`: Out of bounds array indexing, in cases where the array bound can be statically determined.
- `-fsanitize=cfi-cast-strict`: Enables *strict cast checks*.
- `-fsanitize=cfi-derived-cast`: Base-to-derived cast to the wrong dynamic type. Requires `-flto`.
- `-fsanitize=cfi-unrelated-cast`: Cast from `void*` or another unrelated type to the wrong dynamic type. Requires `-flto`.
- `-fsanitize=cfi-nvcall`: Non-virtual call via an object whose `vp`tr is of the wrong dynamic type. Requires `-flto`.
- `-fsanitize=cfi-vcall`: Virtual call via an object whose `vp`tr is of the wrong dynamic type. Requires `-flto`.
- `-fsanitize=enum`: Load of a value of an enumerated type which is not in the range of representable values for that enumerated type.
- `-fsanitize=float-cast-overflow`: Conversion to, from, or between floating-point types which would overflow the destination.
- `-fsanitize=float-divide-by-zero`: Floating point division by zero.
- `-fsanitize=function`: Indirect call of a function through a function pointer of the wrong type (Linux, C++ and x86/x86_64 only).
- `-fsanitize=integer-divide-by-zero`: Integer division by zero.
- `-fsanitize=nonnull-attribute`: Passing null pointer as a function parameter which is declared to never be null.
- `-fsanitize=null`: Use of a null pointer or creation of a null reference.
- `-fsanitize=object-size`: An attempt to use bytes which the optimizer can determine are not part of the object being accessed. The sizes of objects are determined using `__builtin_object_size`, and consequently may be able to detect more problems at higher optimization levels.
- `-fsanitize=return`: In C++, reaching the end of a value-returning function without returning a value.
- `-fsanitize=returns-nonnull-attribute`: Returning null pointer from a function which is declared to never return null.
- `-fsanitize=shift`: Shift operators where the amount shifted is greater or equal to the promoted bit-width of the left hand side or less than zero, or where the left hand side is negative. For a signed left shift, also checks for signed overflow in C, and for unsigned overflow in C++. You can use

`-fsanitize=shift-base` or `-fsanitize=shift-exponent` to check only left-hand side or right-hand side of shift operation, respectively.

- `-fsanitize=signed-integer-overflow`: Signed integer overflow, including all the checks added by `-ftrapv`, and checking for overflow in signed division (`INT_MIN / -1`).
- `-fsanitize=unreachable`: If control flow reaches `__builtin_unreachable`.
- `-fsanitize=unsigned-integer-overflow`: Unsigned integer overflows.
- `-fsanitize=vla-bound`: A variable-length array whose bound does not evaluate to a positive value.
- `-fsanitize=vptr`: Use of an object whose `vptr` indicates that it is of the wrong dynamic type, or that its lifetime has not begun or has ended. Incompatible with `-fno-rtti`.

You can turn off or modify checks for certain source files, functions or even variables by providing a special file:

- `-fsanitize-blacklist=/path/to/blacklist/file`: disable or modify sanitizer checks for objects 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.

Extra features of MemorySanitizer (require explicit `-fsanitize=memory`):

- `-fsanitize-memory-track-origins[=level]`: Enables origin tracking in MemorySanitizer. Adds a second section to MemorySanitizer reports pointing to the heap or stack allocation the uninitialized bits came from. Slows down execution by additional 1.5x-2x.

Possible values for level are 0 (off), 1, 2 (default). Level 2 adds more sections to MemorySanitizer reports describing the order of memory stores the uninitialized value went through. This mode may use extra memory in programs that copy uninitialized memory a lot.

The `-fsanitize=` argument must also be provided when linking, in order to link to the appropriate runtime library. When using `-fsanitize=vptr` (or a group that includes it, such as `-fsanitize=undefined`) with a C++ program, the link must be performed by `clang++`, not `clang`, in order to link against the C++-specific parts of the 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. The `-fsanitize=undefined` checks can only be combined with `-fsanitize=address`.

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

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 (e.g. [AddressSanitizer](#)) may not support recovery, 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 `local-bounds`, `unsigned-integer-overflow`, sanitizers in the `cfi` group and sanitizers in the `undefined` group other than `vptra`. If this flag is supplied together with `-fsanitize=undefined`, the `vptra` sanitizer will be implicitly disabled.

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

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

Enable simple code coverage in addition to certain sanitizers. See [SanitizerCoverage](#) for more details.

-fsanitize-undefined-trap-on-error

Deprecated alias for `-fsanitize-trap=undefined`.

-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.

-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.

-f[no-]max-unknown-pointer-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-unknown-pointer-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.
2. Binary encoding. This uses a more efficient encoding that yields smaller profile files, which may be useful when generating large profiles. It can be generated from the text format using the `llvm-profdata` tool.
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. Similarly to the binary encoding, it can be generated using the `llvm-profdata` tool.

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, `llvm/lib/ProfileData/SampleProfWriter.cpp`).

```
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 ... ]
```

The file may contain blank lines between sections and within a section. However, the spacing within a single line is fixed. Additional spaces will result in an error while reading the file.

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.

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

1. 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).

2. [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`.

3. Number of samples. This is an integer quantity representing the number of samples collected by the profiler at this source location.
4. [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.

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 setting the `LLVM_PROFILE_FILE` environment variable to specify an alternate file. 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
```

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 and use can also 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>]

Without any other arguments, `-fprofile-generate` behaves identically to `-fprofile-instr-generate`. When given a directory name, it generates the profile file `default.profraw` in the directory named `dirname`. If `dirname` does not exist, it will be created at runtime. The environment variable `LLVM_PROFILE_FILE` can be used to override the directory and filename for the profile file at runtime. 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.profraw`. This can be altered at runtime via the `LLVM_PROFILE_FILE` environment variable:

```
$ LLVM_PROFILE_FILE=/tmp/myprofile/code.profraw ./code
```

The above invocation will produce the profile file `/tmp/myprofile/code.profraw` instead of `yyy/zzz/default.profraw`. Notice that `LLVM_PROFILE_FILE` overrides the directory *and* the file name for the profile file.

-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`.

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.

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 `#pragma weak` ([bug 3679](#)). Due to the uses described in the bug, this is likely to be implemented at some point, at least partially.
- 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 does not support global register variables; this 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 some experimental support for extensions from Microsoft Visual C++; to enable it, use the `-fms-extensions` command-line option. This is the default for Windows targets. Note that the support is incomplete. Some constructs such as `__declspec` on classes are ignored with a warning, and others such as [Microsoft IDL annotations](#) are silently ignored.

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.

- clang allows setting `_MSC_VER` with `-fmsc-version=`. It defaults to 1700 which is the same as Visual C/C++ 2012. Any number is supported and can greatly affect what Windows SDK and `c++stdlib` headers clang can compile.
- clang does not support the Microsoft extension where anonymous record members can be declared using user defined typedefs.
- clang supports the Microsoft `#pragma pack` feature for controlling record layout. GCC also contains support for this feature, however where MSVC and GCC are incompatible clang follows the MSVC definition.
- clang supports the Microsoft `#pragma comment(lib, "foo.lib")` feature for automatically linking against the specified library. Currently this feature only works with the Visual C++ linker.
- clang supports the Microsoft `#pragma comment(linker, "/flag:foo")` feature for adding linker flags to COFF object files. The user is responsible for ensuring that the linker understands the flags.
- clang defaults to C++11 for Windows targets.

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

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 driver, 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 cause errors. If they 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
/C              Don't discard comments when preprocessing
/c             Compile only
/D <macro[=value]> Define macro
/EH<value>      Exception handling model
/EP           Disable linemarker output and preprocess to stdout
/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
/GA           Assume thread-local variables are defined in the
↳ executable
/GF-          Disable string pooling
/GR-          Disable emission of RTTI data
```

/GR	Enable emission of RTTI data
/Gs<value>	Set stack probe size
/Gw-	Don't put each data item in its own section
/Gw	Put each data item in its own section
/Gy-	Don't put each function in its own section
/Gy	Put each function in its own section
/help	Display available options
/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
/Ob0	Disable inlining
/Od	Disable optimization
/Oi-	Disable use of builtin functions
/Oi	Enable use of builtin functions
/Os	Optimize for size
/Ot	Optimize for speed
/Ox	Maximum optimization
/Oy-	Disable frame pointer omission
/Oy	Enable frame pointer omission
/O<n>	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
/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
/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
/Wall	Enable -Wall
/WX-	Do not treat warnings as errors
/WX	Treat warnings as errors

```

/w          Disable all warnings
/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
/Zi          Enable debug information
/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
-fdiagnostics-parseable-fixits
                Print fix-its in machine parseable form
-fms-compatibility-version=<value>
                Dot-separated value representing the Microsoft_
↪compiler version
                number to report in _MSC_VER (0 = don't define it_
↪(default))
-fmsc-version=<value> Microsoft compiler version number to report in _
↪MSC_VER (0 = don't
                define it (default))
-fno-sanitize-coverage=<value>
                Disable specified features of coverage_
↪instrumentation for Sanitizers
-fno-sanitize-recover=<value>
                Disable recovery for specified sanitizers
-fno-sanitize-trap=<value>
                Disable trapping for specified sanitizers
-fsanitize-blacklist=<value>
                Path to blacklist file for sanitizers
-fsanitize-coverage=<value>
                Specify the type of coverage instrumentation for_
↪Sanitizers
-fsanitize-recover=<value>
                Enable recovery for specified sanitizers
-fsanitize-trap=<value> Enable trapping for specified sanitizers
-fsanitize=<check> Turn on runtime checks for various forms of_
↪undefined or suspicious
                behavior. See user manual for available checks
-mllvm <value> Additional arguments to forward to LLVM's option_
↪processing
-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.

Clang Language Extensions

- *Introduction*
- *Feature Checking Macros*
- *Include File Checking Macros*
- *Builtin Macros*
- *Vectors and Extended Vectors*
- *Messages on deprecated and unavailable Attributes*
- *Attributes on Enumerators*
- *'User-Specified' System Frameworks*
- *Checks for Standard Language Features*
- *Checks for Type Trait Primitives*
- *Blocks*
- *Objective-C Features*
- *Initializer lists for complex numbers in C*
- *Builtin Functions*
- *Non-standard C++11 Attributes*
- *Target-Specific Extensions*
- *Extensions for Static Analysis*
- *Extensions for Dynamic Analysis*
- *Extensions for selectively disabling optimization*
- *Extensions for loop hint optimizations*

Objective-C Literals

Introduction

Three new features were introduced into clang at the same time: *NSNumber Literals* provide a syntax for creating `NSNumber` from scalar literal expressions; *Collection Literals* provide a short-hand for creating arrays and dictionaries; *Object Subscripting* provides a way to use subscripting with Objective-C objects. Users of Apple compiler releases

can use these features starting with the Apple LLVM Compiler 4.0. Users of open-source LLVM.org compiler releases can use these features starting with clang v3.1.

These language additions simplify common Objective-C programming patterns, make programs more concise, and improve the safety of container creation.

This document describes how the features are implemented in clang, and how to use them in your own programs.

NSNumber Literals

The framework class `NSNumber` is used to wrap scalar values inside objects: signed and unsigned integers (`char`, `short`, `int`, `long`, `long long`), floating point numbers (`float`, `double`), and boolean values (`BOOL`, `C++ bool`). Scalar values wrapped in objects are also known as *boxed* values.

In Objective-C, any character, numeric or boolean literal prefixed with the '@' character will evaluate to a pointer to an `NSNumber` object initialized with that value. C's type suffixes may be used to control the size of numeric literals.

Examples

The following program illustrates the rules for `NSNumber` literals:

```
void main(int argc, const char *argv[]) {
    // character literals.
    NSNumber *theLetterZ = @'Z';           // equivalent to [NSNumber numberWithInt:'Z']

    // integral literals.
    NSNumber *fortyTwo = @42;              // equivalent to [NSNumber numberWithInt:42]
    NSNumber *fortyTwoUnsigned = @42U;     // equivalent to [NSNumber
↳numberWithUnsignedInt:42U]
    NSNumber *fortyTwoLong = @42L;         // equivalent to [NSNumber numberWithInt:42L]
    NSNumber *fortyTwoLongLong = @42LL;    // equivalent to [NSNumber
↳numberWithLongLong:42LL]

    // floating point literals.
    NSNumber *piFloat = @3.141592654F;     // equivalent to [NSNumber numberWithFloat:3.
↳141592654F]
    NSNumber *piDouble = @3.1415926535;    // equivalent to [NSNumber numberWithDouble:3.
↳1415926535]

    // BOOL literals.
    NSNumber *yesNumber = @YES;             // equivalent to [NSNumber numberWithBool:YES]
    NSNumber *noNumber = @NO;              // equivalent to [NSNumber numberWithBool:NO]

#ifdef __cplusplus
    NSNumber *trueNumber = @true;           // equivalent to [NSNumber
↳numberWithBool:(BOOL>true]
    NSNumber *falseNumber = @false;        // equivalent to [NSNumber
↳numberWithBool:(BOOL>false]
#endif
}
```

Discussion

`NSNumber` literals only support literal scalar values after the '@'. Consequently, `@INT_MAX` works, but `@INT_MIN` does not, because they are defined like this:

```
#define INT_MAX    2147483647 /* max value for an int */
#define INT_MIN    (-2147483647-1) /* min value for an int */
```

The definition of `INT_MIN` is not a simple literal, but a parenthesized expression. Parenthesized expressions are supported using the *boxed expression* syntax, which is described in the next section.

Because `NSNumber` does not currently support wrapping long double values, the use of a long double `NSNumber` literal (e.g. `@123.23L`) will be rejected by the compiler.

Previously, the `BOOL` type was simply a typedef for signed `char`, and `YES` and `NO` were macros that expand to `(BOOL) 1` and `(BOOL) 0` respectively. To support `@YES` and `@NO` expressions, these macros are now defined using new language keywords in `<objc/objc.h>`:

```
#if __has_feature(objc_bool)
#define YES      __objc_yes
#define NO       __objc_no
#else
#define YES      ((BOOL)1)
#define NO       ((BOOL)0)
#endif
```

The compiler implicitly converts `__objc_yes` and `__objc_no` to `(BOOL) 1` and `(BOOL) 0`. The keywords are used to disambiguate `BOOL` and integer literals.

Objective-C++ also supports `@true` and `@false` expressions, which are equivalent to `@YES` and `@NO`.

Boxed Expressions

Objective-C provides a new syntax for boxing C expressions:

```
@( <expression> )
```

Expressions of scalar (numeric, enumerated, `BOOL`), C string pointer and some C structures (via `NSValue`) are supported:

```
// numbers.
NSNumber *smallestInt = @(-INT_MAX - 1); // [NSNumber numberWithInt:(-INT_MAX - 1)]
NSNumber *piOverTwo = @(M_PI / 2);      // [NSNumber numberWithDouble:(M_PI / 2)]

// enumerated types.
typedef enum { Red, Green, Blue } Color;
NSNumber *favoriteColor = @(Green);      // [NSNumber numberWithInt:(int)Green)]

// strings.
NSString *path = @(getenv("PATH"));      // [NSString stringWithUTF8String:(getenv(
↳ "PATH"))]
NSArray *pathComponents = [path componentsSeparatedByString:@" "];

// structs.
NSValue *center = @(view.center);        // Point p = view.center;
                                          // [NSValue valueWithBytes:&p_
↳ objcType:@encode(Point)];
NSValue *frame = @(view.frame);          // Rect r = view.frame;
                                          // [NSValue valueWithBytes:&r_
↳ objcType:@encode(Rect)];
```

Boxed Enums

Cocoa frameworks frequently define constant values using *enums*. Although enum values are integral, they may not be used directly as boxed literals (this avoids conflicts with future '@'-prefixed Objective-C keywords). Instead, an enum value must be placed inside a boxed expression. The following example demonstrates configuring an AVAudioRecorder using a dictionary that contains a boxed enumeration value:

```
enum {
    AVAudioQualityMin = 0,
    AVAudioQualityLow = 0x20,
    AVAudioQualityMedium = 0x40,
    AVAudioQualityHigh = 0x60,
    AVAudioQualityMax = 0x7F
};

- (AVAudioRecorder *)recordToFile:(NSURL *)fileURL {
    NSDictionary *settings = @{ AVEncoderAudioQualityKey : @(AVAudioQualityMax) };
    return [[AVAudioRecorder alloc] initWithURL:fileURL settings:settings error:NULL];
}
```

The expression @(AVAudioQualityMax) converts AVAudioQualityMax to an integer type, and boxes the value accordingly. If the enum has a *fixed underlying type* as in:

```
typedef enum : unsigned char { Red, Green, Blue } Color;
NSNumber *red = @(Red), *green = @(Green), *blue = @(Blue); // => [NSNumber
↳ numberWithUnsignedChar:]
```

then the fixed underlying type will be used to select the correct NSNumber creation method.

Boxing a value of enum type will result in a NSNumber pointer with a creation method according to the underlying type of the enum, which can be a *fixed underlying type* or a compiler-defined integer type capable of representing the values of all the members of the enumeration:

```
typedef enum : unsigned char { Red, Green, Blue } Color;
Color col = Red;
NSNumber *nsCol = @(col); // => [NSNumber numberWithUnsignedChar:]
```

Boxed C Strings

A C string literal prefixed by the '@' token denotes an NSString literal in the same way a numeric literal prefixed by the '@' token denotes an NSNumber literal. When the type of the parenthesized expression is (char *) or (const char *), the result of the boxed expression is a pointer to an NSString object containing equivalent character data, which is assumed to be '\0'-terminated and UTF-8 encoded. The following example converts C-style command line arguments into NSString objects.

```
// Partition command line arguments into positional and option arguments.
NSMutableArray *args = [NSMutableArray new];
NSMutableDictionary *options = [NSMutableDictionary new];
while (--argc) {
    const char *arg = **++argv;
    if (strcmp(arg, "--", 2) == 0) {
        options[@(arg + 2)] = @(**++argv); // --key value
    } else {
        [args addObject:@(arg)]; // positional argument
    }
}
```


As with all C pointers, character pointer expressions can involve arbitrary pointer arithmetic, therefore programmers must ensure that the character data is valid. Passing `NULL` as the character pointer will raise an exception at runtime. When possible, the compiler will reject `NULL` character pointers used in boxed expressions.

Boxed C Structures

Boxed expressions support construction of `NSValue` objects. If said that C structures can be used, the only requirement is: structure should be marked with `objc_boxable` attribute. To support older version of frameworks and/or third-party libraries you may need to add the attribute via `typedef`.

```

struct __attribute__((objc_boxable)) Point {
    // ...
};

typedef struct __attribute__((objc_boxable)) _Size {
    // ...
} Size;

typedef struct _Rect {
    // ...
} Rect;

struct Point p;
NSValue *point = @(p);           // ok
Size s;
NSValue *size = @(s);           // ok

Rect r;
NSValue *bad_rect = @(r);       // error

typedef struct __attribute__((objc_boxable)) _Rect Rect;

NSValue *good_rect = @(r);      // ok

```

Container Literals

Objective-C now supports a new expression syntax for creating immutable array and dictionary container objects.

Examples

Immutable array expression:

```

NSArray *array = @[ @"Hello", NSApp, [NSNumber numberWithInt:42] ];

```

This creates an `NSArray` with 3 elements. The comma-separated sub-expressions of an array literal can be any Objective-C object pointer typed expression.

Immutable dictionary expression:

```

NSDictionary *dictionary = @{
    @"name" : NSUserName(),
    @"date" : [NSDate date],
    @"processInfo" : [NSProcessInfo processInfo]
};

```

This creates an `NSDictionary` with 3 key/value pairs. Value sub-expressions of a dictionary literal must be Objective-C object pointer typed, as in array literals. Key sub-expressions must be of an Objective-C object pointer type that implements the `<NSCopying>` protocol.

Discussion

Neither keys nor values can have the value `nil` in containers. If the compiler can prove that a key or value is `nil` at compile time, then a warning will be emitted. Otherwise, a runtime error will occur.

Using array and dictionary literals is safer than the variadic creation forms commonly in use today. Array literal expressions expand to calls to `+[NSArray arrayWithObjects:count:]`, which validates that all objects are non-`nil`. The variadic form, `+[NSArray arrayWithObjects:]` uses `nil` as an argument list terminator, which can lead to malformed array objects. Dictionary literals are similarly created with `+[NSDictionary dictionaryWithObjects:forKeys:count:]` which validates all objects and keys, unlike `+[NSDictionary dictionaryWithObjectsAndKeys:]` which also uses a `nil` parameter as an argument list terminator.

Object Subscripting

Objective-C object pointer values can now be used with C's subscripting operator.

Examples

The following code demonstrates the use of object subscripting syntax with `NSMutableArray` and `NSMutableDictionary` objects:

```
NSMutableArray *array = ...;
NSUInteger idx = ...;
id newObject = ...;
id oldObject = array[idx];
array[idx] = newObject;           // replace oldObject with newObject

NSMutableDictionary *dictionary = ...;
NSString *key = ...;
oldObject = dictionary[key];
dictionary[key] = newObject;     // replace oldObject with newObject
```

The next section explains how subscripting expressions map to accessor methods.

Subscripting Methods

Objective-C supports two kinds of subscript expressions: *array-style* subscript expressions use integer typed subscripts; *dictionary-style* subscript expressions use Objective-C object pointer typed subscripts. Each type of subscript expression is mapped to a message send using a predefined selector. The advantage of this design is flexibility: class designers are free to introduce subscripting by declaring methods or by adopting protocols. Moreover, because the method names are selected by the type of the subscript, an object can be subscripted using both array and dictionary styles.

Array-Style Subscripting

When the subscript operand has an integral type, the expression is rewritten to use one of two different selectors, depending on whether the element is being read or written. When an expression reads an element using an integral index, as in the following example:

```
NSUInteger idx = ...;
id value = object[idx];
```

it is translated into a call to `objectAtIndexedSubscript:`

```
id value = [object objectAtIndexedSubscript:idx];
```

When an expression writes an element using an integral index:

```
object[idx] = newValue;
```

it is translated to a call to `setObjectAtIndexedSubscript:`

```
[object setObject:newValue atIndexedSubscript:idx];
```

These message sends are then type-checked and performed just like explicit message sends. The method used for `objectAtIndexedSubscript:` must be declared with an argument of integral type and a return value of some Objective-C object pointer type. The method used for `setObjectAtIndexedSubscript:` must be declared with its first argument having some Objective-C pointer type and its second argument having integral type.

The meaning of indexes is left up to the declaring class. The compiler will coerce the index to the appropriate argument type of the method it uses for type-checking. For an instance of `NSArray`, reading an element using an index outside the range `[0, array.count)` will raise an exception. For an instance of `NSMutableArray`, assigning to an element using an index within this range will replace that element, but assigning to an element using an index outside this range will raise an exception; no syntax is provided for inserting, appending, or removing elements for mutable arrays.

A class need not declare both methods in order to take advantage of this language feature. For example, the class `NSArray` declares only `objectAtIndexedSubscript:`, so that assignments to elements will fail to type-check; moreover, its subclass `NSMutableArray` declares `setObjectAtIndexedSubscript:`.

Dictionary-Style Subscripting

When the subscript operand has an Objective-C object pointer type, the expression is rewritten to use one of two different selectors, depending on whether the element is being read from or written to. When an expression reads an element using an Objective-C object pointer subscript operand, as in the following example:

```
id key = ...;
id value = object[key];
```

it is translated into a call to the `objectForKeyedSubscript:` method:

```
id value = [object objectForKeyedSubscript:key];
```

When an expression writes an element using an Objective-C object pointer subscript:

```
object[key] = newValue;
```

it is translated to a call to `setObjectForKeyedSubscript:`

```
[object setObject:newValue forKeyedSubscript:key];
```

The behavior of `setObject:forKeyedSubscript:` is class-specific; but in general it should replace an existing value if one is already associated with a key, otherwise it should add a new value for the key. No syntax is provided for removing elements from mutable dictionaries.

Discussion

An Objective-C subscript expression occurs when the base operand of the C subscript operator has an Objective-C object pointer type. Since this potentially collides with pointer arithmetic on the value, these expressions are only supported under the modern Objective-C runtime, which categorically forbids such arithmetic.

Currently, only subscripts of integral or Objective-C object pointer type are supported. In C++, a class type can be used if it has a single conversion function to an integral or Objective-C pointer type, in which case that conversion is applied and analysis continues as appropriate. Otherwise, the expression is ill-formed.

An Objective-C object subscript expression is always an l-value. If the expression appears on the left-hand side of a simple assignment operator (`=`), the element is written as described below. If the expression appears on the left-hand side of a compound assignment operator (e.g. `+=`), the program is ill-formed, because the result of reading an element is always an Objective-C object pointer and no binary operators are legal on such pointers. If the expression appears in any other position, the element is read as described below. It is an error to take the address of a subscript expression, or (in C++) to bind a reference to it.

Programs can use object subscripting with Objective-C object pointers of type `id`. Normal dynamic message send rules apply; the compiler must see *some* declaration of the subscripting methods, and will pick the declaration seen first.

Caveats

Objects created using the literal or boxed expression syntax are not guaranteed to be uniqued by the runtime, but nor are they guaranteed to be newly-allocated. As such, the result of performing direct comparisons against the location of an object literal (using `==`, `!=`, `<`, `<=`, `>`, or `>=`) is not well-defined. This is usually a simple mistake in code that intended to call the `isEqual:` method (or the `compare:` method).

This caveat applies to compile-time string literals as well. Historically, string literals (using the `@"..."` syntax) have been uniqued across translation units during linking. This is an implementation detail of the compiler and should not be relied upon. If you are using such code, please use global string constants instead (`NSString * const MyConst = @"..."`) or use `isEqual:`.

Grammar Additions

To support the new syntax described above, the Objective-C @-expression grammar has the following new productions:

```
objc-at-expression : '@' (string-literal | encode-literal | selector-literal |  
↳protocol-literal | object-literal)  
                    ;  
  
object-literal : ('+' | '-')? numeric-constant  
                | character-constant  
                | boolean-constant  
                | array-literal  
                | dictionary-literal  
                ;
```

```

boolean-constant : '__objc_yes' | '__objc_no' | 'true' | 'false' /* boolean keywords.
↳ */
                ;

array-literal : '[' assignment-expression-list ']'
              ;

assignment-expression-list : assignment-expression (',' assignment-expression-list)?
                           | /* empty */
                           ;

dictionary-literal : '{' key-value-list '}'
                   ;

key-value-list : key-value-pair (',' key-value-list)?
               | /* empty */
               ;

key-value-pair : assignment-expression ':' assignment-expression
               ;

```

Note: @true and @false are only supported in Objective-C++.

Availability Checks

Programs test for the new features by using clang's `__has_feature` checks. Here are examples of their use:

```

#if __has_feature(objc_array_literals)
    // new way.
    NSArray *elements = @[ @"H", @"He", @"O", @"C" ];
#else
    // old way (equivalent).
    id objects[] = { @"H", @"He", @"O", @"C" };
    NSArray *elements = [NSArray arrayWithObjects:objects count:4];
#endif

#if __has_feature(objc_dictionary_literals)
    // new way.
    NSDictionary *masses = @{ @"H" : @1.0078, @"He" : @4.0026, @"O" : @15.9990, @"C"
↳ : @12.0096 };
#else
    // old way (equivalent).
    id keys[] = { @"H", @"He", @"O", @"C" };
    id values[] = { [NSNumber numberWithDouble:1.0078], [NSNumber numberWithDouble:4.
↳ 0026],
                    [NSNumber numberWithDouble:15.9990], [NSNumber
↳ numberWithDouble:12.0096] };
    NSDictionary *masses = [NSDictionary dictionaryWithObjects:objects forKey:keys
↳ count:4];
#endif

#if __has_feature(objc_subscripting)
    NSUInteger i, count = elements.count;
    for (i = 0; i < count; ++i) {
        NSString *element = elements[i];
        NSNumber *mass = masses[element];
        NSLog(@"the mass of %@ is %@", element, mass);
    }

```

```
    }
#else
    NSUInteger i, count = [elements count];
    for (i = 0; i < count; ++i) {
        NSString *element = [elements objectAtIndex:i];
        NSNumber *mass = [masses objectForKey:element];
        NSLog(@"the mass of %@ is %@", element, mass);
    }
#endif

#if __has_attribute(objc_boxable)
    typedef struct __attribute__((objc_boxable)) _Rect Rect;
#endif

#if __has_feature(objc_boxed_nsvalue_expressions)
    CABasicAnimation animation = [CABasicAnimation animationWithKeyPath:@"position"];
    animation.fromValue = @(layer.position);
    animation.toValue = @(newPosition);
    [layer addAnimation:animation forKey:@"move"];
#else
    CABasicAnimation animation = [CABasicAnimation animationWithKeyPath:@"position"];
    animation.fromValue = [NSValue valueWithCGPoint:layer.position];
    animation.toValue = [NSValue valueWithCGPoint:newPosition];
    [layer addAnimation:animation forKey:@"move"];
#endif
```

Code can use also `__has_feature(objc_bool)` to check for the availability of numeric literals support. This checks for the new `__objc_yes` / `__objc_no` keywords, which enable the use of `@YES` / `@NO` literals.

To check whether boxed expressions are supported, use `__has_feature(objc_boxed_expressions)` feature macro.

Language Specification for Blocks

- *Revisions*
- *Overview*
- *The Block Type*
- *Block Variable Declarations*
- *Block Literal Expressions*
- *The Invoke Operator*
- *The Copy and Release Operations*
- *The `__block` Storage Qualifier*
- *Control Flow*
- *Objective-C Extensions*
- *C++ Extensions*

Revisions

- 2008/2/25 — created
- 2008/7/28 — revised, `__block` syntax
- 2008/8/13 — revised, Block globals
- 2008/8/21 — revised, C++ elaboration
- 2008/11/1 — revised, `__weak` support
- 2009/1/12 — revised, explicit return types
- 2009/2/10 — revised, `__block` objects need retain

Overview

A new derived type is introduced to C and, by extension, Objective-C, C++, and Objective-C++

The Block Type

Like function types, the Block type is a pair consisting of a result value type and a list of parameter types very similar to a function type. Blocks are intended to be used much like functions with the key distinction being that in addition to executable code they also contain various variable bindings to automatic (stack) or managed (heap) memory.

The abstract declarator,

```
int (^)(char, float)
```

describes a reference to a Block that, when invoked, takes two parameters, the first of type `char` and the second of type `float`, and returns a value of type `int`. The Block referenced is of opaque data that may reside in automatic (stack) memory, global memory, or heap memory.

Block Variable Declarations

A variable with Block type is declared using function pointer style notation substituting `^` for `*`. The following are valid Block variable declarations:

```
void (^blockReturningVoidWithVoidArgument)(void);
int (^blockReturningIntWithIntAndCharArguments)(int, char);
void (^arrayOfTenBlocksReturningVoidWithIntArgument[10])(int);
```

Variadic . . . arguments are supported. [variadic.c] A Block that takes no arguments must specify `void` in the argument list [voidarg.c]. An empty parameter list does not represent, as K&R provide, an unspecified argument list. Note: both gcc and clang support K&R style as a convenience.

A Block reference may be cast to a pointer of arbitrary type and vice versa. [cast.c] A Block reference may not be dereferenced via the pointer dereference operator `*`, and thus a Block's size may not be computed at compile time. [sizeof.c]

Block Literal Expressions

A Block literal expression produces a reference to a Block. It is introduced by the use of the `^` token as a unary operator.

```
Block_literal_expression ::= ^ block_decl compound_statement_body
block_decl ::=
block_decl ::= parameter_list
block_decl ::= type_expression
```

where type expression is extended to allow `^` as a Block reference (pointer) where `*` is allowed as a function reference (pointer).

The following Block literal:

```
^ void (void) { printf("hello world\n"); }
```

produces a reference to a Block with no arguments with no return value.

The return type is optional and is inferred from the return statements. If the return statements return a value, they all must return a value of the same type. If there is no value returned the inferred type of the Block is void; otherwise it is the type of the return statement value.

If the return type is omitted and the argument list is `(void)`, the `(void)` argument list may also be omitted.

So:

```
^ ( void ) { printf("hello world\n"); }
```

and:

```
^ { printf("hello world\n"); }
```

are exactly equivalent constructs for the same expression.

The `type_expression` extends C expression parsing to accommodate Block reference declarations as it accommodates function pointer declarations.

Given:

```
typedef int (*pointerToFunctionThatReturnsIntWithCharArg) (char);
pointerToFunctionThatReturnsIntWithCharArg functionPointer;
^ pointerToFunctionThatReturnsIntWithCharArg (float x) { return functionPointer; }
```

and:

```
^ int ((*)(float x)) (char) { return functionPointer; }
```

are equivalent expressions, as is:

```
^(float x) { return functionPointer; }
```

[returnfunctionptr.c]

The compound statement body establishes a new lexical scope within that of its parent. Variables used within the scope of the compound statement are bound to the Block in the normal manner with the exception of those in automatic (stack) storage. Thus one may access functions and global variables as one would expect, as well as static local variables. [testme]

Local automatic (stack) variables referenced within the compound statement of a Block are imported and captured by the Block as const copies. The capture (binding) is performed at the time of the Block literal expression evaluation.

The compiler is not required to capture a variable if it can prove that no references to the variable will actually be evaluated. Programmers can force a variable to be captured by referencing it in a statement at the beginning of the Block, like so:


```
(void) foo;
```

This matters when capturing the variable has side-effects, as it can in Objective-C or C++.

The lifetime of variables declared in a Block is that of a function; each activation frame contains a new copy of variables declared within the local scope of the Block. Such variable declarations should be allowed anywhere [testme] rather than only when C99 parsing is requested, including for statements. [testme]

Block literal expressions may occur within Block literal expressions (nest) and all variables captured by any nested blocks are implicitly also captured in the scopes of their enclosing Blocks.

A Block literal expression may be used as the initialization value for Block variables at global or local static scope.

The Invoke Operator

Blocks are invoked using function call syntax with a list of expression parameters of types corresponding to the declaration and returning a result type also according to the declaration. Given:

```
int (^x)(char);
void (^z)(void);
int (^(*y))(char) = &x;
```

the following are all legal Block invocations:

```
x('a');
(*y)('a');
(true ? x : *y)('a')
```

The Copy and Release Operations

The compiler and runtime provide copy and release operations for Block references that create and, in matched use, release allocated storage for referenced Blocks.

The copy operation `Block_copy()` is styled as a function that takes an arbitrary Block reference and returns a Block reference of the same type. The release operation, `Block_release()`, is styled as a function that takes an arbitrary Block reference and, if dynamically matched to a Block copy operation, allows recovery of the referenced allocated memory.

The `__block` Storage Qualifier

In addition to the new Block type we also introduce a new storage qualifier, `__block`, for local variables. [testme: a `__block` declaration within a block literal] The `__block` storage qualifier is mutually exclusive to the existing local storage qualifiers `auto`, `register`, and `static`. [testme] Variables qualified by `__block` act as if they were in allocated storage and this storage is automatically recovered after last use of said variable. An implementation may choose an optimization where the storage is initially automatic and only “moved” to allocated (heap) storage upon a `Block_copy` of a referencing Block. Such variables may be mutated as normal variables are.

In the case where a `__block` variable is a Block one must assume that the `__block` variable resides in allocated storage and as such is assumed to reference a Block that is also in allocated storage (that it is the result of a `Block_copy` operation). Despite this there is no provision to do a `Block_copy` or a `Block_release` if an implementation provides initial automatic storage for Blocks. This is due to the inherent race condition of potentially several threads trying to update the shared variable and the need for synchronization around disposing of older values and copying new ones. Such synchronization is beyond the scope of this language specification.

Control Flow

The compound statement of a Block is treated much like a function body with respect to control flow in that `goto`, `break`, and `continue` do not escape the Block. Exceptions are treated *normally* in that when thrown they pop stack frames until a catch clause is found.

Objective-C Extensions

Objective-C extends the definition of a Block reference type to be that also of `id`. A variable or expression of Block type may be messaged or used as a parameter wherever an `id` may be. The converse is also true. Block references may thus appear as properties and are subject to the `assign`, `retain`, and `copy` attribute logic that is reserved for objects.

All Blocks are constructed to be Objective-C objects regardless of whether the Objective-C runtime is operational in the program or not. Blocks using automatic (stack) memory are objects and may be messaged, although they may not be assigned into `__weak` locations if garbage collection is enabled.

Within a Block literal expression within a method definition references to instance variables are also imported into the lexical scope of the compound statement. These variables are implicitly qualified as references from `self`, and so `self` is imported as a `const` copy. The net effect is that instance variables can be mutated.

The `Block_copy` operator retains all objects held in variables of automatic storage referenced within the Block expression (or form strong references if running under garbage collection). Object variables of `__block` storage type are assumed to hold normal pointers with no provision for `retain` and `release` messages.

Foundation defines (and supplies) `-copy` and `-release` methods for Blocks.

In the Objective-C and Objective-C++ languages, we allow the `__weak` specifier for `__block` variables of object type. If garbage collection is not enabled, this qualifier causes these variables to be kept without `retain` messages being sent. This knowingly leads to dangling pointers if the Block (or a copy) outlives the lifetime of this object.

In garbage collected environments, the `__weak` variable is set to `nil` when the object it references is collected, as long as the `__block` variable resides in the heap (either by default or via `Block_copy()`). The initial Apple implementation does in fact start `__block` variables on the stack and migrate them to the heap only as a result of a `Block_copy()` operation.

It is a runtime error to attempt to assign a reference to a stack-based Block into any storage marked `__weak`, including `__weak __block` variables.

C++ Extensions

Block literal expressions within functions are extended to allow `const` use of C++ objects, pointers, or references held in automatic storage.

As usual, within the block, references to captured variables become `const`-qualified, as if they were references to members of a `const` object. Note that this does not change the type of a variable of reference type.

For example, given a class `Foo`:

```
Foo foo;
Foo &fooRef = foo;
Foo *fooPtr = &foo;
```

A Block that referenced these variables would import the variables as `const` variations:

```
const Foo block_foo = foo;
Foo &block_fooRef = fooRef;
Foo *const block_fooPtr = fooPtr;
```

Captured variables are copied into the Block at the instant of evaluating the Block literal expression. They are also copied when calling `Block_copy()` on a Block allocated on the stack. In both cases, they are copied as if the variable were `const`-qualified, and it's an error if there's no such constructor.

Captured variables in Blocks on the stack are destroyed when control leaves the compound statement that contains the Block literal expression. Captured variables in Blocks on the heap are destroyed when the reference count of the Block drops to zero.

Variables declared as residing in `__block` storage may be initially allocated in the heap or may first appear on the stack and be copied to the heap as a result of a `Block_copy()` operation. When copied from the stack, `__block` variables are copied using their normal qualification (i.e. without adding `const`). In C++11, `__block` variables are copied as x-values if that is possible, then as l-values if not; if both fail, it's an error. The destructor for any initial stack-based version is called at the variable's normal end of scope.

References to `this`, as well as references to non-static members of any enclosing class, are evaluated by capturing `this` just like a normal variable of C pointer type.

Member variables that are Blocks may not be overloaded by the types of their arguments.

Block Implementation Specification

- *History*
- *High Level*
- *Imported Variables*
 - *Imported const copy variables*
 - *Imported const copy of Block reference*
 - * *Importing `__attribute__((NSObject))` variables*
 - *Imported `__block` marked variables*
 - * *Layout of `__block` marked variables*
 - * *Access to `__block` variables from within its lexical scope*
 - * *Importing `__block` variables into Blocks*
 - * *Importing `__attribute__((NSObject)) __block` variables*
 - * *`__block` escapes*
 - * *Nesting*
- *Objective C Extensions to Blocks*
 - *Importing Objects*
 - *Blocks as Objects*
 - *`__weak __block` Support*
- *C++ Support*
- *Runtime Helper Functions*
- *Copyright*

History

- 2008/7/14 - created.
- 2008/8/21 - revised, C++.
- 2008/9/24 - add `NULL isa` field to `__block` storage.
- 2008/10/1 - revise block layout to use a `static` descriptor structure.
- 2008/10/6 - revise block layout to use an unsigned long int flags.
- 2008/10/28 - specify use of `_Block_object_assign` and `_Block_object_dispose` for all “Object” types in helper functions.
- 2008/10/30 - revise new layout to have invoke function in same place.
- 2008/10/30 - add `__weak` support.
- 2010/3/16 - rev for `stret` return, signature field.
- 2010/4/6 - improved wording.
- 2013/1/6 - improved wording and converted to `rst`.

This document describes the Apple ABI implementation specification of Blocks.

The first shipping version of this ABI is found in Mac OS X 10.6, and shall be referred to as 10.6.ABI. As of 2010/3/16, the following describes the ABI contract with the runtime and the compiler, and, as necessary, will be referred to as ABI.2010.3.16.

Since the Apple ABI references symbols from other elements of the system, any attempt to use this ABI on systems prior to SnowLeopard is undefined.

High Level

The ABI of Blocks consist of their layout and the runtime functions required by the compiler. A Block consists of a structure of the following form:

```
struct Block_literal_1 {
    void *isa; // initialized to &_NSConcreteStackBlock or &_NSConcreteGlobalBlock
    int flags;
    int reserved;
    void (*invoke)(void *, ...);
    struct Block_descriptor_1 {
        unsigned long int reserved; // NULL
        unsigned long int size; // sizeof(struct Block_literal_1)
        // optional helper functions
        void (*copy_helper)(void *dst, void *src); // IFF (1<<25)
        void (*dispose_helper)(void *src); // IFF (1<<25)
        // required ABI.2010.3.16
        const char *signature; // IFF (1<<30)
    } *descriptor;
    // imported variables
};
```

The following flags bits are in use thusly for a possible ABI.2010.3.16:

```
enum {
    BLOCK_HAS_COPY_DISPOSE = (1 << 25),
    BLOCK_HAS_CTOR = (1 << 26), // helpers have C++ code
```

```

BLOCK_IS_GLOBAL =      (1 << 28),
BLOCK_HAS_STRET =      (1 << 29), // IFF BLOCK_HAS_SIGNATURE
BLOCK_HAS_SIGNATURE =   (1 << 30),
};

```

In 10.6.ABI the (1<<29) was usually set and was always ignored by the runtime - it had been a transitional marker that did not get deleted after the transition. This bit is now paired with (1<<30), and represented as the pair (3<<30), for the following combinations of valid bit settings, and their meanings:

```

switch (flags & (3<<29)) {
case (0<<29):      10.6.ABI, no signature field available
case (1<<29):      10.6.ABI, no signature field available
case (2<<29): ABI.2010.3.16, regular calling convention, presence of signature field
case (3<<29): ABI.2010.3.16, stret calling convention, presence of signature field,
}

```

The signature field is not always populated.

The following discussions are presented as 10.6.ABI otherwise.

Block literals may occur within functions where the structure is created in stack local memory. They may also appear as initialization expressions for Block variables of global or static local variables.

When a Block literal expression is evaluated the stack based structure is initialized as follows:

1. A static descriptor structure is declared and initialized as follows:
 - a. The `invoke` function pointer is set to a function that takes the Block structure as its first argument and the rest of the arguments (if any) to the Block and executes the Block compound statement.
 - b. The `size` field is set to the size of the following Block literal structure.
 - c. The `copy_helper` and `dispose_helper` function pointers are set to respective helper functions if they are required by the Block literal.
2. A stack (or global) Block literal data structure is created and initialized as follows:
 - a. The `isa` field is set to the address of the external `_NSConcreteStackBlock`, which is a block of uninitialized memory supplied in `libSystem`, or `_NSConcreteGlobalBlock` if this is a static or file level Block literal.
 - b. The `flags` field is set to zero unless there are variables imported into the Block that need helper functions for program level `Block_copy()` and `Block_release()` operations, in which case the (1<<25) flags bit is set.

As an example, the Block literal expression:

```
^ { printf("hello world\n"); }
```

would cause the following to be created on a 32-bit system:

```

struct __block_literal_1 {
    void *isa;
    int flags;
    int reserved;
    void (*invoke)(struct __block_literal_1 *);
    struct __block_descriptor_1 *descriptor;
};

void __block_invoke_1(struct __block_literal_1 *_block) {
    printf("hello world\n");
}

```

```
}

static struct __block_descriptor_1 {
    unsigned long int reserved;
    unsigned long int Block_size;
} __block_descriptor_1 = { 0, sizeof(struct __block_literal_1), __block_invoke_1 };
```

and where the Block literal itself appears:

```
struct __block_literal_1 __block_literal = {
    &NSConcreteStackBlock,
    (1<<29), <uninitialized>,
    __block_invoke_1,
    &__block_descriptor_1
};
```

A Block imports other Block references, `const` copies of other variables, and variables marked `__block`. In Objective-C, variables may additionally be objects.

When a Block literal expression is used as the initial value of a global or `static` local variable, it is initialized as follows:

```
struct __block_literal_1 __block_literal_1 = {
    &NSConcreteGlobalBlock,
    (1<<28)|(1<<29), <uninitialized>,
    __block_invoke_1,
    &__block_descriptor_1
};
```

that is, a different address is provided as the first value and a particular $(1<<28)$ bit is set in the `flags` field, and otherwise it is the same as for stack based Block literals. This is an optimization that can be used for any Block literal that imports no `const` or `__block` storage variables.

Imported Variables

Variables of `auto` storage class are imported as `const` copies. Variables of `__block` storage class are imported as a pointer to an enclosing data structure. Global variables are simply referenced and not considered as imported.

Imported `const` copy variables

Automatic storage variables not marked with `__block` are imported as `const` copies.

The simplest example is that of importing a variable of type `int`:

```
int x = 10;
void (^vv)(void) = ^{ printf("x is %d\n", x); }
x = 11;
vv();
```

which would be compiled to:

```
struct __block_literal_2 {
    void *isa;
    int flags;
    int reserved;
```

```

    void (*invoke)(struct __block_literal_2 *);
    struct __block_descriptor_2 *descriptor;
    const int x;
};

void __block_invoke_2(struct __block_literal_2 *_block) {
    printf("x is %d\n", _block->x);
}

static struct __block_descriptor_2 {
    unsigned long int reserved;
    unsigned long int Block_size;
} __block_descriptor_2 = { 0, sizeof(struct __block_literal_2) };

```

and:

```

struct __block_literal_2 __block_literal_2 = {
    &__NSConcreteStackBlock,
    (1<<29), <uninitialized>,
    __block_invoke_2,
    &__block_descriptor_2,
    x
};

```

In summary, scalars, structures, unions, and function pointers are generally imported as `const` copies with no need for helper functions.

Imported `const` copy of Block reference

The first case where copy and dispose helper functions are required is for the case of when a `Block` itself is imported. In this case both a `copy_helper` function and a `dispose_helper` function are needed. The `copy_helper` function is passed both the existing stack based pointer and the pointer to the new heap version and should call back into the runtime to actually do the copy operation on the imported fields within the `Block`. The runtime functions are all described in [Runtime Helper Functions](#).

A quick example:

```

void (^existingBlock)(void) = ...;
void (^vv)(void) = ^{ existingBlock(); }
vv();

struct __block_literal_3 {
    ...; // existing block
};

struct __block_literal_4 {
    void *isa;
    int flags;
    int reserved;
    void (*invoke)(struct __block_literal_4 *);
    struct __block_literal_3 *const existingBlock;
};

void __block_invoke_4(struct __block_literal_2 *_block) {
    __block->existingBlock->invoke(__block->existingBlock);
}

```

```

void __block_copy_4(struct __block_literal_4 *dst, struct __block_literal_4 *src) {
    // _Block_copy_assign(&dst->existingBlock, src->existingBlock, 0);
    _Block_object_assign(&dst->existingBlock, src->existingBlock, BLOCK_FIELD_IS_
↳BLOCK);
}

void __block_dispose_4(struct __block_literal_4 *src) {
    // was _Block_destroy
    _Block_object_dispose(src->existingBlock, BLOCK_FIELD_IS_BLOCK);
}

static struct __block_descriptor_4 {
    unsigned long int reserved;
    unsigned long int Block_size;
    void (*copy_helper)(struct __block_literal_4 *dst, struct __block_literal_4 *src);
    void (*dispose_helper)(struct __block_literal_4 *);
} __block_descriptor_4 = {
    0,
    sizeof(struct __block_literal_4),
    __block_copy_4,
    __block_dispose_4,
};

```

and where said Block is used:

```

struct __block_literal_4 _block_literal = {
    &NSConcreteStackBlock,
    (1<<25)|(1<<29), <uninitialized>
    __block_invoke_4,
    & __block_descriptor_4
    existingBlock,
};

```

Importing `__attribute__((NSObject))` variables

GCC introduces `__attribute__((NSObject))` on structure pointers to mean “this is an object”. This is useful because many low level data structures are declared as opaque structure pointers, e.g. `CFStringRef`, `CFArrayRef`, etc. When used from C, however, these are still really objects and are the second case where that requires copy and dispose helper functions to be generated. The copy helper functions generated by the compiler should use the `_Block_object_assign` runtime helper function and in the dispose helper the `_Block_object_dispose` runtime helper function should be called.

For example, Block foo in the following:

```

struct Opaque *__attribute__((NSObject)) objectPointer = ...;
...
void (^foo)(void) = ^{ CFPrint(objectPointer); };

```

would have the following helper functions generated:

```

void __block_copy_foo(struct __block_literal_5 *dst, struct __block_literal_5 *src) {
    _Block_object_assign(&dst->objectPointer, src-> objectPointer, BLOCK_FIELD_IS_
↳OBJECT);
}

```



```
void __block_dispose_foo(struct __block_literal_5 *src) {
    _Block_object_dispose(src->objectPointer, BLOCK_FIELD_IS_OBJECT);
}
```

Imported `__block` marked variables

Layout of `__block` marked variables

The compiler must embed variables that are marked `__block` in a specialized structure of the form:

```
struct _block_byref_foo {
    void *isa;
    struct Block_byref *forwarding;
    int flags;    //refcount;
    int size;
    typeof(marked_variable) marked_variable;
};
```

Variables of certain types require helper functions for when `Block_copy()` and `Block_release()` are performed upon a referencing Block. At the “C” level only variables that are of type `Block` or ones that have `__attribute__((NSObject))` marked require helper functions. In Objective-C objects require helper functions and in C++ stack based objects require helper functions. Variables that require helper functions use the form:

```
struct _block_byref_foo {
    void *isa;
    struct _block_byref_foo *forwarding;
    int flags;    //refcount;
    int size;
    // helper functions called via Block_copy() and Block_release()
    void (*byref_keep)(void *dst, void *src);
    void (*byref_dispose)(void *);
    typeof(marked_variable) marked_variable;
};
```

The structure is initialized such that:

- a. The forwarding pointer is set to the beginning of its enclosing structure.
- b. The size field is initialized to the total size of the enclosing structure.
- c. The flags field is set to either 0 if no helper functions are needed or (1<<25) if they are.
4. The helper functions are initialized (if present).
5. The variable itself is set to its initial value.
6. The `isa` field is set to `NULL`.

Access to `__block` variables from within its lexical scope

In order to “move” the variable to the heap upon a `copy_helper` operation the compiler must rewrite access to such a variable to be indirect through the structures forwarding pointer. For example:

```
int __block i = 10;
i = 11;
```

would be rewritten to be:

```
struct _block_byref_i {
    void *isa;
    struct _block_byref_i *forwarding;
    int flags;    //refcount;
    int size;
    int captured_i;
} i = { NULL, &i, 0, sizeof(struct _block_byref_i), 10 };

i.forwarding->captured_i = 11;
```

In the case of a Block reference variable being marked `__block` the helper code generated must use the `_Block_object_assign` and `_Block_object_dispose` routines supplied by the runtime to make the copies. For example:

```
__block void (voidBlock) (void) = blockA;
voidBlock = blockB;
```

would translate into:

```
struct _block_byref_voidBlock {
    void *isa;
    struct _block_byref_voidBlock *forwarding;
    int flags;    //refcount;
    int size;
    void (*byref_keep) (struct _block_byref_voidBlock *dst, struct _block_byref_
↳voidBlock *src);
    void (*byref_dispose) (struct _block_byref_voidBlock *);
    void (^captured_voidBlock) (void);
};

void _block_byref_keep_helper(struct _block_byref_voidBlock *dst, struct _block_byref_
↳voidBlock *src) {
    // _Block_copy_assign(&dst->captured_voidBlock, src->captured_voidBlock, 0);
    _Block_object_assign(&dst->captured_voidBlock, src->captured_voidBlock, BLOCK_
↳FIELD_IS_BLOCK | BLOCK_BYREF_CALLER);
}

void _block_byref_dispose_helper(struct _block_byref_voidBlock *param) {
    // _Block_destroy(param->captured_voidBlock, 0);
    _Block_object_dispose(param->captured_voidBlock, BLOCK_FIELD_IS_BLOCK | BLOCK_
↳BYREF_CALLER) }
```

and:

```
struct _block_byref_voidBlock voidBlock = {( .forwarding=&voidBlock, .flags=(1<<25), .
↳size=sizeof(struct _block_byref_voidBlock *),
    .byref_keep=_block_byref_keep_helper, .byref_dispose=_block_byref_dispose_helper,
    .captured_voidBlock=blockA )};

voidBlock.forwarding->captured_voidBlock = blockB;
```

Importing `__block` variables into Blocks

A Block that uses a `__block` variable in its compound statement body must import the variable and emit `copy_helper` and `dispose_helper` helper functions that, in turn, call back into the runtime to actually copy or

release the byref data block using the functions `_Block_object_assign` and `_Block_object_dispose`.

For example:

```
int __block i = 2;
functioncall(^{ i = 10; });
```

would translate to:

```
struct __block_byref_i {
    void *isa; // set to NULL
    struct __block_byref_voidBlock *forwarding;
    int flags; //refcount;
    int size;
    void (*byref_keep)(struct __block_byref_i *dst, struct __block_byref_i *src);
    void (*byref_dispose)(struct __block_byref_i *);
    int captured_i;
};

struct __block_literal_5 {
    void *isa;
    int flags;
    int reserved;
    void (*invoke)(struct __block_literal_5 *);
    struct __block_descriptor_5 *descriptor;
    struct __block_byref_i *i_holder;
};

void __block_invoke_5(struct __block_literal_5 *block) {
    block->forwarding->captured_i = 10;
}

void __block_copy_5(struct __block_literal_5 *dst, struct __block_literal_5 *src) {
    //Block_byref_assign_copy(&dst->captured_i, src->captured_i);
    _Block_object_assign(&dst->captured_i, src->captured_i, BLOCK_FIELD_IS_BYREF |
↳BLOCK_BYREF_CALLER);
}

void __block_dispose_5(struct __block_literal_5 *src) {
    //Block_byref_release(src->captured_i);
    _Block_object_dispose(src->captured_i, BLOCK_FIELD_IS_BYREF | BLOCK_BYREF_
↳CALLER);
}

static struct __block_descriptor_5 {
    unsigned long int reserved;
    unsigned long int Block_size;
    void (*copy_helper)(struct __block_literal_5 *dst, struct __block_literal_5 *src);
    void (*dispose_helper)(struct __block_literal_5 *);
} __block_descriptor_5 = { 0, sizeof(struct __block_literal_5) __block_copy_5, __
↳block_dispose_5 };

and:

struct __block_byref_i i = {( .forwarding=&i, .flags=0, .size=sizeof(struct __block_
↳byref_i) )};
struct __block_literal_5 __block_literal = {
    &__NSConcreteStackBlock,
```

```
(1<<25)|(1<<29), <uninitialized>,
__block_invoke_5,
&__block_descriptor_5,
2,
};
```

Importing `__attribute__((NSObject)) __block` variables

A `__block` variable that is also marked `__attribute__((NSObject))` should have `byref_keep` and `byref_dispose` helper functions that use `_Block_object_assign` and `_Block_object_dispose`.

`__block` escapes

Because Blocks referencing `__block` variables may have `Block_copy()` performed upon them the underlying storage for the variables may move to the heap. In Objective-C Garbage Collection Only compilation environments the heap used is the garbage collected one and no further action is required. Otherwise the compiler must issue a call to potentially release any heap storage for `__block` variables at all escapes or terminations of their scope. The call should be:

```
_Block_object_dispose(&__block_byref_foo, BLOCK_FIELD_IS_BYREF);
```

Nesting

Blocks may contain Block literal expressions. Any variables used within inner blocks are imported into all enclosing Block scopes even if the variables are not used. This includes `const` imports as well as `__block` variables.

Objective C Extensions to Blocks

Importing Objects

Objects should be treated as `__attribute__((NSObject))` variables; all `copy_helper`, `dispose_helper`, `byref_keep`, and `byref_dispose` helper functions should use `_Block_object_assign` and `_Block_object_dispose`. There should be no code generated that uses `*-retain` or `*-release` methods.

Blocks as Objects

The compiler will treat Blocks as objects when synthesizing property setters and getters, will characterize them as objects when generating garbage collection strong and weak layout information in the same manner as objects, and will issue strong and weak write-barrier assignments in the same manner as objects.

`__weak __block` Support

Objective-C (and Objective-C++) support the `__weak` attribute on `__block` variables. Under normal circumstances the compiler uses the Objective-C runtime helper support functions `objc_assign_weak` and `objc_read_weak`. Both should continue to be used for all reads and writes of `__weak __block` variables:

```
objc_read_weak(&block->byref_i->forwarding->i)
```

The `__weak` variable is stored in a `_block_byref_foo` structure and the `Block` has copy and dispose helpers for this structure that call:

```
_Block_object_assign(&dest->_block_byref_i, src->_block_byref_i, BLOCK_FIELD_IS_WEAK_
↳| BLOCK_FIELD_IS_BYREF);
```

and:

```
_Block_object_dispose(src->_block_byref_i, BLOCK_FIELD_IS_WEAK | BLOCK_FIELD_IS_
↳BYREF);
```

In turn, the `block_byref` copy support helpers distinguish between whether the `__block` variable is a `Block` or not and should either call:

```
_Block_object_assign(&dest->_block_byref_i, src->_block_byref_i, BLOCK_FIELD_IS_WEAK_
↳| BLOCK_FIELD_IS_OBJECT | BLOCK_BYREF_CALLER);
```

for something declared as an object or:

```
_Block_object_assign(&dest->_block_byref_i, src->_block_byref_i, BLOCK_FIELD_IS_WEAK_
↳| BLOCK_FIELD_IS_BLOCK | BLOCK_BYREF_CALLER);
```

for something declared as a `Block`.

A full example follows:

```
__block __weak id obj = <initialization expression>;
functioncall(^{ [obj somemessage]; });
```

would translate to:

```
struct _block_byref_obj {
    void *isa; // uninitialized
    struct _block_byref_obj *forwarding;
    int flags; //refcount;
    int size;
    void (*byref_keep)(struct _block_byref_i *dst, struct _block_byref_i *src);
    void (*byref_dispose)(struct _block_byref_i *);
    id captured_obj;
};

void _block_byref_obj_keep(struct _block_byref_voidBlock *dst, struct _block_byref_
↳voidBlock *src) {
    //Block_copy_assign(&dst->captured_obj, src->captured_obj, 0);
    _Block_object_assign(&dst->captured_obj, src->captured_obj, BLOCK_FIELD_IS_OBJECT_
↳| BLOCK_FIELD_IS_WEAK | BLOCK_BYREF_CALLER);
}

void _block_byref_obj_dispose(struct _block_byref_voidBlock *param) {
    //Block_destroy(param->captured_obj, 0);
    _Block_object_dispose(param->captured_obj, BLOCK_FIELD_IS_OBJECT | BLOCK_FIELD_IS_
↳WEAK | BLOCK_BYREF_CALLER);
};
```

for the `block byref` part and:

```

struct __block_literal_5 {
    void *isa;
    int flags;
    int reserved;
    void (*invoke) (struct __block_literal_5 *);
    struct __block_descriptor_5 *descriptor;
    struct __block_byref_obj *byref_obj;
};

void __block_invoke_5(struct __block_literal_5 *_block) {
    [objc_read_weak(&_block->byref_obj->forwarding->captured_obj) somemessage];
}

void __block_copy_5(struct __block_literal_5 *dst, struct __block_literal_5 *src) {
    //__Block_byref_assign_copy(&dst->byref_obj, src->byref_obj);
    __Block_object_assign(&dst->byref_obj, src->byref_obj, BLOCK_FIELD_IS_BYREF |
↳BLOCK_FIELD_IS_WEAK);
}

void __block_dispose_5(struct __block_literal_5 *src) {
    //__Block_byref_release(src->byref_obj);
    __Block_object_dispose(src->byref_obj, BLOCK_FIELD_IS_BYREF | BLOCK_FIELD_IS_
↳WEAK);
}

static struct __block_descriptor_5 {
    unsigned long int reserved;
    unsigned long int Block_size;
    void (*copy_helper) (struct __block_literal_5 *dst, struct __block_literal_5 *src);
    void (*dispose_helper) (struct __block_literal_5 *);
} __block_descriptor_5 = { 0, sizeof(struct __block_literal_5), __block_copy_5, __
↳block_dispose_5 };

```

and within the compound statement:

```

truct __block_byref_obj obj = {( .forwarding=&obj, .flags=(1<<25), .size=sizeof(struct
↳__block_byref_obj),
    .byref_keep=__block_byref_obj_keep, .byref_dispose=__block_byref_obj_
↳dispose,
    .captured_obj = <initialization expression> });

truct __block_literal_5 __block_literal = {
    &__NSConcreteStackBlock,
    (1<<25)|(1<<29), <uninitialized>,
    __block_invoke_5,
    &__block_descriptor_5,
    &obj,          // a reference to the on-stack structure containing "captured_obj"
};

functioncall(__block_literal->invoke(&__block_literal));

```

C++ Support

Within a block stack based C++ objects are copied into `const` copies using the copy constructor. It is an error if a stack based C++ object is used within a block if it does not have a copy constructor. In addition both copy and

destroy helper routines must be synthesized for the block to support the `Block_copy()` operation, and the flags work marked with the (1<<26) bit in addition to the (1<<25) bit. The copy helper should call the constructor using appropriate offsets of the variable within the supplied stack based block source and heap based destination for all const constructed copies, and similarly should call the destructor in the destroy routine.

As an example, suppose a C++ class `FOO` existed with a copy constructor. Within a code block a stack version of a `FOO` object is declared and used within a `Block` literal expression:

```
{
    FOO foo;
    void (^block)(void) = ^{ printf("%d\n", foo.value()); };
}
```

The compiler would synthesize:

```
struct __block_literal_10 {
    void *isa;
    int flags;
    int reserved;
    void (*invoke)(struct __block_literal_10 *);
    struct __block_descriptor_10 *descriptor;
    const FOO foo;
};

void __block_invoke_10(struct __block_literal_10 *_block) {
    printf("%d\n", _block->foo.value());
}

void __block_literal_10(struct __block_literal_10 *dst, struct __block_literal_10_
↪ *src) {
    FOO_ctor(&dst->foo, &src->foo);
}

void __block_dispose_10(struct __block_literal_10 *src) {
    FOO_dtor(&src->foo);
}

static struct __block_descriptor_10 {
    unsigned long int reserved;
    unsigned long int Block_size;
    void (*copy_helper)(struct __block_literal_10 *dst, struct __block_literal_10_
↪ *src);
    void (*dispose_helper)(struct __block_literal_10 *);
} __block_descriptor_10 = { 0, sizeof(struct __block_literal_10), __block_copy_10, __
↪ block_dispose_10 };

and the code would be:
```

```
{
    FOO foo;
    comp_ctor(&foo); // default constructor
    struct __block_literal_10 _block_literal = {
        &NSConcreteStackBlock,
        (1<<25)|(1<<26)|(1<<29), <uninitialized>,
        __block_invoke_10,
        &__block_descriptor_10,
    };
    comp_ctor(&_block_literal->foo, &foo); // const copy into stack version
    struct __block_literal_10 &block = &_block_literal; // assign literal to block_
↪ variable
```

```
block->invoke(block);    // invoke block
comp_dtor(&_block_literal->foo); // destroy stack version of const block copy
comp_dtor(&foo); // destroy original version
}
```

C++ objects stored in `__block` storage start out on the stack in a `block_byref` data structure as do other variables. Such objects (if not `const` objects) must support a regular copy constructor. The `block_byref` data structure will have copy and destroy helper routines synthesized by the compiler. The copy helper will have code created to perform the copy constructor based on the initial stack `block_byref` data structure, and will also set the (1<<26) bit in addition to the (1<<25) bit. The destroy helper will have code to do the destructor on the object stored within the supplied `block_byref` heap data structure. For example,

```
__block FOO blockStorageFoo;
```

requires the normal constructor for the embedded `blockStorageFoo` object:

```
FOO_ctor(& _block_byref_blockStorageFoo->blockStorageFoo);
```

and at scope termination the destructor:

```
FOO_dtor(& _block_byref_blockStorageFoo->blockStorageFoo);
```

Note that the forwarding indirection is *NOT* used.

The compiler would need to generate (if used from a block literal) the following copy/dispose helpers:

```
void _block_byref_obj_keep(struct _block_byref_blockStorageFoo *dst, struct _block_
↳byref_blockStorageFoo *src) {
    FOO_ctor(&dst->blockStorageFoo, &src->blockStorageFoo);
}

void _block_byref_obj_dispose(struct _block_byref_blockStorageFoo *src) {
    FOO_dtor(&src->blockStorageFoo);
}
```

for the appropriately named constructor and destructor for the class/struct `FOO`.

To support member variable and function access the compiler will synthesize a `const` pointer to a block version of the `this` pointer.

Runtime Helper Functions

The runtime helper functions are described in `/usr/local/include/Block_private.h`. To summarize their use, a Block requires copy/dispose helpers if it imports any block variables, `__block` storage variables, `__attribute__((NSObject))` variables, or C++ `const` copied objects with constructor/destructors. The (1<<26) bit is set and functions are generated.

The block copy helper function should, for each of the variables of the type mentioned above, call:

```
_Block_object_assign(&dst->target, src->target, BLOCK_FIELD_<appropo>);
```

in the copy helper and:

```
_Block_object_dispose(->target, BLOCK_FIELD_<appropo>);
```

in the dispose helper where `<appropo>` is:


```
enum {
    BLOCK_FIELD_IS_OBJECT    = 3,  // id, NSObject, __attribute__((NSObject)), block,
    ↪...
    BLOCK_FIELD_IS_BLOCK     = 7,  // a block variable
    BLOCK_FIELD_IS_BYREF     = 8,  // the on stack structure holding the __block_
    ↪variable

    BLOCK_FIELD_IS_WEAK      = 16,  // declared __weak

    BLOCK_BYREF_CALLER       = 128, // called from byref copy/dispose helpers
};
```

and of course the constructors/destructors for const copied C++ objects.

The `block_byref` data structure similarly requires copy/dispose helpers for block variables, `__attribute__((NSObject))` variables, or C++ const copied objects with constructor/destructors, and again the (1<<26) bit is set and functions are generated in the same manner.

Under ObjC we allow `__weak` as an attribute on `__block` variables, and this causes the addition of `BLOCK_FIELD_IS_WEAK` orred onto the `BLOCK_FIELD_IS_BYREF` flag when copying the `block_byref` structure in the Block copy helper, and onto the `BLOCK_FIELD_<appropo>` field within the `block_byref` copy/dispose helper calls.

The prototypes, and summary, of the helper functions are:

```
/* Certain field types require runtime assistance when being copied to the
   heap. The following function is used to copy fields of types: blocks,
   pointers to byref structures, and objects (including
   __attribute__((NSObject)) pointers. BLOCK_FIELD_IS_WEAK is orthogonal to
   the other choices which are mutually exclusive. Only in a Block copy
   helper will one see BLOCK_FIELD_IS_BYREF.
*/
void _Block_object_assign(void *destAddr, const void *object, const int flags);

/* Similarly a compiler generated dispose helper needs to call back for each
   field of the byref data structure. (Currently the implementation only
   packs one field into the byref structure but in principle there could be
   more). The same flags used in the copy helper should be used for each
   call generated to this function:
*/
void _Block_object_dispose(const void *object, const int flags);
```

Copyright

Copyright 2008-2010 Apple, Inc. Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION

OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Objective-C Automatic Reference Counting (ARC)

- *About this document*
 - *Purpose*
 - *Background*
 - *Evolution*
- *General*
- *Retainable object pointers*
 - *Retain count semantics*
 - *Retainable object pointers as operands and arguments*
 - * *Consumed parameters*
 - * *Retained return values*
 - * *Unretained return values*
 - * *Bridged casts*
 - *Restrictions*
 - * *Conversion of retainable object pointers*
 - * *Conversion to retainable object pointer type of expressions with known semantics*
 - * *Conversion from retainable object pointer type in certain contexts*
- *Ownership qualification*
 - *Spelling*
 - * *Property declarations*
 - *Semantics*
 - *Restrictions*
 - * *Weak-unavailable types*
 - * *Storage duration of `__autoreleasing` objects*
 - * *Conversion of pointers to ownership-qualified types*
 - * *Passing to an out parameter by writeback*
 - * *Ownership-qualified fields of structs and unions*
 - *Ownership inference*
 - * *Objects*
 - * *Indirect parameters*
 - * *Template arguments*
- *Method families*

- *Explicit method family control*
- *Semantics of method families*
 - * *Semantics of `init`*
 - * *Related result types*
- *Optimization*
 - *Object liveness*
 - *No object lifetime extension*
 - *Precise lifetime semantics*
- *Miscellaneous*
 - *Special methods*
 - * *Memory management methods*
 - * *`dealloc`*
 - *`@autoreleasepool`*
 - *`self`*
 - *Fast enumeration iteration variables*
 - *Blocks*
 - *Exceptions*
 - *Interior pointers*
 - *C retainable pointer types*
 - * *Auditing of C retainable pointer interfaces*
- *Runtime support*
 - `id objc_autorelease(id value);`
 - `void objc_autoreleasePoolPop(void *pool);`
 - `void *objc_autoreleasePoolPush(void);`
 - `id objc_autoreleaseReturnValue(id value);`
 - `void objc_copyWeak(id *dest, id *src);`
 - `void objc_destroyWeak(id *object);`
 - `id objc_initWeak(id *object, id value);`
 - `id objc_loadWeak(id *object);`
 - `id objc_loadWeakRetained(id *object);`
 - `void objc_moveWeak(id *dest, id *src);`
 - `void objc_release(id value);`
 - `id objc_retain(id value);`
 - `id objc_retainAutorelease(id value);`
 - `id objc_retainAutoreleaseReturnValue(id value);`

```
- id objc_retainAutoreleasedReturnValue(id value);  
- id objc_retainBlock(id value);  
- id objc_storeStrong(id *object, id value);  
- id objc_storeWeak(id *object, id value);
```

About this document

Purpose

The first and primary purpose of this document is to serve as a complete technical specification of Automatic Reference Counting. Given a core Objective-C compiler and runtime, it should be possible to write a compiler and runtime which implements these new semantics.

The secondary purpose is to act as a rationale for why ARC was designed in this way. This should remain tightly focused on the technical design and should not stray into marketing speculation.

Background

This document assumes a basic familiarity with C.

Blocks are a C language extension for creating anonymous functions. Users interact with and transfer block objects using block pointers, which are represented like a normal pointer. A block may capture values from local variables; when this occurs, memory must be dynamically allocated. The initial allocation is done on the stack, but the runtime provides a `Block_copy` function which, given a block pointer, either copies the underlying block object to the heap, setting its reference count to 1 and returning the new block pointer, or (if the block object is already on the heap) increases its reference count by 1. The paired function is `Block_release`, which decreases the reference count by 1 and destroys the object if the count reaches zero and is on the heap.

Objective-C is a set of language extensions, significant enough to be considered a different language. It is a strict superset of C. The extensions can also be imposed on C++, producing a language called Objective-C++. The primary feature is a single-inheritance object system; we briefly describe the modern dialect.

Objective-C defines a new type kind, collectively called the object pointer types. This kind has two notable builtin members, `id` and `Class`; `id` is the final supertype of all object pointers. The validity of conversions between object pointer types is not checked at runtime. Users may define classes; each class is a type, and the pointer to that type is an object pointer type. A class may have a superclass; its pointer type is a subtype of its superclass's pointer type. A class has a set of ivars, fields which appear on all instances of that class. For every class *T* there's an associated metaclass; it has no fields, its superclass is the metaclass of *T*'s superclass, and its metaclass is a global class. Every class has a global object whose class is the class's metaclass; metaclasses have no associated type, so pointers to this object have type `Class`.

A class declaration (`@interface`) declares a set of methods. A method has a return type, a list of argument types, and a selector: a name like `foo:bar:baz:`, where the number of colons corresponds to the number of formal arguments. A method may be an instance method, in which case it can be invoked on objects of the class, or a class method, in which case it can be invoked on objects of the metaclass. A method may be invoked by providing an object (called the receiver) and a list of formal arguments interspersed with the selector, like so:

```
[receiver foo: fooArg bar: barArg baz: bazArg]
```

This looks in the dynamic class of the receiver for a method with this name, then in that class's superclass, etc., until it finds something it can execute. The receiver "expression" may also be the name of a class, in which case the actual receiver is the class object for that class, or (within method definitions) it may be `super`, in which case the lookup

algorithm starts with the static superclass instead of the dynamic class. The actual methods dynamically found in a class are not those declared in the `@interface`, but those defined in a separate `@implementation` declaration; however, when compiling a call, typechecking is done based on the methods declared in the `@interface`.

Method declarations may also be grouped into protocols, which are not inherently associated with any class, but which classes may claim to follow. Object pointer types may be qualified with additional protocols that the object is known to support.

Class extensions are collections of ivars and methods, designed to allow a class's `@interface` to be split across multiple files; however, there is still a primary implementation file which must see the `@interfaces` of all class extensions. Categories allow methods (but not ivars) to be declared *post hoc* on an arbitrary class; the methods in the category's `@implementation` will be dynamically added to that class's method tables which the category is loaded at runtime, replacing those methods in case of a collision.

In the standard environment, objects are allocated on the heap, and their lifetime is manually managed using a reference count. This is done using two instance methods which all classes are expected to implement: `retain` increases the object's reference count by 1, whereas `release` decreases it by 1 and calls the instance method `dealloc` if the count reaches 0. To simplify certain operations, there is also an autorelease pool, a thread-local list of objects to call `release` on later; an object can be added to this pool by calling `autorelease` on it.

Block pointers may be converted to type `id`; block objects are laid out in a way that makes them compatible with Objective-C objects. There is a builtin class that all block objects are considered to be objects of; this class implements `retain` by adjusting the reference count, not by calling `Block_copy`.

Evolution

ARC is under continual evolution, and this document must be updated as the language progresses.

If a change increases the expressiveness of the language, for example by lifting a restriction or by adding new syntax, the change will be annotated with a revision marker, like so:

ARC applies to Objective-C pointer types, block pointer types, and [beginning Apple 8.0, LLVM 3.8]
BPTRs declared within `extern "BCPL"` blocks.

For now, it is sensible to version this document by the releases of its sole implementation (and its host project), clang. "LLVM X.Y" refers to an open-source release of clang from the LLVM project. "Apple X.Y" refers to an Apple-provided release of the Apple LLVM Compiler. Other organizations that prepare their own, separately-versioned clang releases and wish to maintain similar information in this document should send requests to cfe-dev.

If a change decreases the expressiveness of the language, for example by imposing a new restriction, this should be taken as an oversight in the original specification and something to be avoided in all versions. Such changes are generally to be avoided.

General

Automatic Reference Counting implements automatic memory management for Objective-C objects and blocks, freeing the programmer from the need to explicitly insert retains and releases. It does not provide a cycle collector; users must explicitly manage the lifetime of their objects, breaking cycles manually or with weak or unsafe references.

ARC may be explicitly enabled with the compiler flag `-fobjc-arc`. It may also be explicitly disabled with the compiler flag `-fno-objc-arc`. The last of these two flags appearing on the compile line "wins".

If ARC is enabled, `__has_feature(objc_arc)` will expand to 1 in the preprocessor. For more information about `__has_feature`, see the [language extensions](#) document.

Retainable object pointers

This section describes retainable object pointers, their basic operations, and the restrictions imposed on their use under ARC. Note in particular that it covers the rules for pointer *values* (patterns of bits indicating the location of a pointed-to object), not pointer *objects* (locations in memory which store pointer values). The rules for objects are covered in the next section.

A retainable object pointer (or “retainable pointer”) is a value of a retainable object pointer type (“retainable type”). There are three kinds of retainable object pointer types:

- block pointers (formed by applying the caret (^) declarator sigil to a function type)
- Objective-C object pointers (`id`, `Class`, `NSString*`, etc.)
- typedefs marked with `__attribute__((NSObject))`

Other pointer types, such as `int*` and `CFStringRef`, are not subject to ARC’s semantics and restrictions.

Rationale

We are not at liberty to require all code to be recompiled with ARC; therefore, ARC must interoperate with Objective-C code which manages retains and releases manually. In general, there are three requirements in order for a compiler-supported reference-count system to provide reliable interoperation:

- The type system must reliably identify which objects are to be managed. An `int*` might be a pointer to a `malloc`’ed array, or it might be an interior pointer to such an array, or it might point to some field or local variable. In contrast, values of the retainable object pointer types are never interior.
- The type system must reliably indicate how to manage objects of a type. This usually means that the type must imply a procedure for incrementing and decrementing retain counts. Supporting single-ownership objects requires a lot more explicit mediation in the language.
- There must be reliable conventions for whether and when “ownership” is passed between caller and callee, for both arguments and return values. Objective-C methods follow such a convention very reliably, at least for system libraries on Mac OS X, and functions always pass objects at +0. The C-based APIs for Core Foundation objects, on the other hand, have much more varied transfer semantics.

The use of `__attribute__((NSObject))` typedefs is not recommended. If it’s absolutely necessary to use this attribute, be very explicit about using the typedef, and do not assume that it will be preserved by language features like `__typeof` and C++ template argument substitution.

Rationale

Any compiler operation which incidentally strips type “sugar” from a type will yield a type without the attribute, which may result in unexpected behavior.

Retain count semantics

A retainable object pointer is either a null pointer or a pointer to a valid object. Furthermore, if it has block pointer type and is not `null` then it must actually be a pointer to a block object, and if it has `Class` type (possibly protocol-qualified) then it must actually be a pointer to a class object. Otherwise ARC does not enforce the Objective-C type system as long as the implementing methods follow the signature of the static type. It is undefined behavior if ARC is exposed to an invalid pointer.

For ARC’s purposes, a valid object is one with “well-behaved” retaining operations. Specifically, the object must be laid out such that the Objective-C message send machinery can successfully send it the following messages:

- `retain`, taking no arguments and returning a pointer to the object.
- `release`, taking no arguments and returning `void`.
- `autorelease`, taking no arguments and returning a pointer to the object.

The behavior of these methods is constrained in the following ways. The term high-level semantics is an intentionally vague term; the intent is that programmers must implement these methods in a way such that the compiler, modifying code in ways it deems safe according to these constraints, will not violate their requirements. For example, if the user puts logging statements in `retain`, they should not be surprised if those statements are executed more or less often depending on optimization settings. These constraints are not exhaustive of the optimization opportunities: values held in local variables are subject to additional restrictions, described later in this document.

It is undefined behavior if a computation history featuring a send of `retain` followed by a send of `release` to the same object, with no intervening `release` on that object, is not equivalent under the high-level semantics to a computation history in which these sends are removed. Note that this implies that these methods may not raise exceptions.

It is undefined behavior if a computation history features any use whatsoever of an object following the completion of a send of `release` that is not preceded by a send of `retain` to the same object.

The behavior of `autorelease` must be equivalent to sending `release` when one of the autorelease pools currently in scope is popped. It may not throw an exception.

When the semantics call for performing one of these operations on a retainable object pointer, if that pointer is `null` then the effect is a no-op.

All of the semantics described in this document are subject to additional *optimization rules* which permit the removal or optimization of operations based on local knowledge of data flow. The semantics describe the high-level behaviors that the compiler implements, not an exact sequence of operations that a program will be compiled into.

Retainable object pointers as operands and arguments

In general, ARC does not perform retain or release operations when simply using a retainable object pointer as an operand within an expression. This includes:

- loading a retainable pointer from an object with non-weak *ownership*,
- passing a retainable pointer as an argument to a function or method, and
- receiving a retainable pointer as the result of a function or method call.

Rationale

While this might seem uncontroversial, it is actually unsafe when multiple expressions are evaluated in “parallel”, as with binary operators and calls, because (for example) one expression might load from an object while another writes to it. However, C and C++ already call this undefined behavior because the evaluations are unsequenced, and ARC simply exploits that here to avoid needing to retain arguments across a large number of calls.

The remainder of this section describes exceptions to these rules, how those exceptions are detected, and what those exceptions imply semantically.

Consumed parameters

A function or method parameter of retainable object pointer type may be marked as consumed, signifying that the callee expects to take ownership of a +1 retain count. This is done by adding the `ns_consumed` attribute to the parameter declaration, like so:

```
void foo(__attribute__((ns_consumed)) id x);  
- (void) foo: (id) __attribute__((ns_consumed)) x;
```

This attribute is part of the type of the function or method, not the type of the parameter. It controls only how the argument is passed and received.

When passing such an argument, ARC retains the argument prior to making the call.

When receiving such an argument, ARC releases the argument at the end of the function, subject to the usual optimizations for local values.

Rationale

This formalizes direct transfers of ownership from a caller to a callee. The most common scenario here is passing the `self` parameter to `init`, but it is useful to generalize. Typically, local optimization will remove any extra retains and releases: on the caller side the retain will be merged with a +1 source, and on the callee side the release will be rolled into the initialization of the parameter.

The implicit `self` parameter of a method may be marked as consumed by adding `__attribute__((ns_consumes_self))` to the method declaration. Methods in the `init` family are treated as if they were implicitly marked with this attribute.

It is undefined behavior if an Objective-C message send to a method with `ns_consumed` parameters (other than `self`) is made with a null receiver. It is undefined behavior if the method to which an Objective-C message send statically resolves to has a different set of `ns_consumed` parameters than the method it dynamically resolves to. It is undefined behavior if a block or function call is made through a static type with a different set of `ns_consumed` parameters than the implementation of the called block or function.

Rationale

Consumed parameters with null receiver are a guaranteed leak. Mismatches with consumed parameters will cause over-retains or over-releases, depending on the direction. The rule about function calls is really just an application of the existing C/C++ rule about calling functions through an incompatible function type, but it's useful to state it explicitly.

Retained return values

A function or method which returns a retainable object pointer type may be marked as returning a retained value, signifying that the caller expects to take ownership of a +1 retain count. This is done by adding the `ns_returns_retained` attribute to the function or method declaration, like so:

```
id foo(void) __attribute__((ns_returns_retained));  
- (id) foo __attribute__((ns_returns_retained));
```

This attribute is part of the type of the function or method.

When returning from such a function or method, ARC retains the value at the point of evaluation of the return statement, before leaving all local scopes.

When receiving a return result from such a function or method, ARC releases the value at the end of the full-expression it is contained within, subject to the usual optimizations for local values.

Rationale

This formalizes direct transfers of ownership from a callee to a caller. The most common scenario this models is the retained return from `init`, `alloc`, `new`, and `copy` methods, but there are other cases in the frameworks. After optimization there are typically no extra retains and releases required.

Methods in the `alloc`, `copy`, `init`, `mutableCopy`, and `new` *families* are implicitly marked `__attribute__((ns_returns_retained))`. This may be suppressed by explicitly marking the method `__attribute__((ns_returns_not_retained))`.

It is undefined behavior if the method to which an Objective-C message send statically resolves has different retain semantics on its result from the method it dynamically resolves to. It is undefined behavior if a block or function call is made through a static type with different retain semantics on its result from the implementation of the called block or function.

Rationale

Mismatches with returned results will cause over-retains or over-releases, depending on the direction. Again, the rule about function calls is really just an application of the existing C/C++ rule about calling functions through an incompatible function type.

Unretained return values

A method or function which returns a retainable object type but does not return a retained value must ensure that the object is still valid across the return boundary.

When returning from such a function or method, ARC retains the value at the point of evaluation of the return statement, then leaves all local scopes, and then balances out the retain while ensuring that the value lives across the call boundary. In the worst case, this may involve an `autorelease`, but callers must not assume that the value is actually in the autorelease pool.

ARC performs no extra mandatory work on the caller side, although it may elect to do something to shorten the lifetime of the returned value.

Rationale

It is common in non-ARC code to not return an autoreleased value; therefore the convention does not force either path. It is convenient to not be required to do unnecessary retains and autoreleases; this permits optimizations such as eliding retain/autoreleases when it can be shown that the original pointer will still be valid at the point of return.

A method or function may be marked with `__attribute__((ns_returns_autoreleased))` to indicate that it returns a pointer which is guaranteed to be valid at least as long as the innermost autorelease pool. There are no additional semantics enforced in the definition of such a method; it merely enables optimizations in callers.

Bridged casts

A bridged cast is a C-style cast annotated with one of three keywords:

- `(__bridge T) op` casts the operand to the destination type `T`. If `T` is a retainable object pointer type, then `op` must have a non-retainable pointer type. If `T` is a non-retainable pointer type, then `op` must have a retainable object pointer type. Otherwise the cast is ill-formed. There is no transfer of ownership, and ARC inserts no retain operations.

- (`__bridge_retained T`) `op` casts the operand, which must have retainable object pointer type, to the destination type, which must be a non-retainable pointer type. ARC retains the value, subject to the usual optimizations on local values, and the recipient is responsible for balancing that `+1`.
- (`__bridge_transfer T`) `op` casts the operand, which must have non-retainable pointer type, to the destination type, which must be a retainable object pointer type. ARC will release the value at the end of the enclosing full-expression, subject to the usual optimizations on local values.

These casts are required in order to transfer objects in and out of ARC control; see the rationale in the section on *conversion of retainable object pointers*.

Using a `__bridge_retained` or `__bridge_transfer` cast purely to convince ARC to emit an unbalanced retain or release, respectively, is poor form.

Restrictions

Conversion of retainable object pointers

In general, a program which attempts to implicitly or explicitly convert a value of retainable object pointer type to any non-retainable type, or vice-versa, is ill-formed. For example, an Objective-C object pointer shall not be converted to `void*`. As an exception, cast to `intptr_t` is allowed because such casts are not transferring ownership. The *bridged casts* may be used to perform these conversions where necessary.

Rationale

We cannot ensure the correct management of the lifetime of objects if they may be freely passed around as unmanaged types. The bridged casts are provided so that the programmer may explicitly describe whether the cast transfers control into or out of ARC.

However, the following exceptions apply.

Conversion to retainable object pointer type of expressions with known semantics

[beginning Apple 4.0, LLVM 3.1] These exceptions have been greatly expanded; they previously applied only to a much-reduced subset which is difficult to categorize but which included null pointers, message sends (under the given rules), and the various global constants.

An unbridged conversion to a retainable object pointer type from a type other than a retainable object pointer type is ill-formed, as discussed above, unless the operand of the cast has a syntactic form which is known retained, known unretained, or known retain-agnostic.

An expression is known retain-agnostic if it is:

- an Objective-C string literal,
- a load from a `const` system global variable of *C retainable pointer type*, or
- a null pointer constant.

An expression is known unretained if it is an rvalue of *C retainable pointer type* and it is:

- a direct call to a function, and either that function has the `cf_returns_not_retained` attribute or it is an *audited* function that does not have the `cf_returns_retained` attribute and does not follow the create/copy naming convention,
- a message send, and the declared method either has the `cf_returns_not_retained` attribute or it has neither the `cf_returns_retained` attribute nor a *selector family* that implies a retained result, or

- [beginning LLVM 3.6] a load from a `const` non-system global variable.

An expression is known retained if it is an rvalue of *C retainable pointer type* and it is:

- a message send, and the declared method either has the `cf_returns_retained` attribute, or it does not have the `cf_returns_not_retained` attribute but it does have a *selector family* that implies a retained result.

Furthermore:

- a comma expression is classified according to its right-hand side,
- a statement expression is classified according to its result expression, if it has one,
- an lvalue-to-rvalue conversion applied to an Objective-C property lvalue is classified according to the underlying message send, and
- a conditional operator is classified according to its second and third operands, if they agree in classification, or else the other if one is known retain-agnostic.

If the cast operand is known retained, the conversion is treated as a `__bridge_transfer` cast. If the cast operand is known unretained or known retain-agnostic, the conversion is treated as a `__bridge` cast.

Rationale

Bridging casts are annoying. Absent the ability to completely automate the management of CF objects, however, we are left with relatively poor attempts to reduce the need for a glut of explicit bridges. Hence these rules.

We've so far consciously refrained from implicitly turning retained CF results from function calls into `__bridge_transfer` casts. The worry is that some code patterns — for example, creating a CF value, assigning it to an ObjC-typed local, and then calling `CFRelease` when done — are a bit too likely to be accidentally accepted, leading to mysterious behavior.

For loads from `const` global variables of *C retainable pointer type*, it is reasonable to assume that global system constants were initialized with true constants (e.g. string literals), but user constants might have been initialized with something dynamically allocated, using a global initializer.

Conversion from retainable object pointer type in certain contexts

[beginning Apple 4.0, LLVM 3.1]

If an expression of retainable object pointer type is explicitly cast to a *C retainable pointer type*, the program is ill-formed as discussed above unless the result is immediately used:

- to initialize a parameter in an Objective-C message send where the parameter is not marked with the `cf_consumed` attribute, or
- to initialize a parameter in a direct call to an *audited* function where the parameter is not marked with the `cf_consumed` attribute.

Rationale

Consumed parameters are left out because ARC would naturally balance them with a `retain`, which was judged too treacherous. This is in part because several of the most common consuming functions are in the `Release` family, and it would be quite unfortunate for explicit releases to be silently balanced out in this way.

Ownership qualification

This section describes the behavior of *objects* of retainable object pointer type; that is, locations in memory which store retainable object pointers.

A type is a retainable object owner type if it is a retainable object pointer type or an array type whose element type is a retainable object owner type.

An ownership qualifier is a type qualifier which applies only to retainable object owner types. An array type is ownership-qualified according to its element type, and adding an ownership qualifier to an array type so qualifies its element type.

A program is ill-formed if it attempts to apply an ownership qualifier to a type which is already ownership-qualified, even if it is the same qualifier. There is a single exception to this rule: an ownership qualifier may be applied to a substituted template type parameter, which overrides the ownership qualifier provided by the template argument.

When forming a function type, the result type is adjusted so that any top-level ownership qualifier is deleted.

Except as described under the *inference rules*, a program is ill-formed if it attempts to form a pointer or reference type to a retainable object owner type which lacks an ownership qualifier.

Rationale

These rules, together with the inference rules, ensure that all objects and lvalues of retainable object pointer type have an ownership qualifier. The ability to override an ownership qualifier during template substitution is required to counteract the *inference of `__strong` for template type arguments*. Ownership qualifiers on return types are dropped because they serve no purpose there except to cause spurious problems with overloading and templates.

There are four ownership qualifiers:

- `__autoreleasing`
- `__strong`
- `__unsafe_unretained`
- `__weak`

A type is nontrivially ownership-qualified if it is qualified with `__autoreleasing`, `__strong`, or `__weak`.

Spelling

The names of the ownership qualifiers are reserved for the implementation. A program may not assume that they are or are not implemented with macros, or what those macros expand to.

An ownership qualifier may be written anywhere that any other type qualifier may be written.

If an ownership qualifier appears in the *declaration-specifiers*, the following rules apply:

- if the type specifier is a retainable object owner type, the qualifier initially applies to that type;
- otherwise, if the outermost non-array declarator is a pointer or block pointer declarator, the qualifier initially applies to that type;
- otherwise the program is ill-formed.
- If the qualifier is so applied at a position in the declaration where the next-innermost declarator is a function declarator, and there is an block declarator within that function declarator, then the qualifier applies instead to that block declarator and this rule is considered afresh beginning from the new position.

If an ownership qualifier appears on the declarator name, or on the declared object, it is applied to the innermost pointer or block-pointer type.

If an ownership qualifier appears anywhere else in a declarator, it applies to the type there.

Rationale

Ownership qualifiers are like `const` and `volatile` in the sense that they may sensibly apply at multiple distinct positions within a declarator. However, unlike those qualifiers, there are many situations where they are not meaningful, and so we make an effort to “move” the qualifier to a place where it will be meaningful. The general goal is to allow the programmer to write, say, `__strong` before the entire declaration and have it apply in the leftmost sensible place.

Property declarations

A property of retainable object pointer type may have ownership. If the property’s type is ownership-qualified, then the property has that ownership. If the property has one of the following modifiers, then the property has the corresponding ownership. A property is ill-formed if it has conflicting sources of ownership, or if it has redundant ownership modifiers, or if it has `__autoreleasing` ownership.

- `assign` implies `__unsafe_unretained` ownership.
- `copy` implies `__strong` ownership, as well as the usual behavior of copy semantics on the setter.
- `retain` implies `__strong` ownership.
- `strong` implies `__strong` ownership.
- `unsafe_unretained` implies `__unsafe_unretained` ownership.
- `weak` implies `__weak` ownership.

With the exception of `weak`, these modifiers are available in non-ARC modes.

A property’s specified ownership is preserved in its metadata, but otherwise the meaning is purely conventional unless the property is synthesized. If a property is synthesized, then the associated instance variable is the instance variable which is named, possibly implicitly, by the `@synthesize` declaration. If the associated instance variable already exists, then its ownership qualification must equal the ownership of the property; otherwise, the instance variable is created with that ownership qualification.

A property of retainable object pointer type which is synthesized without a source of ownership has the ownership of its associated instance variable, if it already exists; otherwise, [beginning Apple 3.1, LLVM 3.1] its ownership is implicitly `strong`. Prior to this revision, it was ill-formed to synthesize such a property.

Rationale

Using `strong` by default is safe and consistent with the generic ARC rule about *inferring ownership*. It is, unfortunately, inconsistent with the non-ARC rule which states that such properties are implicitly `assign`. However, that rule is clearly untenable in ARC, since it leads to default-unsafe code. The main merit to banning the properties is to avoid confusion with non-ARC practice, which did not ultimately strike us as sufficient to justify requiring extra syntax and (more importantly) forcing novices to understand ownership rules just to declare a property when the default is so reasonable. Changing the rule away from non-ARC practice was acceptable because we had conservatively banned the synthesis in order to give ourselves exactly this leeway.

Applying `__attribute__((NSObject))` to a property not of retainable object pointer type has the same behavior it does outside of ARC: it requires the property type to be some sort of pointer and permits the use of modifiers

other than `assign`. These modifiers only affect the synthesized getter and setter; direct accesses to the ivar (even if synthesized) still have primitive semantics, and the value in the ivar will not be automatically released during deallocation.

Semantics

There are five managed operations which may be performed on an object of retainable object pointer type. Each qualifier specifies different semantics for each of these operations. It is still undefined behavior to access an object outside of its lifetime.

A load or store with “primitive semantics” has the same semantics as the respective operation would have on an `void*` lvalue with the same alignment and non-ownership qualification.

Reading occurs when performing a lvalue-to-rvalue conversion on an object lvalue.

- For `__weak` objects, the current pointee is retained and then released at the end of the current full-expression. This must execute atomically with respect to assignments and to the final release of the pointee.
- For all other objects, the lvalue is loaded with primitive semantics.

Assignment occurs when evaluating an assignment operator. The semantics vary based on the qualification:

- For `__strong` objects, the new pointee is first retained; second, the lvalue is loaded with primitive semantics; third, the new pointee is stored into the lvalue with primitive semantics; and finally, the old pointee is released. This is not performed atomically; external synchronization must be used to make this safe in the face of concurrent loads and stores.
- For `__weak` objects, the lvalue is updated to point to the new pointee, unless the new pointee is an object currently undergoing deallocation, in which case the lvalue is updated to a null pointer. This must execute atomically with respect to other assignments to the object, to reads from the object, and to the final release of the new pointee.
- For `__unsafe_unretained` objects, the new pointee is stored into the lvalue using primitive semantics.
- For `__autoreleasing` objects, the new pointee is retained, autoreleased, and stored into the lvalue using primitive semantics.

Initialization occurs when an object’s lifetime begins, which depends on its storage duration. Initialization proceeds in two stages:

1. First, a null pointer is stored into the lvalue using primitive semantics. This step is skipped if the object is `__unsafe_unretained`.
2. Second, if the object has an initializer, that expression is evaluated and then assigned into the object using the usual assignment semantics.

Destruction occurs when an object’s lifetime ends. In all cases it is semantically equivalent to assigning a null pointer to the object, with the proviso that of course the object cannot be legally read after the object’s lifetime ends.

Moving occurs in specific situations where an lvalue is “moved from”, meaning that its current pointee will be used but the object may be left in a different (but still valid) state. This arises with `__block` variables and rvalue references in C++. For `__strong` lvalues, moving is equivalent to loading the lvalue with primitive semantics, writing a null pointer to it with primitive semantics, and then releasing the result of the load at the end of the current full-expression. For all other lvalues, moving is equivalent to reading the object.

Restrictions

Weak-unavailable types

It is explicitly permitted for Objective-C classes to not support `__weak` references. It is undefined behavior to perform an operation with weak assignment semantics with a pointer to an Objective-C object whose class does not support `__weak` references.

Rationale

Historically, it has been possible for a class to provide its own reference-count implementation by overriding `retain`, `release`, etc. However, weak references to an object require coordination with its class's reference-count implementation because, among other things, weak loads and stores must be atomic with respect to the final release. Therefore, existing custom reference-count implementations will generally not support weak references without additional effort. This is unavoidable without breaking binary compatibility.

A class may indicate that it does not support weak references by providing the `objc_arc_weak_unavailable` attribute on the class's interface declaration. A retainable object pointer type is **weak-unavailable** if it is a pointer to an (optionally protocol-qualified) Objective-C class `T` where `T` or one of its superclasses has the `objc_arc_weak_unavailable` attribute. A program is ill-formed if it applies the `__weak` ownership qualifier to a weak-unavailable type or if the value operand of a weak assignment operation has a weak-unavailable type.

Storage duration of `__autoreleasing` objects

A program is ill-formed if it declares an `__autoreleasing` object of non-automatic storage duration. A program is ill-formed if it captures an `__autoreleasing` object in a block or, unless by reference, in a C++11 lambda.

Rationale

Autorelease pools are tied to the current thread and scope by their nature. While it is possible to have temporary objects whose instance variables are filled with autoreleased objects, there is no way that ARC can provide any sort of safety guarantee there.

It is undefined behavior if a non-null pointer is assigned to an `__autoreleasing` object while an autorelease pool is in scope and then that object is read after the autorelease pool's scope is left.

Conversion of pointers to ownership-qualified types

A program is ill-formed if an expression of type `T*` is converted, explicitly or implicitly, to the type `U*`, where `T` and `U` have different ownership qualification, unless:

- `T` is qualified with `__strong`, `__autoreleasing`, or `__unsafe_unretained`, and `U` is qualified with both `const` and `__unsafe_unretained`; or
- either `T` or `U` is `cv void`, where `cv` is an optional sequence of non-ownership qualifiers; or
- the conversion is requested with a `reinterpret_cast` in Objective-C++; or
- the conversion is a well-formed *pass-by-writeback*.

The analogous rule applies to `T&` and `U&` in Objective-C++.

Rationale

These rules provide a reasonable level of type-safety for indirect pointers, as long as the underlying memory is not deallocated. The conversion to `const __unsafe_unretained` is permitted because the semantics of reads are equivalent across all these ownership semantics, and that's a very useful and common pattern. The interconversion with `void*` is useful for allocating memory or otherwise escaping the type system, but use it carefully. `reinterpret_cast` is considered to be an obvious enough sign of taking responsibility for any problems.

It is undefined behavior to access an ownership-qualified object through an lvalue of a differently-qualified type, except that any non-`__weak` object may be read through an `__unsafe_unretained` lvalue.

It is undefined behavior if a managed operation is performed on a `__strong` or `__weak` object without a guarantee that it contains a primitive zero bit-pattern, or if the storage for such an object is freed or reused without the object being first assigned a null pointer.

Rationale

ARC cannot differentiate between an assignment operator which is intended to “initialize” dynamic memory and one which is intended to potentially replace a value. Therefore the object's pointer must be valid before letting ARC at it. Similarly, C and Objective-C do not provide any language hooks for destroying objects held in dynamic memory, so it is the programmer's responsibility to avoid leaks (`__strong` objects) and consistency errors (`__weak` objects).

These requirements are followed automatically in Objective-C++ when creating objects of retainable object owner type with `new` or `new[]` and destroying them with `delete`, `delete[]`, or a pseudo-destructor expression. Note that arrays of nontrivially-ownership-qualified type are not ABI compatible with non-ARC code because the element type is non-POD: such arrays that are `new[]`'d in ARC translation units cannot be `delete[]`'d in non-ARC translation units and vice-versa.

Passing to an out parameter by writeback

If the argument passed to a parameter of type `T __autoreleasing *` has type `U oq *`, where `oq` is an ownership qualifier, then the argument is a candidate for pass-by-writeback' if:

- `oq` is `__strong` or `__weak`, and
- it would be legal to initialize a `T __strong *` with a `U __strong *`.

For purposes of overload resolution, an implicit conversion sequence requiring a pass-by-writeback is always worse than an implicit conversion sequence not requiring a pass-by-writeback.

The pass-by-writeback is ill-formed if the argument expression does not have a legal form:

- `&var`, where `var` is a scalar variable of automatic storage duration with retainable object pointer type
 - a conditional expression where the second and third operands are both legal forms
 - a cast whose operand is a legal form
 - a null pointer constant
-

Rationale

The restriction in the form of the argument serves two purposes. First, it makes it impossible to pass the address of an array to the argument, which serves to protect against an otherwise serious risk of mis-inferring an “array” argument as an out-parameter. Second, it makes it much less likely that the user will see confusing aliasing problems due to the implementation, below, where their store to the writeback temporary is not immediately seen in the original argument variable.

A pass-by-writeback is evaluated as follows:

1. The argument is evaluated to yield a pointer `p` of type `U oq *`.
2. If `p` is a null pointer, then a null pointer is passed as the argument, and no further work is required for the pass-by-writeback.
3. Otherwise, a temporary of type `T __autoreleasing` is created and initialized to a null pointer.
4. If the parameter is not an Objective-C method parameter marked `out`, then `*p` is read, and the result is written into the temporary with primitive semantics.
5. The address of the temporary is passed as the argument to the actual call.
6. After the call completes, the temporary is loaded with primitive semantics, and that value is assigned into `*p`.

Rationale

This is all admittedly convoluted. In an ideal world, we would see that a local variable is being passed to an out-parameter and retroactively modify its type to be `__autoreleasing` rather than `__strong`. This would be remarkably difficult and not always well-founded under the C type system. However, it was judged unacceptably invasive to require programmers to write `__autoreleasing` on all the variables they intend to use for out-parameters. This was the least bad solution.

Ownership-qualified fields of structs and unions

A program is ill-formed if it declares a member of a C struct or union to have a nontrivially ownership-qualified type.

Rationale

The resulting type would be non-POD in the C++ sense, but C does not give us very good language tools for managing the lifetime of aggregates, so it is more convenient to simply forbid them. It is still possible to manage this with a `void*` or an `__unsafe_unretained` object.

This restriction does not apply in Objective-C++. However, nontrivially ownership-qualified types are considered non-POD: in C++11 terms, they are not trivially default constructible, copy constructible, move constructible, copy assignable, move assignable, or destructible. It is a violation of C++'s One Definition Rule to use a class outside of ARC that, under ARC, would have a nontrivially ownership-qualified member.

Rationale

Unlike in C, we can express all the necessary ARC semantics for ownership-qualified subobjects as suboperations of the (default) special member functions for the class. These functions then become non-trivial. This has the non-obvious result that the class will have a non-trivial copy constructor and non-trivial destructor; if this would not normally be true outside of ARC, objects of the type will be passed and returned in an ABI-incompatible manner.

Ownership inference

Objects

If an object is declared with retainable object owner type, but without an explicit ownership qualifier, its type is implicitly adjusted to have `__strong` qualification.

As a special case, if the object's base type is `Class` (possibly protocol-qualified), the type is adjusted to have `__unsafe_unretained` qualification instead.

Indirect parameters

If a function or method parameter has type `T*`, where `T` is an ownership-unqualified retainable object pointer type, then:

- if `T` is `const`-qualified or `Class`, then it is implicitly qualified with `__unsafe_unretained`;
- otherwise, it is implicitly qualified with `__autoreleasing`.

Rationale

`__autoreleasing` exists mostly for this case, the Cocoa convention for out-parameters. Since a pointer to `const` is obviously not an out-parameter, we instead use a type more useful for passing arrays. If the user instead intends to pass in a *mutable* array, inferring `__autoreleasing` is the wrong thing to do; this directs some of the caution in the following rules about writeback.

Such a type written anywhere else would be ill-formed by the general rule requiring ownership qualifiers.

This rule does not apply in Objective-C++ if a parameter's type is dependent in a template pattern and is only *instantiated* to a type which would be a pointer to an unqualified retainable object pointer type. Such code is still ill-formed.

Rationale

The convention is very unlikely to be intentional in template code.

Template arguments

If a template argument for a template type parameter is an retainable object owner type that does not have an explicit ownership qualifier, it is adjusted to have `__strong` qualification. This adjustment occurs regardless of whether the template argument was deduced or explicitly specified.

Rationale

`__strong` is a useful default for containers (e.g., `std::vector<id>`), which would otherwise require explicit qualification. Moreover, unqualified retainable object pointer types are unlikely to be useful within templates, since they generally need to have a qualifier applied to the before being used.

Method families

An Objective-C method may fall into a method family, which is a conventional set of behaviors ascribed to it by the Cocoa conventions.

A method is in a certain method family if:

- it has a `objc_method_family` attribute placing it in that family; or if not that,
- it does not have an `objc_method_family` attribute placing it in a different or no family, and
- its selector falls into the corresponding selector family, and

- its signature obeys the added restrictions of the method family.

A selector is in a certain selector family if, ignoring any leading underscores, the first component of the selector either consists entirely of the name of the method family or it begins with that name followed by a character other than a lowercase letter. For example, `_perform:with:` and `performWith:` would fall into the `perform` family (if we recognized one), but `performing:with` would not.

The families and their added restrictions are:

- `alloc` methods must return a retainable object pointer type.
- `copy` methods must return a retainable object pointer type.
- `mutableCopy` methods must return a retainable object pointer type.
- `new` methods must return a retainable object pointer type.
- `init` methods must be instance methods and must return an Objective-C pointer type. Additionally, a program is ill-formed if it declares or contains a call to an `init` method whose return type is neither `id` nor a pointer to a super-class or sub-class of the declaring class (if the method was declared on a class) or the static receiver type of the call (if it was declared on a protocol).

Rationale

There are a fair number of existing methods with `init`-like selectors which nonetheless don't follow the `init` conventions. Typically these are either accidental naming collisions or helper methods called during initialization. Because of the peculiar retain/release behavior of `init` methods, it's very important not to treat these methods as `init` methods if they aren't meant to be. It was felt that implicitly defining these methods out of the family based on the exact relationship between the return type and the declaring class would be much too subtle and fragile. Therefore we identify a small number of legitimate-seeming return types and call everything else an error. This serves the secondary purpose of encouraging programmers not to accidentally give methods names in the `init` family.

Note that a method with an `init`-family selector which returns a non-Objective-C type (e.g. `void`) is perfectly well-formed; it simply isn't in the `init` family.

A program is ill-formed if a method's declarations, implementations, and overrides do not all have the same method family.

Explicit method family control

A method may be annotated with the `objc_method_family` attribute to precisely control which method family it belongs to. If a method in an `@implementation` does not have this attribute, but there is a method declared in the corresponding `@interface` that does, then the attribute is copied to the declaration in the `@implementation`. The attribute is available outside of ARC, and may be tested for with the preprocessor query `__has_attribute(objc_method_family)`.

The attribute is spelled `__attribute__((objc_method_family(family)))`. If *family* is `none`, the method has no family, even if it would otherwise be considered to have one based on its selector and type. Otherwise, *family* must be one of `alloc`, `copy`, `init`, `mutableCopy`, or `new`, in which case the method is considered to belong to the corresponding family regardless of its selector. It is an error if a method that is explicitly added to a family in this way does not meet the requirements of the family other than the selector naming convention.

Rationale

The rules codified in this document describe the standard conventions of Objective-C. However, as these conventions have not heretofore been enforced by an unforgiving mechanical system, they are only imperfectly kept, especially as

they haven't always even been precisely defined. While it is possible to define low-level ownership semantics with attributes like `ns_returns_retained`, this attribute allows the user to communicate semantic intent, which is of use both to ARC (which, e.g., treats calls to `init` specially) and the static analyzer.

Semantics of method families

A method's membership in a method family may imply non-standard semantics for its parameters and return type.

Methods in the `alloc`, `copy`, `mutableCopy`, and `new` families — that is, methods in all the currently-defined families except `init` — implicitly *return a retained object* as if they were annotated with the `ns_returns_retained` attribute. This can be overridden by annotating the method with either of the `ns_returns_autoreleased` or `ns_returns_not_retained` attributes.

Properties also follow same naming rules as methods. This means that those in the `alloc`, `copy`, `mutableCopy`, and `new` families provide access to *retained objects*. This can be overridden by annotating the property with `ns_returns_not_retained` attribute.

Semantics of `init`

Methods in the `init` family implicitly *consume* their `self` parameter and *return a retained object*. Neither of these properties can be altered through attributes.

A call to an `init` method with a receiver that is either `self` (possibly parenthesized or casted) or `super` is called a delegate `init` call. It is an error for a delegate `init` call to be made except from an `init` method, and excluding blocks within such methods.

As an exception to the *usual rule*, the variable `self` is mutable in an `init` method and has the usual semantics for a `__strong` variable. However, it is undefined behavior and the program is ill-formed, no diagnostic required, if an `init` method attempts to use the previous value of `self` after the completion of a delegate `init` call. It is conventional, but not required, for an `init` method to return `self`.

It is undefined behavior for a program to cause two or more calls to `init` methods on the same object, except that each `init` method invocation may perform at most one delegate `init` call.

Related result types

Certain methods are candidates to have related result types:

- class methods in the `alloc` and `new` method families
- instance methods in the `init` family
- the instance method `self`
- outside of ARC, the instance methods `retain` and `autorelease`

If the formal result type of such a method is `id` or protocol-qualified `id`, or a type equal to the declaring class or a superclass, then it is said to have a related result type. In this case, when invoked in an explicit message send, it is assumed to return a type related to the type of the receiver:

- if it is a class method, and the receiver is a class name `T`, the message send expression has type `T*`; otherwise
- if it is an instance method, and the receiver has type `T`, the message send expression has type `T`; otherwise
- the message send expression has the normal result type of the method.

This is a new rule of the Objective-C language and applies outside of ARC.

Rationale

ARC's automatic code emission is more prone than most code to signature errors, i.e. errors where a call was emitted against one method signature, but the implementing method has an incompatible signature. Having more precise type information helps drastically lower this risk, as well as catching a number of latent bugs.

Optimization

Within this section, the word function will be used to refer to any structured unit of code, be it a C function, an Objective-C method, or a block.

This specification describes ARC as performing specific `retain` and `release` operations on retainable object pointers at specific points during the execution of a program. These operations make up a non-contiguous subsequence of the computation history of the program. The portion of this sequence for a particular retainable object pointer for which a specific function execution is directly responsible is the formal local retain history of the object pointer. The corresponding actual sequence executed is the *dynamic local retain history*.

However, under certain circumstances, ARC is permitted to re-order and eliminate operations in a manner which may alter the overall computation history beyond what is permitted by the general “as if” rule of C/C++ and the *restrictions* on the implementation of `retain` and `release`.

Rationale

Specifically, ARC is sometimes permitted to optimize `release` operations in ways which might cause an object to be deallocated before it would otherwise be. Without this, it would be almost impossible to eliminate any `retain/release` pairs. For example, consider the following code:

```
id x = _ivar;
[x foo];
```

If we were not permitted in any event to shorten the lifetime of the object in `x`, then we would not be able to eliminate this `retain` and `release` unless we could prove that the message `send` could not modify `_ivar` (or deallocate `self`). Since message sends are opaque to the optimizer, this is not possible, and so ARC's hands would be almost completely tied.

ARC makes no guarantees about the execution of a computation history which contains undefined behavior. In particular, ARC makes no guarantees in the presence of race conditions.

ARC may assume that any retainable object pointers it receives or generates are instantaneously valid from that point until a point which, by the concurrency model of the host language, happens-after the generation of the pointer and happens-before a `release` of that object (possibly via an aliasing pointer or indirectly due to destruction of a different object).

Rationale

There is very little point in trying to guarantee correctness in the presence of race conditions. ARC does not have a stack-scanning garbage collector, and guaranteeing the atomicity of every load and store operation would be prohibitive and preclude a vast amount of optimization.

ARC may assume that non-ARC code engages in sensible balancing behavior and does not rely on exact or minimum retain count values except as guaranteed by `__strong` object invariants or `+1` transfer conventions. For example, if

an object is provably double-retained and double-released, ARC may eliminate the inner retain and release; it does not need to guard against code which performs an unbalanced release followed by a “balancing” retain.

Object liveness

ARC may not allow a retainable object *X* to be deallocated at a time *T* in a computation history if:

- *X* is the value stored in a `__strong` object *S* with *precise lifetime semantics*, or
- *X* is the value stored in a `__strong` object *S* with imprecise lifetime semantics and, at some point after *T* but before the next store to *S*, the computation history features a load from *S* and in some way depends on the value loaded, or
- *X* is a value described as being released at the end of the current full-expression and, at some point after *T* but before the end of the full-expression, the computation history depends on that value.

Rationale

The intent of the second rule is to say that objects held in normal `__strong` local variables may be released as soon as the value in the variable is no longer being used: either the variable stops being used completely or a new value is stored in the variable.

The intent of the third rule is to say that return values may be released after they’ve been used.

A computation history depends on a pointer value *P* if it:

- performs a pointer comparison with *P*,
- loads from *P*,
- stores to *P*,
- depends on a pointer value *Q* derived via pointer arithmetic from *P* (including an instance-variable or field access), or
- depends on a pointer value *Q* loaded from *P*.

Dependency applies only to values derived directly or indirectly from a particular expression result and does not occur merely because a separate pointer value dynamically aliases *P*. Furthermore, this dependency is not carried by values that are stored to objects.

Rationale

The restrictions on dependency are intended to make this analysis feasible by an optimizer with only incomplete information about a program. Essentially, dependence is carried to “obvious” uses of a pointer. Merely passing a pointer argument to a function does not itself cause dependence, but since generally the optimizer will not be able to prove that the function doesn’t depend on that parameter, it will be forced to conservatively assume it does.

Dependency propagates to values loaded from a pointer because those values might be invalidated by deallocating the object. For example, given the code `__strong id x = p->ivar;`, ARC must not move the release of *p* to between the load of `p->ivar` and the retain of that value for storing into *x*.

Dependency does not propagate through stores of dependent pointer values because doing so would allow dependency to outlive the full-expression which produced the original value. For example, the address of an instance variable could be written to some global location and then freely accessed during the lifetime of the local, or a function could return an inner pointer of an object and store it to a local. These cases would be potentially impossible to reason about and so would basically prevent any optimizations based on imprecise lifetime. There are also uncommon enough to make it reasonable to require the precise-lifetime annotation if someone really wants to rely on them.

Dependency does propagate through return values of pointer type. The compelling source of need for this rule is a property accessor which returns an un-autoreleased result; the calling function must have the chance to operate on the value, e.g. to retain it, before ARC releases the original pointer. Note again, however, that dependence does not survive a store, so ARC does not guarantee the continued validity of the return value past the end of the full-expression.

No object lifetime extension

If, in the formal computation history of the program, an object `X` has been deallocated by the time of an observable side-effect, then ARC must cause `X` to be deallocated by no later than the occurrence of that side-effect, except as influenced by the re-ordering of the destruction of objects.

Rationale

This rule is intended to prohibit ARC from observably extending the lifetime of a retainable object, other than as specified in this document. Together with the rule limiting the transformation of releases, this rule requires ARC to eliminate retains and release only in pairs.

ARC's power to reorder the destruction of objects is critical to its ability to do any optimization, for essentially the same reason that it must retain the power to decrease the lifetime of an object. Unfortunately, while it's generally poor style for the destruction of objects to have arbitrary side-effects, it's certainly possible. Hence the caveat.

Precise lifetime semantics

In general, ARC maintains an invariant that a retainable object pointer held in a `__strong` object will be retained for the full formal lifetime of the object. Objects subject to this invariant have precise lifetime semantics.

By default, local variables of automatic storage duration do not have precise lifetime semantics. Such objects are simply strong references which hold values of retainable object pointer type, and these values are still fully subject to the optimizations on values under local control.

Rationale

Applying these precise-lifetime semantics strictly would be prohibitive. Many useful optimizations that might theoretically decrease the lifetime of an object would be rendered impossible. Essentially, it promises too much.

A local variable of retainable object owner type and automatic storage duration may be annotated with the `objc_precise_lifetime` attribute to indicate that it should be considered to be an object with precise lifetime semantics.

Rationale

Nonetheless, it is sometimes useful to be able to force an object to be released at a precise time, even if that object does not appear to be used. This is likely to be uncommon enough that the syntactic weight of explicitly requesting these semantics will not be burdensome, and may even make the code clearer.

Miscellaneous

Special methods

Memory management methods

A program is ill-formed if it contains a method definition, message send, or `@selector` expression for any of the following selectors:

- `autorelease`
- `release`
- `retain`
- `retainCount`

Rationale

`retainCount` is banned because ARC robs it of consistent semantics. The others were banned after weighing three options for how to deal with message sends:

Honoring them would work out very poorly if a programmer naively or accidentally tried to incorporate code written for manual retain/release code into an ARC program. At best, such code would do twice as much work as necessary; quite frequently, however, ARC and the explicit code would both try to balance the same retain, leading to crashes. The cost is losing the ability to perform “unrooted” retains, i.e. retains not logically corresponding to a strong reference in the object graph.

Ignoring them would badly violate user expectations about their code. While it *would* make it easier to develop code simultaneously for ARC and non-ARC, there is very little reason to do so except for certain library developers. ARC and non-ARC translation units share an execution model and can seamlessly interoperate. Within a translation unit, a developer who faithfully maintains their code in non-ARC mode is suffering all the restrictions of ARC for zero benefit, while a developer who isn’t testing the non-ARC mode is likely to be unpleasantly surprised if they try to go back to it.

Banning them has the disadvantage of making it very awkward to migrate existing code to ARC. The best answer to that, given a number of other changes and restrictions in ARC, is to provide a specialized tool to assist users in that migration.

Implementing these methods was banned because they are too integral to the semantics of ARC; many tricks which worked tolerably under manual reference counting will misbehave if ARC performs an ephemeral extra retain or two. If absolutely required, it is still possible to implement them in non-ARC code, for example in a category; the implementations must obey the *semantics* laid out elsewhere in this document.

`dealloc`

A program is ill-formed if it contains a message send or `@selector` expression for the selector `dealloc`.

Rationale

There are no legitimate reasons to call `dealloc` directly.

A class may provide a method definition for an instance method named `dealloc`. This method will be called after the final `release` of the object but before it is deallocated or any of its instance variables are destroyed. The superclass’s implementation of `dealloc` will be called automatically when the method returns.

Rationale

Even though ARC destroys instance variables automatically, there are still legitimate reasons to write a `dealloc` method, such as freeing non-retainable resources. Failing to call `[super dealloc]` in such a method is nearly always a bug. Sometimes, the object is simply trying to prevent itself from being destroyed, but `dealloc` is really far too late for the object to be raising such objections. Somewhat more legitimately, an object may have been pool-allocated and should not be deallocated with `free`; for now, this can only be supported with a `dealloc` implementation outside of ARC. Such an implementation must be very careful to do all the other work that `NSObject`'s `dealloc` would, which is outside the scope of this document to describe.

The instance variables for an ARC-compiled class will be destroyed at some point after control enters the `dealloc` method for the root class of the class. The ordering of the destruction of instance variables is unspecified, both within a single class and between subclasses and superclasses.

Rationale

The traditional, non-ARC pattern for destroying instance variables is to destroy them immediately before calling `[super dealloc]`. Unfortunately, message sends from the superclass are quite capable of reaching methods in the subclass, and those methods may well read or write to those instance variables. Making such message sends from `dealloc` is generally discouraged, since the subclass may well rely on other invariants that were broken during `dealloc`, but it's not so inescapably dangerous that we felt comfortable calling it undefined behavior. Therefore we chose to delay destroying the instance variables to a point at which message sends are clearly disallowed: the point at which the root class's deallocation routines take over.

In most code, the difference is not observable. It can, however, be observed if an instance variable holds a strong reference to an object whose deallocation will trigger a side-effect which must be carefully ordered with respect to the destruction of the super class. Such code violates the design principle that semantically important behavior should be explicit. A simple fix is to clear the instance variable manually during `dealloc`; a more holistic solution is to move semantically important side-effects out of `dealloc` and into a separate teardown phase which can rely on working with well-formed objects.

@autoreleasepool

To simplify the use of autorelease pools, and to bring them under the control of the compiler, a new kind of statement is available in Objective-C. It is written `@autoreleasepool` followed by a *compound-statement*, i.e. by a new scope delimited by curly braces. Upon entry to this block, the current state of the autorelease pool is captured. When the block is exited normally, whether by fallthrough or directed control flow (such as `return` or `break`), the autorelease pool is restored to the saved state, releasing all the objects in it. When the block is exited with an exception, the pool is not drained.

`@autoreleasepool` may be used in non-ARC translation units, with equivalent semantics.

A program is ill-formed if it refers to the `NSAutoreleasePool` class.

Rationale

Autorelease pools are clearly important for the compiler to reason about, but it is far too much to expect the compiler to accurately reason about control dependencies between two calls. It is also very easy to accidentally forget to drain an autorelease pool when using the manual API, and this can significantly inflate the process's high-water-mark. The introduction of a new scope is unfortunate but basically required for sane interaction with the rest of the language. Not draining the pool during an unwind is apparently required by the Objective-C exceptions implementation.

self

The `self` parameter variable of an Objective-C method is never actually retained by the implementation. It is undefined behavior, or at least dangerous, to cause an object to be deallocated during a message send to that object.

To make this safe, for Objective-C instance methods `self` is implicitly `const` unless the method is in the *init family*. Further, `self` is **always** implicitly `const` within a class method.

Rationale

The cost of retaining `self` in all methods was found to be prohibitive, as it tends to be live across calls, preventing the optimizer from proving that the retain and release are unnecessary — for good reason, as it's quite possible in theory to cause an object to be deallocated during its execution without this retain and release. Since it's extremely uncommon to actually do so, even unintentionally, and since there's no natural way for the programmer to remove this retain/release pair otherwise (as there is for other parameters by, say, making the variable `__unsafe_unretained`), we chose to make this optimizing assumption and shift some amount of risk to the user.

Fast enumeration iteration variables

If a variable is declared in the condition of an Objective-C fast enumeration loop, and the variable has no explicit ownership qualifier, then it is qualified with `const __strong` and objects encountered during the enumeration are not actually retained.

Rationale

This is an optimization made possible because fast enumeration loops promise to keep the objects retained during enumeration, and the collection itself cannot be synchronously modified. It can be overridden by explicitly qualifying the variable with `__strong`, which will make the variable mutable again and cause the loop to retain the objects it encounters.

Blocks

The implicit `const` capture variables created when evaluating a block literal expression have the same ownership semantics as the local variables they capture. The capture is performed by reading from the captured variable and initializing the capture variable with that value; the capture variable is destroyed when the block literal is, i.e. at the end of the enclosing scope.

The *inference* rules apply equally to `__block` variables, which is a shift in semantics from non-ARC, where `__block` variables did not implicitly retain during capture.

`__block` variables of retainable object owner type are moved off the stack by initializing the heap copy with the result of moving from the stack copy.

With the exception of retains done as part of initializing a `__strong` parameter variable or reading a `__weak` variable, whenever these semantics call for retaining a value of block-pointer type, it has the effect of a `Block_copy`. The optimizer may remove such copies when it sees that the result is used only as an argument to a call.

Exceptions

By default in Objective C, ARC is not exception-safe for normal releases:

- It does not end the lifetime of `__strong` variables when their scopes are abnormally terminated by an exception.
- It does not perform releases which would occur at the end of a full-expression if that full-expression throws an exception.

A program may be compiled with the option `-fobjc-arc-exceptions` in order to enable these, or with the option `-fno-objc-arc-exceptions` to explicitly disable them, with the last such argument “winning”.

Rationale

The standard Cocoa convention is that exceptions signal programmer error and are not intended to be recovered from. Making code exceptions-safe by default would impose severe runtime and code size penalties on code that typically does not actually care about exceptions safety. Therefore, ARC-generated code leaks by default on exceptions, which is just fine if the process is going to be immediately terminated anyway. Programs which do care about recovering from exceptions should enable the option.

In Objective-C++, `-fobjc-arc-exceptions` is enabled by default.

Rationale

C++ already introduces pervasive exceptions-cleanup code of the sort that ARC introduces. C++ programmers who have not already disabled exceptions are much more likely to actually require exception-safety.

ARC does end the lifetimes of `__weak` objects when an exception terminates their scope unless exceptions are disabled in the compiler.

Rationale

The consequence of a local `__weak` object not being destroyed is very likely to be corruption of the Objective-C runtime, so we want to be safer here. Of course, potentially massive leaks are about as likely to take down the process as this corruption is if the program does try to recover from exceptions.

Interior pointers

An Objective-C method returning a non-retainable pointer may be annotated with the `objc_returns_inner_pointer` attribute to indicate that it returns a handle to the internal data of an object, and that this reference will be invalidated if the object is destroyed. When such a message is sent to an object, the object’s lifetime will be extended until at least the earliest of:

- the last use of the returned pointer, or any pointer derived from it, in the calling function or
- the autorelease pool is restored to a previous state.

Rationale

Rationale: not all memory and resources are managed with reference counts; it is common for objects to manage private resources in their own, private way. Typically these resources are completely encapsulated within the object, but some classes offer their users direct access for efficiency. If ARC is not aware of methods that return such “interior” pointers, its optimizations can cause the owning object to be reclaimed too soon. This attribute informs ARC that it must tread lightly.

The extension rules are somewhat intentionally vague. The autorelease pool limit is there to permit a simple implementation to simply retain and autorelease the receiver. The other limit permits some amount of optimization. The phrase “derived from” is intended to encompass the results both of pointer transformations, such as casts and arithmetic, and of loading from such derived pointers; furthermore, it applies whether or not such derivations are applied directly in the calling code or by other utility code (for example, the C library routine `strchr`). However, the implementation never need account for uses after a return from the code which calls the method returning an interior pointer.

As an exception, no extension is required if the receiver is loaded directly from a `__strong` object with *precise lifetime semantics*.

Rationale

Implicit autoreleases carry the risk of significantly inflating memory use, so it’s important to provide users a way of avoiding these autoreleases. Tying this to precise lifetime semantics is ideal, as for local variables this requires a very explicit annotation, which allows ARC to trust the user with good cheer.

C retainable pointer types

A type is a C retainable pointer type if it is a pointer to (possibly qualified) `void` or a pointer to a (possibly qualified) `struct` or `class` type.

Rationale

ARC does not manage pointers of CoreFoundation type (or any of the related families of retainable C pointers which interoperate with Objective-C for retain/release operation). In fact, ARC does not even know how to distinguish these types from arbitrary C pointer types. The intent of this concept is to filter out some obviously non-object types while leaving a hook for later tightening if a means of exhaustively marking CF types is made available.

Auditing of C retainable pointer interfaces

[beginning Apple 4.0, LLVM 3.1]

A C function may be marked with the `cf_audited_transfer` attribute to express that, except as otherwise marked with attributes, it obeys the parameter (consuming vs. non-consuming) and return (retained vs. non-retained) conventions for a C function of its name, namely:

- A parameter of C retainable pointer type is assumed to not be consumed unless it is marked with the `cf_consumed` attribute, and
- A result of C retainable pointer type is assumed to not be returned retained unless the function is either marked `cf_returns_retained` or it follows the create/copy naming convention and is not marked `cf_returns_not_retained`.

A function obeys the create/copy naming convention if its name contains as a substring:

- either “Create” or “Copy” not followed by a lowercase letter, or
- either “create” or “copy” not followed by a lowercase letter and not preceded by any letter, whether uppercase or lowercase.

A second attribute, `cf_unknown_transfer`, signifies that a function’s transfer semantics cannot be accurately captured using any of these annotations. A program is ill-formed if it annotates the same function with both `cf_audited_transfer` and `cf_unknown_transfer`.

A pragma is provided to facilitate the mass annotation of interfaces:

```
#pragma clang arc_cf_code_audited begin
...
#pragma clang arc_cf_code_audited end
```

All C functions declared within the extent of this pragma are treated as if annotated with the `cf_audited_transfer` attribute unless they otherwise have the `cf_unknown_transfer` attribute. The pragma is accepted in all language modes. A program is ill-formed if it attempts to change files, whether by including a file or ending the current file, within the extent of this pragma.

It is possible to test for all the features in this section with `__has_feature(arc_cf_code_audited)`.

Rationale

A significant inconvenience in ARC programming is the necessity of interacting with APIs based around C retainable pointers. These features are designed to make it relatively easy for API authors to quickly review and annotate their interfaces, in turn improving the fidelity of tools such as the static analyzer and ARC. The single-file restriction on the pragma is designed to eliminate the risk of accidentally annotating some other header's interfaces.

Runtime support

This section describes the interaction between the ARC runtime and the code generated by the ARC compiler. This is not part of the ARC language specification; instead, it is effectively a language-specific ABI supplement, akin to the “Itanium” generic ABI for C++.

Ownership qualification does not alter the storage requirements for objects, except that it is undefined behavior if a `__weak` object is inadequately aligned for an object of type `id`. The other qualifiers may be used on explicitly under-aligned memory.

The runtime tracks `__weak` objects which holds non-null values. It is undefined behavior to direct modify a `__weak` object which is being tracked by the runtime except through an *`objc_storeWeak`*, *`objc_destroyWeak`*, or *`objc_moveWeak`* call.

The runtime must provide a number of new entrypoints which the compiler may emit, which are described in the remainder of this section.

Rationale

Several of these functions are semantically equivalent to a message send; we emit calls to C functions instead because:

- the machine code to do so is significantly smaller,
- it is much easier to recognize the C functions in the ARC optimizer, and
- a sufficient sophisticated runtime may be able to avoid the message send in common cases.

Several other of these functions are “fused” operations which can be described entirely in terms of other operations. We use the fused operations primarily as a code-size optimization, although in some cases there is also a real potential for avoiding redundant operations in the runtime.

```
id objc_autorelease(id value);
```

Precondition: `value` is null or a pointer to a valid object.

If `value` is null, this call has no effect. Otherwise, it adds the object to the innermost autorelease pool exactly as if the object had been sent the `autorelease` message.

Always returns `value`.

```
void objc_autoreleasePoolPop(void *pool);
```

Precondition: `pool` is the result of a previous call to `objc_autoreleasePoolPush` on the current thread, where neither `pool` nor any enclosing pool have previously been popped.

Releases all the objects added to the given autorelease pool and any autorelease pools it encloses, then sets the current autorelease pool to the pool directly enclosing `pool`.

```
void *objc_autoreleasePoolPush(void);
```

Creates a new autorelease pool that is enclosed by the current pool, makes that the current pool, and returns an opaque “handle” to it.

Rationale

While the interface is described as an explicit hierarchy of pools, the rules allow the implementation to just keep a stack of objects, using the stack depth as the opaque pool handle.

```
id objc_autoreleaseReturnValue(id value);
```

Precondition: `value` is null or a pointer to a valid object.

If `value` is null, this call has no effect. Otherwise, it makes a best effort to hand off ownership of a retain count on the object to a call to `objc_retainAutoreleasedReturnValue` for the same object in an enclosing call frame. If this is not possible, the object is autoreleased as above.

Always returns `value`.

```
void objc_copyWeak(id *dest, id *src);
```

Precondition: `src` is a valid pointer which either contains a null pointer or has been registered as a `__weak` object. `dest` is a valid pointer which has not been registered as a `__weak` object.

`dest` is initialized to be equivalent to `src`, potentially registering it with the runtime. Equivalent to the following code:

```
void objc_copyWeak(id *dest, id *src) {  
    objc_release(objc_initWeak(dest, objc_loadWeakRetained(src)));  
}
```

Must be atomic with respect to calls to `objc_storeWeak` on `src`.

```
void objc_destroyWeak(id *object);
```

Precondition: `object` is a valid pointer which either contains a null pointer or has been registered as a `__weak` object.

`object` is unregistered as a weak object, if it ever was. The current value of `object` is left unspecified; otherwise, equivalent to the following code:

```
void objc_destroyWeak(id *object) {  
    objc_storeWeak(object, nil);  
}
```

Does not need to be atomic with respect to calls to `objc_storeWeak` on `object`.

`id objc_initWeak(id *object, id value);`

Precondition: `object` is a valid pointer which has not been registered as a `__weak` object. `value` is null or a pointer to a valid object.

If `value` is a null pointer or the object to which it points has begun deallocation, `object` is zero-initialized. Otherwise, `object` is registered as a `__weak` object pointing to `value`. Equivalent to the following code:

```
id objc_initWeak(id *object, id value) {  
    *object = nil;  
    return objc_storeWeak(object, value);  
}
```

Returns the value of `object` after the call.

Does not need to be atomic with respect to calls to `objc_storeWeak` on `object`.

`id objc_loadWeak(id *object);`

Precondition: `object` is a valid pointer which either contains a null pointer or has been registered as a `__weak` object.

If `object` is registered as a `__weak` object, and the last value stored into `object` has not yet been deallocated or begun deallocation, retains and autoreleases that value and returns it. Otherwise returns null. Equivalent to the following code:

```
id objc_loadWeak(id *object) {  
    return objc_autorelease(objc_loadWeakRetained(object));  
}
```

Must be atomic with respect to calls to `objc_storeWeak` on `object`.

Rationale

Loading weak references would be inherently prone to race conditions without the retain.

`id objc_loadWeakRetained(id *object);`

Precondition: `object` is a valid pointer which either contains a null pointer or has been registered as a `__weak` object.

If `object` is registered as a `__weak` object, and the last value stored into `object` has not yet been deallocated or begun deallocation, retains that value and returns it. Otherwise returns null.

Must be atomic with respect to calls to `objc_storeWeak` on `object`.

```
void objc_moveWeak(id *dest, id *src);
```

Precondition: `src` is a valid pointer which either contains a null pointer or has been registered as a `__weak` object. `dest` is a valid pointer which has not been registered as a `__weak` object.

`dest` is initialized to be equivalent to `src`, potentially registering it with the runtime. `src` may then be left in its original state, in which case this call is equivalent to *objc_copyWeak*, or it may be left as null.

Must be atomic with respect to calls to `objc_storeWeak` on `src`.

```
void objc_release(id value);
```

Precondition: `value` is null or a pointer to a valid object.

If `value` is null, this call has no effect. Otherwise, it performs a release operation exactly as if the object had been sent the `release` message.

```
id objc_retain(id value);
```

Precondition: `value` is null or a pointer to a valid object.

If `value` is null, this call has no effect. Otherwise, it performs a retain operation exactly as if the object had been sent the `retain` message.

Always returns `value`.

```
id objc_retainAutorelease(id value);
```

Precondition: `value` is null or a pointer to a valid object.

If `value` is null, this call has no effect. Otherwise, it performs a retain operation followed by an autorelease operation. Equivalent to the following code:

```
id objc_retainAutorelease(id value) {  
    return objc_autorelease(objc_retain(value));  
}
```

Always returns `value`.

```
id objc_retainAutoreleaseReturnValue(id value);
```

Precondition: `value` is null or a pointer to a valid object.

If `value` is null, this call has no effect. Otherwise, it performs a retain operation followed by the operation described in *objc_autoreleaseReturnValue*. Equivalent to the following code:

```
id objc_retainAutoreleaseReturnValue(id value) {  
    return objc_autoreleaseReturnValue(objc_retain(value));  
}
```

Always returns `value`.


```
id objc_retainAutoreleasedReturnValue(id value);
```

Precondition: `value` is null or a pointer to a valid object.

If `value` is null, this call has no effect. Otherwise, it attempts to accept a hand off of a retain count from a call to `objc_autoreleaseReturnValue` on `value` in a recently-called function or something it calls. If that fails, it performs a retain operation exactly like `objc_retain`.

Always returns `value`.

```
id objc_retainBlock(id value);
```

Precondition: `value` is null or a pointer to a valid block object.

If `value` is null, this call has no effect. Otherwise, if the block pointed to by `value` is still on the stack, it is copied to the heap and the address of the copy is returned. Otherwise a retain operation is performed on the block exactly as if it had been sent the `retain` message.

```
id objc_storeStrong(id *object, id value);
```

Precondition: `object` is a valid pointer to a `__strong` object which is adequately aligned for a pointer. `value` is null or a pointer to a valid object.

Performs the complete sequence for assigning to a `__strong` object of non-block type⁰. Equivalent to the following code:

```
id objc_storeStrong(id *object, id value) {
    value = [value retain];
    id oldValue = *object;
    *object = value;
    [oldValue release];
    return value;
}
```

Always returns `value`.

```
id objc_storeWeak(id *object, id value);
```

Precondition: `object` is a valid pointer which either contains a null pointer or has been registered as a `__weak` object. `value` is null or a pointer to a valid object.

If `value` is a null pointer or the object to which it points has begun deallocation, `object` is assigned null and unregistered as a `__weak` object. Otherwise, `object` is registered as a `__weak` object or has its registration updated to point to `value`.

Returns the value of `object` after the call.

Introduction

This document describes the language extensions provided by Clang. In addition to the language extensions listed here, Clang aims to support a broad range of GCC extensions. Please see the [GCC manual](#) for more information on these extensions.

⁰ This does not imply that a `__strong` object of block type is an invalid argument to this function. Rather it implies that an `objc_retain` and not an `objc_retainBlock` operation will be emitted if the argument is a block.

Feature Checking Macros

Language extensions can be very useful, but only if you know you can depend on them. In order to allow fine-grain features checks, we support three builtin function-like macros. This allows you to directly test for a feature in your code without having to resort to something like autoconf or fragile “compiler version checks”.

`__has_builtin`

This function-like macro takes a single identifier argument that is the name of a builtin function. It evaluates to 1 if the builtin is supported or 0 if not. It can be used like this:

```
#ifndef __has_builtin           // Optional of course.
#define __has_builtin(x) 0    // Compatibility with non-clang compilers.
#endif

...
#if __has_builtin(__builtin_trap)
    __builtin_trap();
#else
    abort();
#endif
...
```

`__has_feature` and `__has_extension`

These function-like macros take a single identifier argument that is the name of a feature. `__has_feature` evaluates to 1 if the feature is both supported by Clang and standardized in the current language standard or 0 if not (but see *below*), while `__has_extension` evaluates to 1 if the feature is supported by Clang in the current language (either as a language extension or a standard language feature) or 0 if not. They can be used like this:

```
#ifndef __has_feature           // Optional of course.
#define __has_feature(x) 0    // Compatibility with non-clang compilers.
#endif
#ifndef __has_extension
#define __has_extension __has_feature // Compatibility with pre-3.0 compilers.
#endif

...
#if __has_feature(cxx_rvalue_references)
// This code will only be compiled with the -std=c++11 and -std=gnu++11
// options, because rvalue references are only standardized in C++11.
#endif

#if __has_extension(cxx_rvalue_references)
// This code will be compiled with the -std=c++11, -std=gnu++11, -std=c++98
// and -std=gnu++98 options, because rvalue references are supported as a
// language extension in C++98.
#endif
```

For backward compatibility, `__has_feature` can also be used to test for support for non-standardized features, i.e. features not prefixed `c_`, `cxx_` or `objc_`.

Another use of `__has_feature` is to check for compiler features not related to the language standard, such as e.g. *AddressSanitizer*.

If the `-pedantic-errors` option is given, `__has_extension` is equivalent to `__has_feature`.

The feature tag is described along with the language feature below.

The feature name or extension name can also be specified with a preceding and following `__` (double underscore) to avoid interference from a macro with the same name. For instance, `__cxx_rvalue_references__` can be used instead of `cxx_rvalue_references`.

`__has_cpp_attribute`

This function-like macro takes a single argument that is the name of a C++11-style attribute. The argument can either be a single identifier, or a scoped identifier. If the attribute is supported, a nonzero value is returned. If the attribute is a standards-based attribute, this macro returns a nonzero value based on the year and month in which the attribute was voted into the working draft. If the attribute is not supported by the current compilation target, this macro evaluates to 0. It can be used like this:

```
#ifndef __has_cpp_attribute      // Optional of course.
#define __has_cpp_attribute(x) 0 // Compatibility with non-clang compilers.
#endif

...
#if __has_cpp_attribute(clang::fallthrough)
#define FALLTHROUGH [[clang::fallthrough]]
#else
#define FALLTHROUGH
#endif
...
```

The attribute identifier (but not scope) can also be specified with a preceding and following `__` (double underscore) to avoid interference from a macro with the same name. For instance, `gnu::__const__` can be used instead of `gnu::const`.

`__has_attribute`

This function-like macro takes a single identifier argument that is the name of a GNU-style attribute. It evaluates to 1 if the attribute is supported by the current compilation target, or 0 if not. It can be used like this:

```
#ifndef __has_attribute      // Optional of course.
#define __has_attribute(x) 0 // Compatibility with non-clang compilers.
#endif

...
#if __has_attribute(always_inline)
#define ALWAYS_INLINE __attribute__((always_inline))
#else
#define ALWAYS_INLINE
#endif
...
```

The attribute name can also be specified with a preceding and following `__` (double underscore) to avoid interference from a macro with the same name. For instance, `__always_inline__` can be used instead of `always_inline`.

`__has_declspec_attribute`

This function-like macro takes a single identifier argument that is the name of an attribute implemented as a Microsoft-style `__declspec` attribute. It evaluates to 1 if the attribute is supported by the current compilation target, or 0 if not. It can be used like this:

```
#ifndef __has_declspec_attribute      // Optional of course.
#define __has_declspec_attribute(x) 0 // Compatibility with non-clang compilers.
#endif

...
#if __has_declspec_attribute(dlllexport)
#define DLLEXPORT __declspec(dlllexport)
#else
#define DLLEXPORT
#endif
...
```

The attribute name can also be specified with a preceding and following `__` (double underscore) to avoid interference from a macro with the same name. For instance, `__dlllexport__` can be used instead of `dlllexport`.

`__is_identifier`

This function-like macro takes a single identifier argument that might be either a reserved word or a regular identifier. It evaluates to 1 if the argument is just a regular identifier and not a reserved word, in the sense that it can then be used as the name of a user-defined function or variable. Otherwise it evaluates to 0. It can be used like this:

```
...
#ifdef __is_identifier      // Compatibility with non-clang compilers.
    #if __is_identifier(__wchar_t)
        typedef wchar_t __wchar_t;
    #endif
#endif

__wchar_t WideCharacter;
...
```

Include File Checking Macros

Not all development systems have the same include files. The `__has_include` and `__has_include_next` macros allow you to check for the existence of an include file before doing a possibly failing `#include` directive. Include file checking macros must be used as expressions in `#if` or `#elif` preprocessing directives.

`__has_include`

This function-like macro takes a single file name string argument that is the name of an include file. It evaluates to 1 if the file can be found using the include paths, or 0 otherwise:

```
// Note the two possible file name string formats.
#if __has_include("myinclude.h") && __has_include(<stdint.h>)
# include "myinclude.h"
#endif
```

To test for this feature, use `#if defined(__has_include)`:

```
// To avoid problem with non-clang compilers not having this macro.
#if defined(__has_include)
    #if __has_include("myinclude.h")
        # include "myinclude.h"
    #endif
#endif
```

```
#endif
#endif
```

`__has_include_next`

This function-like macro takes a single file name string argument that is the name of an include file. It is like `__has_include` except that it looks for the second instance of the given file found in the include paths. It evaluates to 1 if the second instance of the file can be found using the include paths, or 0 otherwise:

```
// Note the two possible file name string formats.
#if __has_include_next("myinclude.h") && __has_include_next(<stdint.h>)
# include_next "myinclude.h"
#endif

// To avoid problem with non-clang compilers not having this macro.
#if defined(__has_include_next)
#if __has_include_next("myinclude.h")
# include_next "myinclude.h"
#endif
#endif
```

Note that `__has_include_next`, like the GNU extension `#include_next` directive, is intended for use in headers only, and will issue a warning if used in the top-level compilation file. A warning will also be issued if an absolute path is used in the file argument.

`__has_warning`

This function-like macro takes a string literal that represents a command line option for a warning and returns true if that is a valid warning option.

```
#if __has_warning("-Wformat")
...
#endif
```

Builtin Macros

`__BASE_FILE__` Defined to a string that contains the name of the main input file passed to Clang.

`__COUNTER__` Defined to an integer value that starts at zero and is incremented each time the `__COUNTER__` macro is expanded.

`__INCLUDE_LEVEL__` Defined to an integral value that is the include depth of the file currently being translated. For the main file, this value is zero.

`__TIMESTAMP__` Defined to the date and time of the last modification of the current source file.

`__clang__` Defined when compiling with Clang

`__clang_major__` Defined to the major marketing version number of Clang (e.g., the 2 in 2.0.1). Note that marketing version numbers should not be used to check for language features, as different vendors use different numbering schemes. Instead, use the [Feature Checking Macros](#).

`__clang_minor__` Defined to the minor version number of Clang (e.g., the 0 in 2.0.1). Note that marketing version numbers should not be used to check for language features, as different vendors use different numbering schemes. Instead, use the [Feature Checking Macros](#).

`__clang_patchlevel__` Defined to the marketing patch level of Clang (e.g., the 1 in 2.0.1).

`__clang_version__` Defined to a string that captures the Clang marketing version, including the Subversion tag or revision number, e.g., “1.5 (trunk 102332)”.

Vectors and Extended Vectors

Supports the GCC, OpenCL, AltiVec and NEON vector extensions.

OpenCL vector types are created using `ext_vector_type` attribute. It support for `V.xyzw` syntax and other tidbits as seen in OpenCL. An example is:

```
typedef float float4 __attribute__((ext_vector_type(4)));
typedef float float2 __attribute__((ext_vector_type(2)));

float4 foo(float2 a, float2 b) {
    float4 c;
    c.xz = a;
    c.yw = b;
    return c;
}
```

Query for this feature with `__has_extension(attribute_ext_vector_type)`.

Giving `-faltivec` option to clang enables support for AltiVec vector syntax and functions. For example:

```
vector float foo(vector int a) {
    vector int b;
    b = vec_add(a, a) + a;
    return (vector float)b;
}
```

NEON vector types are created using `neon_vector_type` and `neon_polyvector_type` attributes. For example:

```
typedef __attribute__((neon_vector_type(8))) int8_t int8x8_t;
typedef __attribute__((neon_polyvector_type(16))) poly8_t poly8x16_t;

int8x8_t foo(int8x8_t a) {
    int8x8_t v;
    v = a;
    return v;
}
```

Vector Literals

Vector literals can be used to create vectors from a set of scalars, or vectors. Either parentheses or braces form can be used. In the parentheses form the number of literal values specified must be one, i.e. referring to a scalar value, or must match the size of the vector type being created. If a single scalar literal value is specified, the scalar literal value will be replicated to all the components of the vector type. In the brackets form any number of literals can be specified. For example:

```
typedef int v4si __attribute__((__vector_size__(16)));
typedef float float4 __attribute__((ext_vector_type(4)));
typedef float float2 __attribute__((ext_vector_type(2)));
```

```

v4si vsi = (v4si){1, 2, 3, 4};
float4 vf = (float4)(1.0f, 2.0f, 3.0f, 4.0f);
vector<int> vi1 = (vector<int>)(1);    // vi1 will be (1, 1, 1, 1).
vector<int> vi2 = (vector<int>){1};    // vi2 will be (1, 0, 0, 0).
vector<int> vi3 = (vector<int>)(1, 2); // error
vector<int> vi4 = (vector<int>){1, 2}; // vi4 will be (1, 2, 0, 0).
vector<int> vi5 = (vector<int>)(1, 2, 3, 4);
float4 vf = (float4)((float2)(1.0f, 2.0f), (float2)(3.0f, 4.0f));

```

Vector Operations

The table below shows the support for each operation by vector extension. A dash indicates that an operation is not accepted according to a corresponding specification.

Operator	OpenCL	Altivec	GCC	NEON
[]	yes	yes	yes	–
unary operators +, –	yes	yes	yes	–
++, --	yes	yes	yes	–
+, -, *, /, %	yes	yes	yes	–
bitwise operators &, , ^, ~	yes	yes	yes	–
>>, <<	yes	yes	yes	–
!, &&,	yes	–	–	–
==, !=, >, <, >=, <=	yes	yes	–	–
=	yes	yes	yes	yes
:?	yes	–	–	–
sizeof	yes	yes	yes	yes
C-style cast	yes	yes	yes	no
reinterpret_cast	yes	no	yes	no
static_cast	yes	no	yes	no
const_cast	no	no	no	no

See also [__builtin_shufflevector](#), [__builtin_convertvector](#).

Messages on deprecated and unavailable Attributes

An optional string message can be added to the deprecated and unavailable attributes. For example:

```

void explode(void) __attribute__((deprecated("extremely unsafe, use 'combust' instead!
↳!!!"))));

```

If the deprecated or unavailable declaration is used, the message will be incorporated into the appropriate diagnostic:

```

harmless.c:4:3: warning: 'explode' is deprecated: extremely unsafe, use 'combust'
↳instead!!!
    [-Wdeprecated-declarations]
    explode();
    ^

```

Query for this feature with `__has_extension(attribute_deprecated_with_message)` and `__has_extension(attribute_unavailable_with_message)`.

Attributes on Enumerators

Clang allows attributes to be written on individual enumerators. This allows enumerators to be deprecated, made unavailable, etc. The attribute must appear after the enumerator name and before any initializer, like so:

```
enum OperationMode {
    OM_Invalid,
    OM_Normal,
    OM_Terrified __attribute__((deprecated)),
    OM_AbortOnError __attribute__((deprecated)) = 4
};
```

Attributes on the `enum` declaration do not apply to individual enumerators.

Query for this feature with `__has_extension(enumerator_attributes)`.

‘User-Specified’ System Frameworks

Clang provides a mechanism by which frameworks can be built in such a way that they will always be treated as being “system frameworks”, even if they are not present in a system framework directory. This can be useful to system framework developers who want to be able to test building other applications with development builds of their framework, including the manner in which the compiler changes warning behavior for system headers.

Framework developers can opt-in to this mechanism by creating a “.system_framework” file at the top-level of their framework. That is, the framework should have contents like:

```
.../TestFramework.framework
.../TestFramework.framework/.system_framework
.../TestFramework.framework/Headers
.../TestFramework.framework/Headers/TestFramework.h
...
```

Clang will treat the presence of this file as an indicator that the framework should be treated as a system framework, regardless of how it was found in the framework search path. For consistency, we recommend that such files never be included in installed versions of the framework.

Checks for Standard Language Features

The `__has_feature` macro can be used to query if certain standard language features are enabled. The `__has_extension` macro can be used to query if language features are available as an extension when compiling for a standard which does not provide them. The features which can be tested are listed here.

Since Clang 3.4, the C++ SD-6 feature test macros are also supported. These are macros with names of the form `__cpp_<feature_name>`, and are intended to be a portable way to query the supported features of the compiler. See the [C++ status page](#) for information on the version of SD-6 supported by each Clang release, and the macros provided by that revision of the recommendations.

C++98

The features listed below are part of the C++98 standard. These features are enabled by default when compiling C++ code.

C++ exceptions

Use `__has_feature(cxx_exceptions)` to determine if C++ exceptions have been enabled. For example, compiling code with `-fno-exceptions` disables C++ exceptions.

C++ RTTI

Use `__has_feature(cxx_rtti)` to determine if C++ RTTI has been enabled. For example, compiling code with `-fno-rtti` disables the use of RTTI.

C++11

The features listed below are part of the C++11 standard. As a result, all these features are enabled with the `-std=c++11` or `-std=gnu++11` option when compiling C++ code.

C++11 SFINAE includes access control

Use `__has_feature(cxx_access_control_sfinae)` or `__has_extension(cxx_access_control_sfinae)` to determine whether access-control errors (e.g., calling a private constructor) are considered to be template argument deduction errors (aka SFINAE errors), per [C++ DR1170](#).

C++11 alias templates

Use `__has_feature(cxx_alias_templates)` or `__has_extension(cxx_alias_templates)` to determine if support for C++11's alias declarations and alias templates is enabled.

C++11 alignment specifiers

Use `__has_feature(cxx_alignas)` or `__has_extension(cxx_alignas)` to determine if support for alignment specifiers using `alignas` is enabled.

Use `__has_feature(cxx_alignof)` or `__has_extension(cxx_alignof)` to determine if support for the `alignof` keyword is enabled.

C++11 attributes

Use `__has_feature(cxx_attributes)` or `__has_extension(cxx_attributes)` to determine if support for attribute parsing with C++11's square bracket notation is enabled.

C++11 generalized constant expressions

Use `__has_feature(cxx_constexpr)` to determine if support for generalized constant expressions (e.g., `constexpr`) is enabled.

C++11 `decltype()`

Use `__has_feature(cxx_decltype)` or `__has_extension(cxx_decltype)` to determine if support for the `decltype()` specifier is enabled. C++11's `decltype` does not require type-completeness of a function call expression. Use `__has_feature(cxx_decltype_incomplete_return_types)` or `__has_extension(cxx_decltype_incomplete_return_types)` to determine if support for this feature is enabled.

C++11 default template arguments in function templates

Use `__has_feature(cxx_default_function_template_args)` or `__has_extension(cxx_default_function_` to determine if support for default template arguments in function templates is enabled.

C++11 defaulted functions

Use `__has_feature(cxx_defaulted_functions)` or `__has_extension(cxx_defaulted_functions)` to determine if support for defaulted function definitions (with `= default`) is enabled.

C++11 delegating constructors

Use `__has_feature(cxx_delegating_constructors)` to determine if support for delegating constructors is enabled.

C++11 deleted functions

Use `__has_feature(cxx_deleted_functions)` or `__has_extension(cxx_deleted_functions)` to determine if support for deleted function definitions (with `= delete`) is enabled.

C++11 explicit conversion functions

Use `__has_feature(cxx_explicit_conversions)` to determine if support for explicit conversion functions is enabled.

C++11 generalized initializers

Use `__has_feature(cxx_generalized_initializers)` to determine if support for generalized initializers (using braced lists and `std::initializer_list`) is enabled.

C++11 implicit move constructors/assignment operators

Use `__has_feature(cxx_implicit_moves)` to determine if Clang will implicitly generate move constructors and move assignment operators where needed.

C++11 inheriting constructors

Use `__has_feature(cxx_inheriting_constructors)` to determine if support for inheriting constructors is enabled.

C++11 inline namespaces

Use `__has_feature(cxx_inline_namespaces)` or `__has_extension(cxx_inline_namespaces)` to determine if support for inline namespaces is enabled.

C++11 lambdas

Use `__has_feature(cxx_lambdas)` or `__has_extension(cxx_lambdas)` to determine if support for lambdas is enabled.

C++11 local and unnamed types as template arguments

Use `__has_feature(cxx_local_type_template_args)` or `__has_extension(cxx_local_type_template_args)` to determine if support for local and unnamed types as template arguments is enabled.

C++11 noexcept

Use `__has_feature(cxx_noexcept)` or `__has_extension(cxx_noexcept)` to determine if support for noexcept exception specifications is enabled.

C++11 in-class non-static data member initialization

Use `__has_feature(cxx_nonstatic_member_init)` to determine whether in-class initialization of non-static data members is enabled.

C++11 nullptr

Use `__has_feature(cxx_nullptr)` or `__has_extension(cxx_nullptr)` to determine if support for nullptr is enabled.

C++11 override control

Use `__has_feature(cxx_override_control)` or `__has_extension(cxx_override_control)` to determine if support for the override control keywords is enabled.

C++11 reference-qualified functions

Use `__has_feature(cxx_reference_qualified_functions)` or `__has_extension(cxx_reference_qualified_functions)` to determine if support for reference-qualified functions (e.g., member functions with `&` or `&&` applied to `*this`) is enabled.

C++11 range-based for loop

Use `__has_feature(cxx_range_for)` or `__has_extension(cxx_range_for)` to determine if support for the range-based for loop is enabled.

C++11 raw string literals

Use `__has_feature(cxx_raw_string_literals)` to determine if support for raw string literals (e.g., `R"x (foo\bar) x"`) is enabled.

C++11 rvalue references

Use `__has_feature(cxx_rvalue_references)` or `__has_extension(cxx_rvalue_references)` to determine if support for rvalue references is enabled.

C++11 `static_assert()`

Use `__has_feature(cxx_static_assert)` or `__has_extension(cxx_static_assert)` to determine if support for compile-time assertions using `static_assert` is enabled.

C++11 `thread_local`

Use `__has_feature(cxx_thread_local)` to determine if support for `thread_local` variables is enabled.

C++11 type inference

Use `__has_feature(cxx_auto_type)` or `__has_extension(cxx_auto_type)` to determine C++11 type inference is supported using the `auto` specifier. If this is disabled, `auto` will instead be a storage class specifier, as in C or C++98.

C++11 strongly typed enumerations

Use `__has_feature(cxx_strong_enums)` or `__has_extension(cxx_strong_enums)` to determine if support for strongly typed, scoped enumerations is enabled.

C++11 trailing return type

Use `__has_feature(cxx_trailing_return)` or `__has_extension(cxx_trailing_return)` to determine if support for the alternate function declaration syntax with trailing return type is enabled.

C++11 Unicode string literals

Use `__has_feature(cxx_unicode_literals)` to determine if support for Unicode string literals is enabled.

C++11 unrestricted unions

Use `__has_feature(cxx_unrestricted_unions)` to determine if support for unrestricted unions is enabled.

C++11 user-defined literals

Use `__has_feature(cxx_user_literals)` to determine if support for user-defined literals is enabled.

C++11 variadic templates

Use `__has_feature(cxx_variadic_templates)` or `__has_extension(cxx_variadic_templates)` to determine if support for variadic templates is enabled.

C++1y

The features listed below are part of the committee draft for the C++1y standard. As a result, all these features are enabled with the `-std=c++1y` or `-std=gnu++1y` option when compiling C++ code.

C++1y binary literals

Use `__has_feature(cxx_binary_literals)` or `__has_extension(cxx_binary_literals)` to determine whether binary literals (for instance, `0b10010`) are recognized. Clang supports this feature as an extension in all language modes.

C++1y contextual conversions

Use `__has_feature(cxx_contextual_conversions)` or `__has_extension(cxx_contextual_conversions)` to determine if the C++1y rules are used when performing an implicit conversion for an array bound in a *new-expression*, the operand of a *delete-expression*, an integral constant expression, or a condition in a *switch* statement.

C++1y decltype(auto)

Use `__has_feature(cxx_decltype_auto)` or `__has_extension(cxx_decltype_auto)` to determine if support for the `decltype(auto)` placeholder type is enabled.

C++1y default initializers for aggregates

Use `__has_feature(cxx_aggregate_nsdmi)` or `__has_extension(cxx_aggregate_nsdmi)` to determine if support for default initializers in aggregate members is enabled.

C++1y digit separators

Use `__cpp_digit_separators` to determine if support for digit separators using single quotes (for instance, `10'000`) is enabled. At this time, there is no corresponding `__has_feature` name

C++1y generalized lambda capture

Use `__has_feature(cxx_init_captures)` or `__has_extension(cxx_init_captures)` to determine if support for lambda captures with explicit initializers is enabled (for instance, `[n(0)] { return ++n; }`).

C++1y generic lambdas

Use `__has_feature(cxx_generic_lambdas)` or `__has_extension(cxx_generic_lambdas)` to determine if support for generic (polymorphic) lambdas is enabled (for instance, `[] (auto x) { return x + 1; }`).

C++1y relaxed constexpr

Use `__has_feature(cxx_relaxed_constexpr)` or `__has_extension(cxx_relaxed_constexpr)` to determine if variable declarations, local variable modification, and control flow constructs are permitted in `constexpr` functions.

C++1y return type deduction

Use `__has_feature(cxx_return_type_deduction)` or `__has_extension(cxx_return_type_deduction)` to determine if support for return type deduction for functions (using `auto` as a return type) is enabled.

C++1y runtime-sized arrays

Use `__has_feature(cxx_runtime_array)` or `__has_extension(cxx_runtime_array)` to determine if support for arrays of runtime bound (a restricted form of variable-length arrays) is enabled. Clang's implementation of this feature is incomplete.

C++1y variable templates

Use `__has_feature(cxx_variable_templates)` or `__has_extension(cxx_variable_templates)` to determine if support for templated variable declarations is enabled.

C11

The features listed below are part of the C11 standard. As a result, all these features are enabled with the `-std=c11` or `-std=gnu11` option when compiling C code. Additionally, because these features are all backward-compatible, they are available as extensions in all language modes.

C11 alignment specifiers

Use `__has_feature(c_alignas)` or `__has_extension(c_alignas)` to determine if support for alignment specifiers using `_Alignas` is enabled.

Use `__has_feature(c_alignof)` or `__has_extension(c_alignof)` to determine if support for the `_Alignof` keyword is enabled.

C11 atomic operations

Use `__has_feature(c_atomic)` or `__has_extension(c_atomic)` to determine if support for atomic types using `_Atomic` is enabled. Clang also provides *a set of builtins* which can be used to implement the `<stdatomic.h>` operations on `_Atomic` types. Use `__has_include(<stdatomic.h>)` to determine if C11's `<stdatomic.h>` header is available.

Clang will use the system's `<stdatomic.h>` header when one is available, and will otherwise use its own. When using its own, implementations of the atomic operations are provided as macros. In the cases where C11 also requires a real function, this header provides only the declaration of that function (along with a shadowing macro implementation), and you must link to a library which provides a definition of the function if you use it instead of the macro.

C11 generic selections

Use `__has_feature(c_generic_selections)` or `__has_extension(c_generic_selections)` to determine if support for generic selections is enabled.

As an extension, the C11 generic selection expression is available in all languages supported by Clang. The syntax is the same as that given in the C11 standard.

In C, type compatibility is decided according to the rules given in the appropriate standard, but in C++, which lacks the type compatibility rules used in C, types are considered compatible only if they are equivalent.

C11 `_Static_assert()`

Use `__has_feature(c_static_assert)` or `__has_extension(c_static_assert)` to determine if support for compile-time assertions using `_Static_assert` is enabled.

C11 `_Thread_local`

Use `__has_feature(c_thread_local)` or `__has_extension(c_thread_local)` to determine if support for `_Thread_local` variables is enabled.

Modules

Use `__has_feature(modules)` to determine if Modules have been enabled. For example, compiling code with `-fmodules` enables the use of Modules.

More information could be found [here](#).

Checks for Type Trait Primitives

Type trait primitives are special builtin constant expressions that can be used by the standard C++ library to facilitate or simplify the implementation of user-facing type traits in the `<type_traits>` header.

They are not intended to be used directly by user code because they are implementation-defined and subject to change – as such they're tied closely to the supported set of system headers, currently:

- LLVM's own `libc++`
- GNU `libstdc++`
- The Microsoft standard C++ library

Clang supports the [GNU C++ type traits](#) and a subset of the [Microsoft Visual C++ Type traits](#).

Feature detection is supported only for some of the primitives at present. User code should not use these checks because they bear no direct relation to the actual set of type traits supported by the C++ standard library.

For type trait `__X`, `__has_extension(X)` indicates the presence of the type trait primitive in the compiler. A simplistic usage example as might be seen in standard C++ headers follows:

```
#if __has_extension(is_convertible_to)
template<typename From, typename To>
struct is_convertible_to {
    static const bool value = __is_convertible_to(From, To);
};
#else
// Emulate type trait for compatibility with other compilers.
#endif
```

The following type trait primitives are supported by Clang:

- `__has_nothrow_assign` (GNU, Microsoft)
- `__has_nothrow_copy` (GNU, Microsoft)
- `__has_nothrow_constructor` (GNU, Microsoft)
- `__has_trivial_assign` (GNU, Microsoft)
- `__has_trivial_copy` (GNU, Microsoft)
- `__has_trivial_constructor` (GNU, Microsoft)
- `__has_trivial_destructor` (GNU, Microsoft)
- `__has_virtual_destructor` (GNU, Microsoft)
- `__is_abstract` (GNU, Microsoft)
- `__is_base_of` (GNU, Microsoft)
- `__is_class` (GNU, Microsoft)
- `__is_convertible_to` (Microsoft)
- `__is_empty` (GNU, Microsoft)
- `__is_enum` (GNU, Microsoft)
- `__is_interface_class` (Microsoft)
- `__is_pod` (GNU, Microsoft)
- `__is_polymorphic` (GNU, Microsoft)
- `__is_union` (GNU, Microsoft)
- `__is_literal(type)`: Determines whether the given type is a literal type
- `__is_final`: Determines whether the given type is declared with a `final` class-virt-specifier.
- `__underlying_type(type)`: Retrieves the underlying type for a given enum type. This trait is required to implement the C++11 standard library.
- `__is_trivially_assignable(totype, fromtype)`: Determines whether a value of type `totype` can be assigned to from a value of type `fromtype` such that no non-trivial functions are called as part of that assignment. This trait is required to implement the C++11 standard library.

- `__is_trivially_constructible(type, argtypes...)`: Determines whether a value of type `type` can be direct-initialized with arguments of types `argtypes...` such that no non-trivial functions are called as part of that initialization. This trait is required to implement the C++11 standard library.
- `__is_destructible` (MSVC 2013): partially implemented
- `__is_nothrow_destructible` (MSVC 2013): partially implemented
- `__is_nothrow_assignable` (MSVC 2013, clang)
- `__is_constructible` (MSVC 2013, clang)
- `__is_nothrow_constructible` (MSVC 2013, clang)

Blocks

The syntax and high level language feature description is in [BlockLanguageSpec](#). Implementation and ABI details for the clang implementation are in [Block-ABI-Apple](#).

Query for this feature with `__has_extension(blocks)`.

Objective-C Features

Related result types

According to Cocoa conventions, Objective-C methods with certain names (“init”, “alloc”, etc.) always return objects that are an instance of the receiving class’s type. Such methods are said to have a “related result type”, meaning that a message send to one of these methods will have the same static type as an instance of the receiver class. For example, given the following classes:

```
@interface NSObject
+ (id)alloc;
- (id)init;
@end

@interface NSArray : NSObject
@end
```

and this common initialization pattern

```
NSArray *array = [[NSArray alloc] init];
```

the type of the expression `[NSArray alloc]` is `NSArray*` because `alloc` implicitly has a related result type. Similarly, the type of the expression `[[NSArray alloc] init]` is `NSArray*`, since `init` has a related result type and its receiver is known to have the type `NSArray *`. If neither `alloc` nor `init` had a related result type, the expressions would have had type `id`, as declared in the method signature.

A method with a related result type can be declared by using the type `instancetype` as its result type. `instancetype` is a contextual keyword that is only permitted in the result type of an Objective-C method, e.g.

```
@interface A
+ (instancetype)constructAnA;
@end
```

The related result type can also be inferred for some methods. To determine whether a method has an inferred related result type, the first word in the camel-case selector (e.g., “init” in “initWithObjects”) is considered, and the method will have a related result type if its return type is compatible with the type of its class and if:

- the first word is “alloc” or “new”, and the method is a class method, or
- the first word is “autorelease”, “init”, “retain”, or “self”, and the method is an instance method.

If a method with a related result type is overridden by a subclass method, the subclass method must also return a type that is compatible with the subclass type. For example:

```
@interface NSString : NSObject
- (NSUnrelated *)init; // incorrect usage: NSUnrelated is not NSString or a
↳ superclass of NSString
@end
```

Related result types only affect the type of a message send or property access via the given method. In all other respects, a method with a related result type is treated the same way as method that returns `id`.

Use `__has_feature(objc_instancetype)` to determine whether the `instancetype` contextual keyword is available.

Automatic reference counting

Clang provides support for *automated reference counting* in Objective-C, which eliminates the need for manual retain/release/autorelease message sends. There are two feature macros associated with automatic reference counting: `__has_feature(objc_arc)` indicates the availability of automated reference counting in general, while `__has_feature(objc_arc_weak)` indicates that automated reference counting also includes support for `__weak` pointers to Objective-C objects.

Enumerations with a fixed underlying type

Clang provides support for C++11 enumerations with a fixed underlying type within Objective-C. For example, one can write an enumeration type as:

```
typedef enum : unsigned char { Red, Green, Blue } Color;
```

This specifies that the underlying type, which is used to store the enumeration value, is `unsigned char`.

Use `__has_feature(objc_fixed_enum)` to determine whether support for fixed underlying types is available in Objective-C.

Interoperability with C++11 lambdas

Clang provides interoperability between C++11 lambdas and blocks-based APIs, by permitting a lambda to be implicitly converted to a block pointer with the corresponding signature. For example, consider an API such as `NSArray`’s array-sorting method:

```
- (NSArray *)sortedArrayUsingComparator: (NSComparator) cmptr;
```

`NSComparator` is simply a typedef for the block pointer `NSComparisonResult (^)(id, id)`, and parameters of this type are generally provided with block literals as arguments. However, one can also use a C++11 lambda so long as it provides the same signature (in this case, accepting two parameters of type `id` and returning an `NSComparisonResult`):

```
NSArray *array = @[@"string 1", @"string 21", @"string 12", @"String 11",
                  @"String 02"];
const NSStringCompareOptions comparisonOptions
    = NSCaseInsensitiveSearch | NSNumericSearch |
```

```

    NSWidthInsensitiveSearch | NSForcedOrderingSearch;
    NSLocale *currentLocale = [NSLocale currentLocale];
    NSArray *sorted
    = [array sortedArrayUsingComparator:^(id s1, id s2) -> NSComparisonResult {
        NSRange string1Range = NSMakeRange(0, [s1 length]);
        return [s1 compare:s2 options:comparisonOptions
            range:string1Range locale:currentLocale];
    }];
    NSLog(@"sorted: %@", sorted);

```

This code relies on an implicit conversion from the type of the lambda expression (an unnamed, local class type called the *closure type*) to the corresponding block pointer type. The conversion itself is expressed by a conversion operator in that closure type that produces a block pointer with the same signature as the lambda itself, e.g.,

```
operator NSComparisonResult (^)(id, id)() const;
```

This conversion function returns a new block that simply forwards the two parameters to the lambda object (which it captures by copy), then returns the result. The returned block is first copied (with `Block_copy`) and then autoreleased. As an optimization, if a lambda expression is immediately converted to a block pointer (as in the first example, above), then the block is not copied and autoreleased: rather, it is given the same lifetime as a block literal written at that point in the program, which avoids the overhead of copying a block to the heap in the common case.

The conversion from a lambda to a block pointer is only available in Objective-C++, and not in C++ with blocks, due to its use of Objective-C memory management (autorelease).

Object Literals and Subscripting

Clang provides support for *Object Literals and Subscripting* in Objective-C, which simplifies common Objective-C programming patterns, makes programs more concise, and improves the safety of container creation. There are several feature macros associated with object literals and subscripting: `__has_feature(objc_array_literals)` tests the availability of array literals; `__has_feature(objc_dictionary_literals)` tests the availability of dictionary literals; `__has_feature(objc_subscripting)` tests the availability of object subscripting.

Objective-C Autosynthesis of Properties

Clang provides support for autosynthesis of declared properties. Using this feature, clang provides default synthesis of those properties not declared `@dynamic` and not having user provided backing getter and setter methods. `__has_feature(objc_default_synthesize_properties)` checks for availability of this feature in version of clang being used.

Objective-C retaining behavior attributes

In Objective-C, functions and methods are generally assumed to follow the [Cocoa Memory Management](#) conventions for ownership of object arguments and return values. However, there are exceptions, and so Clang provides attributes to allow these exceptions to be documented. These are used by ARC and the [static analyzer](#). Some exceptions may be better described using the `objc_method_family` attribute instead.

Usage: The `ns_returns_retained`, `ns_returns_not_retained`, `ns_returns_autoreleased`, `cf_returns_retained`, and `cf_returns_not_retained` attributes can be placed on methods and functions that return Objective-C or CoreFoundation objects. They are commonly placed at the end of a function prototype or method declaration:

```
id foo() __attribute__((ns_returns_retained));  
  
- (NSString *)bar:(int)x __attribute__((ns_returns_retained));
```

The `*_returns_retained` attributes specify that the returned object has a +1 retain count. The `*_returns_not_retained` attributes specify that the return object has a +0 retain count, even if the normal convention for its selector would be +1. `ns_returns_autoreleased` specifies that the returned object is +0, but is guaranteed to live at least as long as the next flush of an autorelease pool.

Usage: The `ns_consumed` and `cf_consumed` attributes can be placed on a parameter declaration; they specify that the argument is expected to have a +1 retain count, which will be balanced in some way by the function or method. The `ns_consumes_self` attribute can only be placed on an Objective-C method; it specifies that the method expects its `self` parameter to have a +1 retain count, which it will balance in some way.

```
void foo(__attribute__((ns_consumed)) NSString *string);  
  
- (void) bar __attribute__((ns_consumes_self));  
- (void) baz:(id) __attribute__((ns_consumed)) x;
```

Further examples of these attributes are available in the static analyzer's [list of annotations for analysis](#).

Query for these features with `__has_attribute(ns_consumed)`, `__has_attribute(ns_returns_retained)`, etc.

Objective-C++ ABI: protocol-qualifier mangling of parameters

Starting with LLVM 3.4, Clang produces a new mangling for parameters whose type is a qualified-id (e.g., `id<Foo>`). This mangling allows such parameters to be differentiated from those with the regular unqualified `id` type.

This was a non-backward compatible mangling change to the ABI. This change allows proper overloading, and also prevents mangling conflicts with template parameters of protocol-qualified type.

Query the presence of this new mangling with `__has_feature(objc_protocol_qualifier_mangling)`.

Initializer lists for complex numbers in C

clang supports an extension which allows the following in C:

```
#include <math.h>  
#include <complex.h>  
complex float x = { 1.0f, INFINITY }; // Init to (1, Inf)
```

This construct is useful because there is no way to separately initialize the real and imaginary parts of a complex variable in standard C, given that clang does not support `__Imaginary`. (Clang also supports the `__real__` and `__imag__` extensions from gcc, which help in some cases, but are not usable in static initializers.)

Note that this extension does not allow eliding the braces; the meaning of the following two lines is different:

```
complex float x[] = { { 1.0f, 1.0f } }; // [0] = (1, 1)  
complex float x[] = { 1.0f, 1.0f }; // [0] = (1, 0), [1] = (1, 0)
```

This extension also works in C++ mode, as far as that goes, but does not apply to the C++ `std::complex`. (In C++11, list initialization allows the same syntax to be used with `std::complex` with the same meaning.)

Builtin Functions

Clang supports a number of builtin library functions with the same syntax as GCC, including things like `__builtin_nan`, `__builtin_constant_p`, `__builtin_choose_expr`, `__builtin_types_compatible_p`, `__builtin_assume_aligned`, `__sync_fetch_and_add`, etc. In addition to the GCC builtins, Clang supports a number of builtins that GCC does not, which are listed here.

Please note that Clang does not and will not support all of the GCC builtins for vector operations. Instead of using builtins, you should use the functions defined in target-specific header files like `<xmmintrin.h>`, which define portable wrappers for these. Many of the Clang versions of these functions are implemented directly in terms of *extended vector support* instead of builtins, in order to reduce the number of builtins that we need to implement.

`__builtin_assume`

`__builtin_assume` is used to provide the optimizer with a boolean invariant that is defined to be true.

Syntax:

```
__builtin_assume(bool)
```

Example of Use:

```
int foo(int x) {
    __builtin_assume(x != 0);

    // The optimizer may short-circuit this check using the invariant.
    if (x == 0)
        return do_something();

    return do_something_else();
}
```

Description:

The boolean argument to this function is defined to be true. The optimizer may analyze the form of the expression provided as the argument and deduce from that information used to optimize the program. If the condition is violated during execution, the behavior is undefined. The argument itself is never evaluated, so any side effects of the expression will be discarded.

Query for this feature with `__has_builtin(__builtin_assume)`.

`__builtin_readcyclecounter`

`__builtin_readcyclecounter` is used to access the cycle counter register (or a similar low-latency, high-accuracy clock) on those targets that support it.

Syntax:

```
__builtin_readcyclecounter()
```

Example of Use:

```
unsigned long long t0 = __builtin_readcyclecounter();
do_something();
unsigned long long t1 = __builtin_readcyclecounter();
unsigned long long cycles_to_do_something = t1 - t0; // assuming no overflow
```

Description:

The `__builtin_readcyclecounter()` builtin returns the cycle counter value, which may be either global or process/thread-specific depending on the target. As the backing counters often overflow quickly (on the order of seconds) this should only be used for timing small intervals. When not supported by the target, the return value is always zero. This builtin takes no arguments and produces an unsigned long long result.

Query for this feature with `__has_builtin(__builtin_readcyclecounter)`. Note that even if present, its use may depend on run-time privilege or other OS controlled state.

`__builtin_shufflevector`

`__builtin_shufflevector` is used to express generic vector permutation/shuffle/swizzle operations. This builtin is also very important for the implementation of various target-specific header files like `<xmmintrin.h>`.

Syntax:

`__builtin_shufflevector(vec1, vec2, index1, index2, ...)`

Examples:

```
// identity operation - return 4-element vector v1.
__builtin_shufflevector(v1, v1, 0, 1, 2, 3)

// "Splat" element 0 of V1 into a 4-element result.
__builtin_shufflevector(V1, V1, 0, 0, 0, 0)

// Reverse 4-element vector V1.
__builtin_shufflevector(V1, V1, 3, 2, 1, 0)

// Concatenate every other element of 4-element vectors V1 and V2.
__builtin_shufflevector(V1, V2, 0, 2, 4, 6)

// Concatenate every other element of 8-element vectors V1 and V2.
__builtin_shufflevector(V1, V2, 0, 2, 4, 6, 8, 10, 12, 14)

// Shuffle v1 with some elements being undefined
__builtin_shufflevector(v1, v1, 3, -1, 1, -1)
```

Description:

The first two arguments to `__builtin_shufflevector` are vectors that have the same element type. The remaining arguments are a list of integers that specify the elements indices of the first two vectors that should be extracted and returned in a new vector. These element indices are numbered sequentially starting with the first vector, continuing into the second vector. Thus, if `vec1` is a 4-element vector, index 5 would refer to the second element of `vec2`. An index of -1 can be used to indicate that the corresponding element in the returned vector is a don't care and can be optimized by the backend.

The result of `__builtin_shufflevector` is a vector with the same element type as `vec1/vec2` but that has an element count equal to the number of indices specified.

Query for this feature with `__has_builtin(__builtin_shufflevector)`.

`__builtin_convertvector`

`__builtin_convertvector` is used to express generic vector type-conversion operations. The input vector and the output vector type must have the same number of elements.

Syntax:

```
__builtin_convertvector(src_vec, dst_vec_type)
```

Examples:

```
typedef double vector4double __attribute__((__vector_size__(32)));
typedef float vector4float __attribute__((__vector_size__(16)));
typedef short vector4short __attribute__((__vector_size__(8)));
vector4float vf; vector4short vs;

// convert from a vector of 4 floats to a vector of 4 doubles.
__builtin_convertvector(vf, vector4double)
// equivalent to:
(vector4double) { (double) vf[0], (double) vf[1], (double) vf[2], (double) vf[3] }

// convert from a vector of 4 shorts to a vector of 4 floats.
__builtin_convertvector(vs, vector4float)
// equivalent to:
(vector4float) { (float) vs[0], (float) vs[1], (float) vs[2], (float) vs[3] }
```

Description:

The first argument to `__builtin_convertvector` is a vector, and the second argument is a vector type with the same number of elements as the first argument.

The result of `__builtin_convertvector` is a vector with the same element type as the second argument, with a value defined in terms of the action of a C-style cast applied to each element of the first argument.

Query for this feature with `__has_builtin(__builtin_convertvector)`.

`__builtin_unreachable`

`__builtin_unreachable` is used to indicate that a specific point in the program cannot be reached, even if the compiler might otherwise think it can. This is useful to improve optimization and eliminates certain warnings. For example, without the `__builtin_unreachable` in the example below, the compiler assumes that the inline asm can fall through and prints a “function declared ‘noreturn’ should not return” warning.

Syntax:

```
__builtin_unreachable()
```

Example of use:

```
void myabort(void) __attribute__((noreturn));
void myabort(void) {
    asm("int3");
    __builtin_unreachable();
}
```

Description:

The `__builtin_unreachable()` builtin has completely undefined behavior. Since it has undefined behavior, it is a statement that it is never reached and the optimizer can take advantage of this to produce better code. This builtin takes no arguments and produces a void result.

Query for this feature with `__has_builtin(__builtin_unreachable)`.

`__sync_swap`

`__sync_swap` is used to atomically swap integers or pointers in memory.

Syntax:

```
type __sync_swap(type *ptr, type value, ...)
```

Example of Use:

```
int old_value = __sync_swap(&value, new_value);
```

Description:

The `__sync_swap()` builtin extends the existing `__sync_*()` family of atomic intrinsics to allow code to atomically swap the current value with the new value. More importantly, it helps developers write more efficient and correct code by avoiding expensive loops around `__sync_bool_compare_and_swap()` or relying on the platform specific implementation details of `__sync_lock_test_and_set()`. The `__sync_swap()` builtin is a full barrier.

`__builtin_addressof`

`__builtin_addressof` performs the functionality of the built-in `&` operator, ignoring any `operator&` overload. This is useful in constant expressions in C++11, where there is no other way to take the address of an object that overloads `operator&`.

Example of use:

```
template<typename T> constexpr T *addressof(T &value) {  
    return __builtin_addressof(value);  
}
```

`__builtin_operator_new` and `__builtin_operator_delete`

`__builtin_operator_new` allocates memory just like a non-placement non-class *new-expression*. This is exactly like directly calling the normal non-placement `::operator new`, except that it allows certain optimizations that the C++ standard does not permit for a direct function call to `::operator new` (in particular, removing `new / delete` pairs and merging allocations).

Likewise, `__builtin_operator_delete` deallocates memory just like a non-class *delete-expression*, and is exactly like directly calling the normal `::operator delete`, except that it permits optimizations. Only the unsized form of `__builtin_operator_delete` is currently available.

These builtins are intended for use in the implementation of `std::allocator` and other similar allocation libraries, and are only available in C++.

Multiprecision Arithmetic Builtins

Clang provides a set of builtins which expose multiprecision arithmetic in a manner amenable to C. They all have the following form:

```
unsigned x = ..., y = ..., carryin = ..., carryout;  
unsigned sum = __builtin_addc(x, y, carryin, &carryout);
```

Thus one can form a multiprecision addition chain in the following manner:


```

unsigned *x, *y, *z, carryin=0, carryout;
z[0] = __builtin_addc(x[0], y[0], carryin, &carryout);
carryin = carryout;
z[1] = __builtin_addc(x[1], y[1], carryin, &carryout);
carryin = carryout;
z[2] = __builtin_addc(x[2], y[2], carryin, &carryout);
carryin = carryout;
z[3] = __builtin_addc(x[3], y[3], carryin, &carryout);

```

The complete list of builtins are:

```

unsigned char    __builtin_addcb (unsigned char x, unsigned char y, unsigned char
↳carryin, unsigned char *carryout);
unsigned short  __builtin_addcs (unsigned short x, unsigned short y, unsigned
↳short carryin, unsigned short *carryout);
unsigned         __builtin_addc  (unsigned x, unsigned y, unsigned carryin,
↳unsigned *carryout);
unsigned long   __builtin_addcl (unsigned long x, unsigned long y, unsigned long
↳carryin, unsigned long *carryout);
unsigned long long __builtin_addcll(unsigned long long x, unsigned long long y,
↳unsigned long long carryin, unsigned long long *carryout);
unsigned char    __builtin_subcb (unsigned char x, unsigned char y, unsigned char
↳carryin, unsigned char *carryout);
unsigned short  __builtin_subcs (unsigned short x, unsigned short y, unsigned
↳short carryin, unsigned short *carryout);
unsigned         __builtin_subc  (unsigned x, unsigned y, unsigned carryin,
↳unsigned *carryout);
unsigned long   __builtin_subcl (unsigned long x, unsigned long y, unsigned long
↳carryin, unsigned long *carryout);
unsigned long long __builtin_subcll(unsigned long long x, unsigned long long y,
↳unsigned long long carryin, unsigned long long *carryout);

```

Checked Arithmetic Builtins

Clang provides a set of builtins that implement checked arithmetic for security critical applications in a manner that is fast and easily expressible in C. As an example of their usage:

```

errorcode_t security_critical_application(...) {
    unsigned x, y, result;
    ...
    if (__builtin_umul_overflow(x, y, &result))
        return kErrorCodeHackers;
    ...
    use_multiply(result);
    ...
}

```

A complete enumeration of the builtins are:

```

bool __builtin_uadd_overflow (unsigned x, unsigned y, unsigned *sum);
bool __builtin_uaddl_overflow (unsigned long x, unsigned long y, unsigned long *sum);
bool __builtin_uaddll_overflow(unsigned long long x, unsigned long long y, unsigned
↳long long *sum);
bool __builtin_usub_overflow  (unsigned x, unsigned y, unsigned *diff);
bool __builtin_usubl_overflow (unsigned long x, unsigned long y, unsigned long *diff);
bool __builtin_usubll_overflow(unsigned long long x, unsigned long long y, unsigned
↳long long *diff);

```

```
bool __builtin_umul_overflow (unsigned x, unsigned y, unsigned *prod);
bool __builtin_umull_overflow (unsigned long x, unsigned long y, unsigned long *prod);
bool __builtin_umulll_overflow (unsigned long long x, unsigned long long y, unsigned_
↳long long *prod);
bool __builtin_sadd_overflow (int x, int y, int *sum);
bool __builtin_saddl_overflow (long x, long y, long *sum);
bool __builtin_saddll_overflow (long long x, long long y, long long *sum);
bool __builtin_ssub_overflow (int x, int y, int *diff);
bool __builtin_ssubl_overflow (long x, long y, long *diff);
bool __builtin_ssubll_overflow (long long x, long long y, long long *diff);
bool __builtin_smul_overflow (int x, int y, int *prod);
bool __builtin_smull_overflow (long x, long y, long *prod);
bool __builtin_smulll_overflow (long long x, long long y, long long *prod);
```

__c11_atomic builtins

Clang provides a set of builtins which are intended to be used to implement C11's `<stdatomic.h>` header. These builtins provide the semantics of the `__explicit` form of the corresponding C11 operation, and are named with a `__c11_` prefix. The supported operations, and the differences from the corresponding C11 operations, are:

- `__c11_atomic_init`
- `__c11_atomic_thread_fence`
- `__c11_atomic_signal_fence`
- `__c11_atomic_is_lock_free` (The argument is the size of the `_Atomic(...)` object, instead of its address)
- `__c11_atomic_store`
- `__c11_atomic_load`
- `__c11_atomic_exchange`
- `__c11_atomic_compare_exchange_strong`
- `__c11_atomic_compare_exchange_weak`
- `__c11_atomic_fetch_add`
- `__c11_atomic_fetch_sub`
- `__c11_atomic_fetch_and`
- `__c11_atomic_fetch_or`
- `__c11_atomic_fetch_xor`

The macros `__ATOMIC_RELAXED`, `__ATOMIC_CONSUME`, `__ATOMIC_ACQUIRE`, `__ATOMIC_RELEASE`, `__ATOMIC_ACQ_REL`, and `__ATOMIC_SEQ_CST` are provided, with values corresponding to the enumerators of C11's `memory_order` enumeration.

Low-level ARM exclusive memory builtins

Clang provides overloaded builtins giving direct access to the three key ARM instructions for implementing atomic operations.

```
T __builtin_arm_ldrex(const volatile T *addr);
T __builtin_arm_ldaex(const volatile T *addr);
int __builtin_arm_strex(T val, volatile T *addr);
int __builtin_arm_stlex(T val, volatile T *addr);
void __builtin_arm_clrex(void);
```

The types `T` currently supported are: * Integer types with width at most 64 bits (or 128 bits on AArch64). * Floating-point types * Pointer types.

Note that the compiler does not guarantee it will not insert stores which clear the exclusive monitor in between an `ldrex` type operation and its paired `strex`. In practice this is only usually a risk when the extra store is on the same cache line as the variable being modified and Clang will only insert stack stores on its own, so it is best not to use these operations on variables with automatic storage duration.

Also, loads and stores may be implicit in code written between the `ldrex` and `strex`. Clang will not necessarily mitigate the effects of these either, so care should be exercised.

For these reasons the higher level atomic primitives should be preferred where possible.

Non-standard C++11 Attributes

Clang's non-standard C++11 attributes live in the `clang` attribute namespace.

Clang supports GCC's `gnu` attribute namespace. All GCC attributes which are accepted with the `__attribute__((foo))` syntax are also accepted as `[[gnu::foo]]`. This only extends to attributes which are specified by GCC (see the list of [GCC function attributes](#), [GCC variable attributes](#), and [GCC type attributes](#)). As with the GCC implementation, these attributes must appertain to the *declarator-id* in a declaration, which means they must go either at the start of the declaration or immediately after the name being declared.

For example, this applies the GNU `unused` attribute to `a` and `f`, and also applies the GNU `noreturn` attribute to `f`.

```
[[gnu::unused]] int a, f [[gnu::noreturn]] ();
```

Target-Specific Extensions

Clang supports some language features conditionally on some targets.

ARM/AArch64 Language Extensions

Memory Barrier Ininsics

Clang implements the `__dmb`, `__dsb` and `__isb` intrinsics as defined in the [ARM C Language Extensions Release 2.0](#). Note that these intrinsics are implemented as motion barriers that block reordering of memory accesses and side effect instructions. Other instructions like simple arithmetic may be reordered around the intrinsic. If you expect to have no reordering at all, use inline assembly instead.

X86/X86-64 Language Extensions

The X86 backend has these language extensions:

Memory references off the GS segment

Annotating a pointer with address space #256 causes it to be code generated relative to the X86 GS segment register, and address space #257 causes it to be relative to the X86 FS segment. Note that this is a very very low-level feature that should only be used if you know what you're doing (for example in an OS kernel).

Here is an example:

```
#define GS_RELATIVE __attribute__((address_space(256)))
int foo(int GS_RELATIVE *P) {
    return *P;
}
```

Which compiles to (on X86-32):

```
_foo:
    movl    4(%esp), %eax
    movl    %gs:(%eax), %eax
    ret
```

Extensions for Static Analysis

Clang supports additional attributes that are useful for documenting program invariants and rules for static analysis tools, such as the [Clang Static Analyzer](#). These attributes are documented in the analyzer's [list of source-level annotations](#).

Extensions for Dynamic Analysis

Use `__has_feature(address_sanitizer)` to check if the code is being built with *AddressSanitizer*.

Use `__has_feature(thread_sanitizer)` to check if the code is being built with *ThreadSanitizer*.

Use `__has_feature(memory_sanitizer)` to check if the code is being built with *MemorySanitizer*.

Use `__has_feature(safe_stack)` to check if the code is being built with *SafeStack*.

Extensions for selectively disabling optimization

Clang provides a mechanism for selectively disabling optimizations in functions and methods.

To disable optimizations in a single function definition, the GNU-style or C++11 non-standard attribute `optnone` can be used.

```
// The following functions will not be optimized.
// GNU-style attribute
__attribute__((optnone)) int foo() {
    // ... code
}
// C++11 attribute
[[clang::optnone]] int bar() {
    // ... code
}
```

To facilitate disabling optimization for a range of function definitions, a range-based pragma is provided. Its syntax is `#pragma clang optimize` followed by `off` or `on`.

All function definitions in the region between an `off` and the following `on` will be decorated with the `optnone` attribute unless doing so would conflict with explicit attributes already present on the function (e.g. the ones that control inlining).

```
#pragma clang optimize off
// This function will be decorated with optnone.
int foo() {
    // ... code
}

// optnone conflicts with always_inline, so bar() will not be decorated.
__attribute__((always_inline)) int bar() {
    // ... code
}
#pragma clang optimize on
```

If no `on` is found to close an `off` region, the end of the region is the end of the compilation unit.

Note that a stray `#pragma clang optimize on` does not selectively enable additional optimizations when compiling at low optimization levels. This feature can only be used to selectively disable optimizations.

The pragma has an effect on functions only at the point of their definition; for function templates, this means that the state of the pragma at the point of an instantiation is not necessarily relevant. Consider the following example:

```
template<typename T> T twice(T t) {
    return 2 * t;
}

#pragma clang optimize off
template<typename T> T thrice(T t) {
    return 3 * t;
}

int container(int a, int b) {
    return twice(a) + thrice(b);
}
#pragma clang optimize on
```

In this example, the definition of the template function `twice` is outside the pragma region, whereas the definition of `thrice` is inside the region. The `container` function is also in the region and will not be optimized, but it causes the instantiation of `twice` and `thrice` with an `int` type; of these two instantiations, `twice` will be optimized (because its definition was outside the region) and `thrice` will not be optimized.

Extensions for loop hint optimizations

The `#pragma clang loop` directive is used to specify hints for optimizing the subsequent `for`, `while`, `do-while`, or `c++11 range-based for` loop. The directive provides options for vectorization, interleaving, and unrolling. Loop hints can be specified before any loop and will be ignored if the optimization is not safe to apply.

Vectorization and Interleaving

A vectorized loop performs multiple iterations of the original loop in parallel using vector instructions. The instruction set of the target processor determines which vector instructions are available and their vector widths. This restricts

the types of loops that can be vectorized. The vectorizer automatically determines if the loop is safe and profitable to vectorize. A vector instruction cost model is used to select the vector width.

Interleaving multiple loop iterations allows modern processors to further improve instruction-level parallelism (ILP) using advanced hardware features, such as multiple execution units and out-of-order execution. The vectorizer uses a cost model that depends on the register pressure and generated code size to select the interleaving count.

Vectorization is enabled by `vectorize(enable)` and interleaving is enabled by `interleave(enable)`. This is useful when compiling with `-Os` to manually enable vectorization or interleaving.

```
#pragma clang loop vectorize(enable)
#pragma clang loop interleave(enable)
for(...) {
    ...
}
```

The vector width is specified by `vectorize_width(_value_)` and the interleave count is specified by `interleave_count(_value_)`, where `_value_` is a positive integer. This is useful for specifying the optimal width/count of the set of target architectures supported by your application.

```
#pragma clang loop vectorize_width(2)
#pragma clang loop interleave_count(2)
for(...) {
    ...
}
```

Specifying a width/count of 1 disables the optimization, and is equivalent to `vectorize(disable)` or `interleave(disable)`.

Loop Unrolling

Unrolling a loop reduces the loop control overhead and exposes more opportunities for ILP. Loops can be fully or partially unrolled. Full unrolling eliminates the loop and replaces it with an enumerated sequence of loop iterations. Full unrolling is only possible if the loop trip count is known at compile time. Partial unrolling replicates the loop body within the loop and reduces the trip count.

If `unroll(full)` is specified the unroller will attempt to fully unroll the loop if the trip count is known at compile time. If the fully unrolled code size is greater than an internal limit the loop will be partially unrolled up to this limit. If the loop count is not known at compile time the loop will not be unrolled.

```
#pragma clang loop unroll(full)
for(...) {
    ...
}
```

The unroll count can be specified explicitly with `unroll_count(_value_)` where `_value_` is a positive integer. If this value is greater than the trip count the loop will be fully unrolled. Otherwise the loop is partially unrolled subject to the same code size limit as with `unroll(full)`.

```
#pragma clang loop unroll_count(8)
for(...) {
    ...
}
```

Unrolling of a loop can be prevented by specifying `unroll(disable)`.

Additional Information

For convenience multiple loop hints can be specified on a single line.

```
#pragma clang loop vectorize_width(4) interleave_count(8)
for (...) {
    ...
}
```

If an optimization cannot be applied any hints that apply to it will be ignored. For example, the hint `vectorize_width(4)` is ignored if the loop is not proven safe to vectorize. To identify and diagnose optimization issues use `-Rpass`, `-Rpass-missed`, and `-Rpass-analysis` command line options. See the user guide for details.

Attributes in Clang

- *Introduction*
- *AMD GPU Register Attributes*
 - *amdgpu_num_sgpr*
 - *amdgpu_num_vgpr*
- *Function Attributes*
 - *interrupt*
 - *acquire_capability* (*acquire_shared_capability*, *clang::acquire_capability*, *clang::acquire_shared_capability*)
 - *assert_capability* (*assert_shared_capability*, *clang::assert_capability*, *clang::assert_shared_capability*)
 - *assume_aligned* (*gnu::assume_aligned*)
 - *availability*
 - *_Noreturn*
 - *noreturn*
 - *carries_dependency*
 - *enable_if*
 - *flatten* (*gnu::flatten*)
 - *format* (*gnu::format*)
 - *noduplicate* (*clang::noduplicate*)
 - *no_sanitize* (*clang::no_sanitize*)
 - *no_sanitize_address* (*no_address_safety_analysis*, *gnu::no_address_safety_analysis*, *gnu::no_sanitize_address*)
 - *no_sanitize_thread*
 - *no_sanitize_memory*
 - *no_split_stack* (*gnu::no_split_stack*)

- *objc_boxable*
- *objc_method_family*
- *objc_requires_super*
- *objc_runtime_name*
- *optnone* (*clang::optnone*)
- *overloadable*
- *release_capability* (*release_shared_capability*, *clang::release_capability*, *clang::release_shared_capability*)
- *target* (*gnu::target*)
- *try_acquire_capability* (*try_acquire_shared_capability*, *clang::try_acquire_capability*, *clang::try_acquire_shared_capability*)
- *Variable Attributes*
 - *init_seg*
 - *section* (*gnu::section*, *__declspec(allocate)*)
 - *tls_model* (*gnu::tls_model*)
 - *thread*
- *Type Attributes*
 - *align_value*
 - *flag_enum*
 - *__single_inheritance*, *__multiple_inheritance*, *__virtual_inheritance*
 - *novtable*
- *Statement Attributes*
 - *fallthrough* (*clang::fallthrough*)
 - *#pragma clang loop*
 - *#pragma unroll*, *#pragma nounroll*
- *Calling Conventions*
 - *fastcall* (*gnu::fastcall*, *__fastcall*, *_fastcall*)
 - *ms_abi* (*gnu::ms_abi*)
 - *pcs* (*gnu::pcs*)
 - *regparm* (*gnu::regparm*)
 - *stdcall* (*gnu::stdcall*, *__stdcall*, *_stdcall*)
 - *thiscall* (*gnu::thiscall*, *__thiscall*, *_thiscall*)
 - *vectorcall* (*__vectorcall*, *_vectorcall*)
- *Consumed Annotation Checking*
 - *callable_when*
 - *consumable*

- *param_tystate*
- *return_tystate*
- *set_tystate*
- *test_tystate*
- *Type Safety Checking*
 - *argument_with_type_tag*
 - *pointer_with_type_tag*
 - *type_tag_for_datatype*
- *OpenCL Address Spaces*
 - *__constant(constant)*
 - *__generic(generic)*
 - *__global(global)*
 - *__local(local)*
 - *__private(private)*
- *Nullability Attributes*
 - *nonnull*
 - *returns_nonnull*
 - *_Nonnull*
 - *_Null_unspecified*
 - *_Nullable*

Introduction

This page lists the attributes currently supported by Clang.

AMD GPU Register Attributes

Clang supports attributes for controlling register usage on AMD GPU targets. These attributes may be attached to a kernel function definition and is an optimization hint to the backend for the maximum number of registers to use. This is useful in cases where register limited occupancy is known to be an important factor for the performance for the kernel.

The semantics are as follows:

- The backend will attempt to limit the number of used registers to the specified value, but the exact number used is not guaranteed. The number used may be rounded up to satisfy the allocation requirements or ABI constraints of the subtarget. For example, on Southern Islands VGPRs may only be allocated in increments of 4, so requesting a limit of 39 VGPRs will really attempt to use up to 40. Requesting more registers than the subtarget supports will truncate to the maximum allowed. The backend may also use fewer registers than requested whenever possible.
- 0 implies the default no limit on register usage.

- Ignored on older VLIW subtargets which did not have separate scalar and vector registers, R600 through Northern Islands.

amdgpu_num_sgpr

Table 4.1: Supported Syntaxes

GNU	C++11	__declspec	Keyword	Pragma
X				

Clang supports the `__attribute__((amdgpu_num_sgpr(<num_registers>)))` attribute on AMD Southern Islands GPUs and later for controlling the number of scalar registers. A typical value would be between 8 and 104 in increments of 8.

Due to common instruction constraints, an additional 2-4 SGPRs are typically required for internal use depending on features used. This value is a hint for the total number of SGPRs to use, and not the number of user SGPRs, so no special consideration needs to be given for these.

amdgpu_num_vgpr

Table 4.2: Supported Syntaxes

GNU	C++11	__declspec	Keyword	Pragma
X				

Clang supports the `__attribute__((amdgpu_num_vgpr(<num_registers>)))` attribute on AMD Southern Islands GPUs and later for controlling the number of vector registers. A typical value would be between 4 and 256 in increments of 4.

Function Attributes

interrupt

Table 4.3: Supported Syntaxes

GNU	C++11	__declspec	Keyword	Pragma
X				

Clang supports the GNU style `__attribute__((interrupt("TYPE")))` attribute on ARM targets. This attribute may be attached to a function definition and instructs the backend to generate appropriate function entry/exit code so that it can be used directly as an interrupt service routine.

The parameter passed to the interrupt attribute is optional, but if provided it must be a string literal with one of the following values: “IRQ”, “FIQ”, “SWI”, “ABORT”, “UNDEF”.

The semantics are as follows:

- If the function is AAPCS, Clang instructs the backend to realign the stack to 8 bytes on entry. This is a general requirement of the AAPCS at public interfaces, but may not hold when an exception is taken. Doing this allows other AAPCS functions to be called.
- If the CPU is M-class this is all that needs to be done since the architecture itself is designed in such a way that functions obeying the normal AAPCS ABI constraints are valid exception handlers.

- If the CPU is not M-class, the prologue and epilogue are modified to save all non-banked registers that are used, so that upon return the user-mode state will not be corrupted. Note that to avoid unnecessary overhead, only general-purpose (integer) registers are saved in this way. If VFP operations are needed, that state must be saved manually.

Specifically, interrupt kinds other than “FIQ” will save all core registers except “lr” and “sp”. “FIQ” interrupts will save r0-r7.

- If the CPU is not M-class, the return instruction is changed to one of the canonical sequences permitted by the architecture for exception return. Where possible the function itself will make the necessary “lr” adjustments so that the “preferred return address” is selected.

Unfortunately the compiler is unable to make this guarantee for an “UNDEF” handler, where the offset from “lr” to the preferred return address depends on the execution state of the code which generated the exception. In this case a sequence equivalent to “movs pc, lr” will be used.

acquire_capability (acquire_shared_capability, clang::acquire_capability, clang::acquire_shared_capability)

Table 4.4: Supported Syntaxes

GNU	C++11	__declspec	Keyword	Pragma
X	X			

Marks a function as acquiring a capability.

assert_capability (assert_shared_capability, clang::assert_capability, clang::assert_shared_capability)

Table 4.5: Supported Syntaxes

GNU	C++11	__declspec	Keyword	Pragma
X	X			

Marks a function that dynamically tests whether a capability is held, and halts the program if it is not held.

assume_aligned (gnu::assume_aligned)

Table 4.6: Supported Syntaxes

GNU	C++11	__declspec	Keyword	Pragma
X	X			

Use `__attribute__((assume_aligned(<alignment>[,<offset>])))` on a function declaration to specify that the return value of the function (which must be a pointer type) has the specified offset, in bytes, from an address with the specified alignment. The offset is taken to be zero if omitted.

```
// The returned pointer value has 32-byte alignment.
void *a() __attribute__((assume_aligned (32)));

// The returned pointer value is 4 bytes greater than an address having
// 32-byte alignment.
void *b() __attribute__((assume_aligned (32, 4)));
```

Note that this attribute provides information to the compiler regarding a condition that the code already ensures is true. It does not cause the compiler to enforce the provided alignment assumption.

availability

Table 4.7: Supported Syntaxes

GNU	C++11	__declspec	Keyword	Pragma
X				

The `availability` attribute can be placed on declarations to describe the lifecycle of that declaration relative to operating system versions. Consider the function declaration for a hypothetical function `f`:

```
void f(void) __attribute__((availability(macosx,introduced=10.4,deprecated=10.6,  
↪obsoleted=10.7)));
```

The `availability` attribute states that `f` was introduced in Mac OS X 10.4, deprecated in Mac OS X 10.6, and obsoleted in Mac OS X 10.7. This information is used by Clang to determine when it is safe to use `f`: for example, if Clang is instructed to compile code for Mac OS X 10.5, a call to `f()` succeeds. If Clang is instructed to compile code for Mac OS X 10.6, the call succeeds but Clang emits a warning specifying that the function is deprecated. Finally, if Clang is instructed to compile code for Mac OS X 10.7, the call fails because `f()` is no longer available.

The `availability` attribute is a comma-separated list starting with the platform name and then including clauses specifying important milestones in the declaration's lifetime (in any order) along with additional information. Those clauses can be:

introduced=version The first version in which this declaration was introduced.

deprecated=version The first version in which this declaration was deprecated, meaning that users should migrate away from this API.

obsoleted=version The first version in which this declaration was obsoleted, meaning that it was removed completely and can no longer be used.

unavailable This declaration is never available on this platform.

message=string-literal Additional message text that Clang will provide when emitting a warning or error about use of a deprecated or obsoleted declaration. Useful to direct users to replacement APIs.

Multiple `availability` attributes can be placed on a declaration, which may correspond to different platforms. Only the `availability` attribute with the platform corresponding to the target platform will be used; any others will be ignored. If no `availability` attribute specifies availability for the current target platform, the `availability` attributes are ignored. Supported platforms are:

ios Apple's iOS operating system. The minimum deployment target is specified by the `-mios-version-min=*version*` or `-miphoneos-version-min=*version*` command-line arguments.

macosx Apple's Mac OS X operating system. The minimum deployment target is specified by the `-mmacosx-version-min=*version*` command-line argument.

A declaration can be used even when deploying back to a platform version prior to when the declaration was introduced. When this happens, the declaration is **weakly linked**, as if the `weak_import` attribute were added to the declaration. A weakly-linked declaration may or may not be present a run-time, and a program can determine whether the declaration is present by checking whether the address of that declaration is non-NULL.

If there are multiple declarations of the same entity, the `availability` attributes must either match on a per-platform basis or later declarations must not have `availability` attributes for that platform. For example:

```

void g(void) __attribute__((availability(macosx,introduced=10.4)));
void g(void) __attribute__((availability(macosx,introduced=10.4))); // okay, matches
void g(void) __attribute__((availability(ios,introduced=4.0))); // okay, adds a new_
↳platform
void g(void); // okay, inherits both macosx and ios availability from above.
void g(void) __attribute__((availability(macosx,introduced=10.5))); // error: mismatch

```

When one method overrides another, the overriding method can be more widely available than the overridden method, e.g.,:

```

@interface A
- (id)method __attribute__((availability(macosx,introduced=10.4)));
- (id)method2 __attribute__((availability(macosx,introduced=10.4)));
@end

@interface B : A
- (id)method __attribute__((availability(macosx,introduced=10.3))); // okay: method_
↳moved into base class later
- (id)method __attribute__((availability(macosx,introduced=10.5))); // error: this_
↳method was available via the base class in 10.4
@end

```

_Noreturn

Table 4.8: Supported Syntaxes

GNU	C++11	__declspec	Keyword	Pragma
			X	

A function declared as `_Noreturn` shall not return to its caller. The compiler will generate a diagnostic for a function declared as `_Noreturn` that appears to be capable of returning to its caller.

noreturn

Table 4.9: Supported Syntaxes

GNU	C++11	__declspec	Keyword	Pragma
	X			

A function declared as `[[noreturn]]` shall not return to its caller. The compiler will generate a diagnostic for a function declared as `[[noreturn]]` that appears to be capable of returning to its caller.

carries_dependency

Table 4.10: Supported Syntaxes

GNU	C++11	__declspec	Keyword	Pragma
X	X			

The `carries_dependency` attribute specifies dependency propagation into and out of functions.

When specified on a function or Objective-C method, the `carries_dependency` attribute means that the return value carries a dependency out of the function, so that the implementation need not constrain ordering upon return

from that function. Implementations of the function and its caller may choose to preserve dependencies instead of emitting memory ordering instructions such as fences.

Note, this attribute does not change the meaning of the program, but may result in generation of more efficient code.

enable_if

Table 4.11: Supported Syntaxes

GNU	C++11	__declspec	Keyword	Pragma
X				

Note: Some features of this attribute are experimental. The meaning of multiple `enable_if` attributes on a single declaration is subject to change in a future version of clang. Also, the ABI is not standardized and the name mangling may change in future versions. To avoid that, use asm labels.

The `enable_if` attribute can be placed on function declarations to control which overload is selected based on the values of the function's arguments. When combined with the `overloadable` attribute, this feature is also available in C.

```
int isdigit(int c);
int isdigit(int c) __attribute__((enable_if(c <= -1 || c > 255, "chosen when 'c' is_
↳out of range"))) __attribute__((unavailable("'c' must have the value of an unsigned_
↳char or EOF")));

void foo(char c) {
    isdigit(c);
    isdigit(10);
    isdigit(-10); // results in a compile-time error.
}
```

The `enable_if` attribute takes two arguments, the first is an expression written in terms of the function parameters, the second is a string explaining why this overload candidate could not be selected to be displayed in diagnostics. The expression is part of the function signature for the purposes of determining whether it is a redeclaration (following the rules used when determining whether a C++ template specialization is ODR-equivalent), but is not part of the type.

The `enable_if` expression is evaluated as if it were the body of a bool-returning constexpr function declared with the arguments of the function it is being applied to, then called with the parameters at the call site. If the result is false or could not be determined through constant expression evaluation, then this overload will not be chosen and the provided string may be used in a diagnostic if the compile fails as a result.

Because the `enable_if` expression is an unevaluated context, there are no global state changes, nor the ability to pass information from the `enable_if` expression to the function body. For example, suppose we want calls to `strlen(strbuf, maxlen)` to resolve to `strlen_chk(strbuf, maxlen, size of strbuf)` only if the size of `strbuf` can be determined:

```
__attribute__((always_inline))
static inline size_t strlen(const char *s, size_t maxlen)
    __attribute__((overloadable))
    __attribute__((enable_if(__builtin_object_size(s, 0) != -1))),
    "chosen when the buffer size is known but 'maxlen' is not
↳"))
{
    return strlen_chk(s, maxlen, __builtin_object_size(s, 0));
}
```

Multiple `enable_if` attributes may be applied to a single declaration. In this case, the `enable_if` expressions are evaluated from left to right in the following manner. First, the candidates whose `enable_if` expressions evaluate to false or cannot be evaluated are discarded. If the remaining candidates do not share ODR-equivalent `enable_if` expressions, the overload resolution is ambiguous. Otherwise, `enable_if` overload resolution continues with the next `enable_if` attribute on the candidates that have not been discarded and have remaining `enable_if` attributes. In this way, we pick the most specific overload out of a number of viable overloads using `enable_if`.

```
void f() __attribute__((enable_if(true, ""))); // #1
void f() __attribute__((enable_if(true, ""))) __attribute__((enable_if(true, ""))); /
↪ / #2

void g(int i, int j) __attribute__((enable_if(i, ""))); // #1
void g(int i, int j) __attribute__((enable_if(j, ""))) __attribute__((enable_
↪ if(true))); // #2
```

In this example, a call to `f()` is always resolved to #2, as the first `enable_if` expression is ODR-equivalent for both declarations, but #1 does not have another `enable_if` expression to continue evaluating, so the next round of evaluation has only a single candidate. In a call to `g(1, 1)`, the call is ambiguous even though #2 has more `enable_if` attributes, because the first `enable_if` expressions are not ODR-equivalent.

Query for this feature with `__has_attribute(enable_if)`.

flatten (gnu::flatten)

Table 4.12: Supported Syntaxes

GNU	C++11	__declspec	Keyword	Pragma
X	X			

The `flatten` attribute causes calls within the attributed function to be inlined unless it is impossible to do so, for example if the body of the callee is unavailable or if the callee has the `noinline` attribute.

format (gnu::format)

Table 4.13: Supported Syntaxes

GNU	C++11	__declspec	Keyword	Pragma
X	X			

Clang supports the `format` attribute, which indicates that the function accepts a `printf` or `scanf`-like format string and corresponding arguments or a `va_list` that contains these arguments.

Please see [GCC documentation about format attribute](#) to find details about attribute syntax.

Clang implements two kinds of checks with this attribute.

1. Clang checks that the function with the `format` attribute is called with a format string that uses format specifiers that are allowed, and that arguments match the format string. This is the `-Wformat` warning, it is on by default.
2. Clang checks that the format string argument is a literal string. This is the `-Wformat-nonliteral` warning, it is off by default.

Clang implements this mostly the same way as GCC, but there is a difference for functions that accept a `va_list` argument (for example, `vprintf`). GCC does not emit `-Wformat-nonliteral` warning for calls to such functions. Clang does not warn if the format string comes from a function parameter, where the function is annotated with a compatible attribute, otherwise it warns. For example:

```
__attribute__((__format__(__scanf__, 1, 3)))
void foo(const char* s, char *buf, ...) {
    va_list ap;
    va_start(ap, buf);

    vprintf(s, ap); // warning: format string is not a string literal
}
```

In this case we warn because `s` contains a format string for a `scanf`-like function, but it is passed to a `printf`-like function.

If the attribute is removed, clang still warns, because the format string is not a string literal.

Another example:

```
__attribute__((__format__(__printf__, 1, 3)))
void foo(const char* s, char *buf, ...) {
    va_list ap;
    va_start(ap, buf);

    vprintf(s, ap); // warning
}
```

In this case Clang does not warn because the format string `s` and the corresponding arguments are annotated. If the arguments are incorrect, the caller of `foo` will receive a warning.

noduplicate (clang::noduplicate)

Table 4.14: Supported Syntaxes

GNU	C++11	__declspec	Keyword	Pragma
X	X			

The `noduplicate` attribute can be placed on function declarations to control whether function calls to this function can be duplicated or not as a result of optimizations. This is required for the implementation of functions with certain special requirements, like the OpenCL “barrier” function, that might need to be run concurrently by all the threads that are executing in lockstep on the hardware. For example this attribute applied on the function “`nodupfunc`” in the code below avoids that:

```
void nodupfunc() __attribute__((noduplicate));
// Setting it as a C++11 attribute is also valid
// void nodupfunc() [[clang::noduplicate]];
void foo();
void bar();

nodupfunc();
if (a > n) {
    foo();
} else {
    bar();
}
```

gets possibly modified by some optimizations into code similar to this:

```
if (a > n) {
    nodupfunc();
}
```



```

    foo();
} else {
    nodupfunc();
    bar();
}

```

where the call to “nodupfunc” is duplicated and sunk into the two branches of the condition.

no_sanitize (clang::no_sanitize)

Table 4.15: Supported Syntaxes

GNU	C++11	__declspec	Keyword	Pragma
X	X			

Use the `no_sanitize` attribute on a function declaration to specify that a particular instrumentation or set of instrumentations should not be applied to that function. The attribute takes a list of string literals, which have the same meaning as values accepted by the `-fno-sanitize=` flag. For example, `__attribute__((no_sanitize("address", "thread")))` specifies that AddressSanitizer and ThreadSanitizer should not be applied to the function.

See *Controlling Code Generation* for a full list of supported sanitizer flags.

no_sanitize_address (no_address_safety_analysis, gnu::no_address_safety_analysis, gnu::no_sanitize_address)

Table 4.16: Supported Syntaxes

GNU	C++11	__declspec	Keyword	Pragma
X	X			

Use `__attribute__((no_sanitize_address))` on a function declaration to specify that address safety instrumentation (e.g. AddressSanitizer) should not be applied to that function.

no_sanitize_thread

Table 4.17: Supported Syntaxes

GNU	C++11	__declspec	Keyword	Pragma
X	X			

Use `__attribute__((no_sanitize_thread))` on a function declaration to specify that checks for data races on plain (non-atomic) memory accesses should not be inserted by ThreadSanitizer. The function is still instrumented by the tool to avoid false positives and provide meaningful stack traces.

no_sanitize_memory

Table 4.18: Supported Syntaxes

GNU	C++11	__declspec	Keyword	Pragma
X	X			

Use `__attribute__((no_sanitize_memory))` on a function declaration to specify that checks for uninitialized memory should not be inserted (e.g. by MemorySanitizer). The function may still be instrumented by the tool to avoid false positives in other places.

`no_split_stack (gnu::no_split_stack)`

Table 4.19: Supported Syntaxes

GNU	C++11	__declspec	Keyword	Pragma
X	X			

The `no_split_stack` attribute disables the emission of the split stack preamble for a particular function. It has no effect if `-fsplit-stack` is not specified.

`objc_boxable`

Table 4.20: Supported Syntaxes

GNU	C++11	__declspec	Keyword	Pragma
X				

Structs and unions marked with the `objc_boxable` attribute can be used with the Objective-C boxed expression syntax, `@(...)`.

Usage: `__attribute__((objc_boxable))`. This attribute can only be placed on a declaration of a trivially-copyable struct or union:

```
struct __attribute__((objc_boxable)) some_struct {
    int i;
};
union __attribute__((objc_boxable)) some_union {
    int i;
    float f;
};
typedef struct __attribute__((objc_boxable)) _some_struct some_struct;

// ...

some_struct ss;
NSValue *boxed = @(ss);
```

`objc_method_family`

Table 4.21: Supported Syntaxes

GNU	C++11	__declspec	Keyword	Pragma
X				

Many methods in Objective-C have conventional meanings determined by their selectors. It is sometimes useful to be able to mark a method as having a particular conventional meaning despite not having the right selector, or as not having the conventional meaning that its selector would suggest. For these use cases, we provide an attribute to specifically describe the “method family” that a method belongs to.

Usage: `__attribute__((objc_method_family(X)))`, where X is one of `none`, `alloc`, `copy`, `init`, `mutableCopy`, or `new`. This attribute can only be placed at the end of a method declaration:

```
- (NSString *)initMyStringValue __attribute__((objc_method_family(none)));
```

Users who do not wish to change the conventional meaning of a method, and who merely want to document its non-standard retain and release semantics, should use the retaining behavior attributes (`ns_returns_retained`, `ns_returns_not_retained`, etc).

Query for this feature with `__has_attribute(objc_method_family)`.

objc_requires_super

Table 4.22: Supported Syntaxes

GNU	C++11	__declspec	Keyword	Pragma
X				

Some Objective-C classes allow a subclass to override a particular method in a parent class but expect that the overriding method also calls the overridden method in the parent class. For these cases, we provide an attribute to designate that a method requires a “call to super” in the overriding method in the subclass.

Usage: `__attribute__((objc_requires_super))`. This attribute can only be placed at the end of a method declaration:

```
- (void)foo __attribute__((objc_requires_super));
```

This attribute can only be applied the method declarations within a class, and not a protocol. Currently this attribute does not enforce any placement of where the call occurs in the overriding method (such as in the case of `-dealloc` where the call must appear at the end). It checks only that it exists.

Note that on both OS X and iOS that the Foundation framework provides a convenience macro `NS_REQUIRES_SUPER` that provides syntactic sugar for this attribute:

```
- (void)foo NS_REQUIRES_SUPER;
```

This macro is conditionally defined depending on the compiler’s support for this attribute. If the compiler does not support the attribute the macro expands to nothing.

Operationally, when a method has this annotation the compiler will warn if the implementation of an override in a subclass does not call super. For example:

```
warning: method possibly missing a [super AnnotMeth] call
- (void) AnnotMeth{;;
    ^
```

objc_runtime_name

Table 4.23: Supported Syntaxes

GNU	C++11	__declspec	Keyword	Pragma
X				

By default, the Objective-C interface or protocol identifier is used in the metadata name for that object. The `objc_runtime_name` attribute allows annotated interfaces or protocols to use the specified string argument in the object’s metadata name instead of the default name.

Usage: `__attribute__((objc_runtime_name("MyLocalName")))`. This attribute can only be placed before an `@protocol` or `@interface` declaration:

```
__attribute__((objc_runtime_name("MyLocalName")))
@interface Message
@end
```

optnone (clang::optnone)

Table 4.24: Supported Syntaxes

GNU	C++11	__declspec	Keyword	Pragma
X	X			

The `optnone` attribute suppresses essentially all optimizations on a function or method, regardless of the optimization level applied to the compilation unit as a whole. This is particularly useful when you need to debug a particular function, but it is infeasible to build the entire application without optimization. Avoiding optimization on the specified function can improve the quality of the debugging information for that function.

This attribute is incompatible with the `always_inline` and `minsize` attributes.

overloadable

Table 4.25: Supported Syntaxes

GNU	C++11	__declspec	Keyword	Pragma
X				

Clang provides support for C++ function overloading in C. Function overloading in C is introduced using the `overloadable` attribute. For example, one might provide several overloaded versions of a `tgsin` function that invokes the appropriate standard function computing the sine of a value with `float`, `double`, or `long double` precision:

```
#include <math.h>
float __attribute__((overloadable)) tgsin(float x) { return sinf(x); }
double __attribute__((overloadable)) tgsin(double x) { return sin(x); }
long double __attribute__((overloadable)) tgsin(long double x) { return sinl(x); }
```

Given these declarations, one can call `tgsin` with a `float` value to receive a `float` result, with a `double` to receive a `double` result, etc. Function overloading in C follows the rules of C++ function overloading to pick the best overload given the call arguments, with a few C-specific semantics:

- Conversion from `float` or `double` to `long double` is ranked as a floating-point promotion (per C99) rather than as a floating-point conversion (as in C++).
- A conversion from a pointer of type `T*` to a pointer of type `U*` is considered a pointer conversion (with conversion rank) if `T` and `U` are compatible types.
- A conversion from type `T` to a value of type `U` is permitted if `T` and `U` are compatible types. This conversion is given “conversion” rank.

The declaration of `overloadable` functions is restricted to function declarations and definitions. Most importantly, if any function with a given name is given the `overloadable` attribute, then all function declarations and definitions with that name (and in that scope) must have the `overloadable` attribute. This rule even applies to redeclarations of functions whose original declaration had the `overloadable` attribute, e.g.,

```
int f(int) __attribute__((overloadable));
float f(float); // error: declaration of "f" must have the "overloadable" attribute

int g(int) __attribute__((overloadable));
int g(int) { } // error: redeclaration of "g" must also have the "overloadable" attribute
↪attribute
```

Functions marked `overloadable` must have prototypes. Therefore, the following code is ill-formed:

```
int h() __attribute__((overloadable)); // error: h does not have a prototype
```

However, `overloadable` functions are allowed to use an ellipsis even if there are no named parameters (as is permitted in C++). This feature is particularly useful when combined with the `unavailable` attribute:

```
void honeypot(...) __attribute__((overloadable, unavailable)); // calling me is an error
↪error
```

Functions declared with the `overloadable` attribute have their names mangled according to the same rules as C++ function names. For example, the three `tgsin` functions in our motivating example get the mangled names `_Z5tgsinf`, `_Z5tgsind`, and `_Z5tgsine`, respectively. There are two caveats to this use of name mangling:

- Future versions of Clang may change the name mangling of functions overloaded in C, so you should not depend on an specific mangling. To be completely safe, we strongly urge the use of `static inline` with `overloadable` functions.
- The `overloadable` attribute has almost no meaning when used in C++, because names will already be mangled and functions are already overloadable. However, when an `overloadable` function occurs within an `extern "C"` linkage specification, its name *will* be mangled in the same way as it would in C.

Query for this feature with `__has_extension(attribute_overloadable)`.

release_capability (release_shared_capability, clang::release_capability, clang::release_shared_capability)

Table 4.26: Supported Syntaxes

GNU	C++11	__declspec	Keyword	Pragma
X	X			

Marks a function as releasing a capability.

target (gnu::target)

Table 4.27: Supported Syntaxes

GNU	C++11	__declspec	Keyword	Pragma
X	X			

Clang supports the GNU style `__attribute__((target("OPTIONS")))` attribute. This attribute may be attached to a function definition and instructs the backend to use different code generation options than were passed on the command line.

The current set of options correspond to the existing “subtarget features” for the target with or without a “-mno-” in front corresponding to the absence of the feature, as well as `arch="CPU"` which will change the default “CPU” for the function.

Example “subtarget features” from the x86 backend include: “mmx”, “sse”, “sse4.2”, “avx”, “xop” and largely correspond to the machine specific options handled by the front end.

try_acquire_capability (try_acquire_shared_capability, clang::try_acquire_capability, clang::try_acquire_shared_capability)

Table 4.28: Supported Syntaxes

GNU	C++11	__declspec	Keyword	Pragma
X	X			

Marks a function that attempts to acquire a capability. This function may fail to actually acquire the capability; they accept a Boolean value determining whether acquiring the capability means success (true), or failing to acquire the capability means success (false).

Variable Attributes

init_seg

Table 4.29: Supported Syntaxes

GNU	C++11	__declspec	Keyword	Pragma
				X

The attribute applied by `pragma init_seg()` controls the section into which global initialization function pointers are emitted. It is only available with `-fms-extensions`. Typically, this function pointer is emitted into `.CRT$XCU` on Windows. The user can change the order of initialization by using a different section name with the same `.CRT$XC` prefix and a suffix that sorts lexicographically before or after the standard `.CRT$XCU` sections. See the [init_seg](#) documentation on MSDN for more information.

section (gnu::section, __declspec(allocate))

Table 4.30: Supported Syntaxes

GNU	C++11	__declspec	Keyword	Pragma
X	X	X		

The `section` attribute allows you to specify a specific section a global variable or function should be in after translation.

tls_model (gnu::tls_model)

Table 4.31: Supported Syntaxes

GNU	C++11	__declspec	Keyword	Pragma
X	X			

The `tls_model` attribute allows you to specify which thread-local storage model to use. It accepts the following strings:

- global-dynamic

- local-dynamic
- initial-exec
- local-exec

TLS models are mutually exclusive.

thread

Table 4.32: Supported Syntaxes

GNU	C++11	__declspec	Keyword	Pragma
		X		

The `__declspec(thread)` attribute declares a variable with thread local storage. It is available under the `-fms-extensions` flag for MSVC compatibility. See the documentation for `__declspec(thread)` on MSDN.

In Clang, `__declspec(thread)` is generally equivalent in functionality to the GNU `__thread` keyword. The variable must not have a destructor and must have a constant initializer, if any. The attribute only applies to variables declared with static storage duration, such as globals, class static data members, and static locals.

Type Attributes

align_value

Table 4.33: Supported Syntaxes

GNU	C++11	__declspec	Keyword	Pragma
X				

The `align_value` attribute can be added to the typedef of a pointer type or the declaration of a variable of pointer or reference type. It specifies that the pointer will point to, or the reference will bind to, only objects with at least the provided alignment. This alignment value must be some positive power of 2.

```
typedef double * aligned_double_ptr __attribute__((align_value(64)));
void foo(double & x __attribute__((align_value(128))),
        aligned_double_ptr y) { ... }
```

If the pointer value does not have the specified alignment at runtime, the behavior of the program is undefined.

flag_enum

Table 4.34: Supported Syntaxes

GNU	C++11	__declspec	Keyword	Pragma
X				

This attribute can be added to an enumerator to signal to the compiler that it is intended to be used as a flag type. This will cause the compiler to assume that the range of the type includes all of the values that you can get by manipulating bits of the enumerator when issuing warnings.

`__single_inheritance`, `__multiple_inheritance`, `__virtual_inheritance`

Table 4.35: Supported Syntaxes

GNU	C++11	__declspec	Keyword	Pragma
			X	

This collection of keywords is enabled under `-fms-extensions` and controls the pointer-to-member representation used on `*--win32` targets.

The `*--win32` targets utilize a pointer-to-member representation which varies in size and alignment depending on the definition of the underlying class.

However, this is problematic when a forward declaration is only available and no definition has been made yet. In such cases, Clang is forced to utilize the most general representation that is available to it.

These keywords make it possible to use a pointer-to-member representation other than the most general one regardless of whether or not the definition will ever be present in the current translation unit.

This family of keywords belong between the `class-key` and `class-name`:

```
struct __single_inheritance S;
int S::*i;
struct S {};
```

This keyword can be applied to class templates but only has an effect when used on full specializations:

```
template <typename T, typename U> struct __single_inheritance A; // warning: ↳inheritance model ignored on primary template
template <typename T> struct __multiple_inheritance A<T, T>; // warning: inheritance_↳model ignored on partial specialization
template <> struct __single_inheritance A<int, float>;
```

Note that choosing an inheritance model less general than strictly necessary is an error:

```
struct __multiple_inheritance S; // error: inheritance model does not match definition
int S::*i;
struct S {};
```

`nothrow`

Table 4.36: Supported Syntaxes

GNU	C++11	__declspec	Keyword	Pragma
		X		

This attribute can be added to a class declaration or definition to signal to the compiler that constructors and destructors will not reference the virtual function table.

Statement Attributes

fallthrough (clang::fallthrough)

Table 4.37: Supported Syntaxes

GNU	C++11	__declspec	Keyword	Pragma
	X			

The `clang::fallthrough` attribute is used along with the `-Wimplicit-fallthrough` argument to annotate intentional fall-through between switch labels. It can only be applied to a null statement placed at a point of execution between any statement and the next switch label. It is common to mark these places with a specific comment, but this attribute is meant to replace comments with a more strict annotation, which can be checked by the compiler. This attribute doesn't change semantics of the code and can be used wherever an intended fall-through occurs. It is designed to mimic control-flow statements like `break`; so it can be placed in most places where `break`; can, but only if there are no statements on the execution path between it and the next switch label.

Here is an example:

```
// compile with -Wimplicit-fallthrough
switch (n) {
case 22:
case 33: // no warning: no statements between case labels
    f();
case 44: // warning: unannotated fall-through
    g();
    [[clang::fallthrough]];
case 55: // no warning
    if (x) {
        h();
        break;
    }
    else {
        i();
        [[clang::fallthrough]];
    }
case 66: // no warning
    p();
    [[clang::fallthrough]]; // warning: fallthrough annotation does not
                           // directly precede case label
    q();
case 77: // warning: unannotated fall-through
    r();
}
```

#pragma clang loop

Table 4.38: Supported Syntaxes

GNU	C++11	__declspec	Keyword	Pragma
				X

The `#pragma clang loop` directive allows loop optimization hints to be specified for the subsequent loop. The directive allows vectorization, interleaving, and unrolling to be enabled or disabled. Vector width as well as interleave and unrolling count can be manually specified. See [language extensions](#) for details.

#pragma unroll, #pragma nounroll

Table 4.39: Supported Syntaxes

GNU	C++11	__declspec	Keyword	Pragma
				X

Loop unrolling optimization hints can be specified with `#pragma unroll` and `#pragma nounroll`. The pragma is placed immediately before a `for`, `while`, `do-while`, or `c++11` range-based `for` loop.

Specifying `#pragma unroll` without a parameter directs the loop unroller to attempt to fully unroll the loop if the trip count is known at compile time:

```
#pragma unroll
for (...) {
    ...
}
```

Specifying the optional parameter, `#pragma unroll _value_`, directs the unroller to unroll the loop `_value_` times. The parameter may optionally be enclosed in parentheses:

```
#pragma unroll 16
for (...) {
    ...
}

#pragma unroll(16)
for (...) {
    ...
}
```

Specifying `#pragma nounroll` indicates that the loop should not be unrolled:

```
#pragma nounroll
for (...) {
    ...
}
```

`#pragma unroll` and `#pragma unroll _value_` have identical semantics to `#pragma clang loop unroll(full)` and `#pragma clang loop unroll_count(_value_)` respectively. `#pragma nounroll` is equivalent to `#pragma clang loop unroll(disable)`. See [language extensions](#) for further details including limitations of the unroll hints.

Calling Conventions

Clang supports several different calling conventions, depending on the target platform and architecture. The calling convention used for a function determines how parameters are passed, how results are returned to the caller, and other low-level details of calling a function.

fastcall (gnu::fastcall, __fastcall, _fastcall)

Table 4.40: Supported Syntaxes

GNU	C++11	__declspec	Keyword	Pragma
X	X		X	

On 32-bit x86 targets, this attribute changes the calling convention of a function to use ECX and EDX as register parameters and clear parameters off of the stack on return. This convention does not support variadic calls or unprototyped functions in C, and has no effect on x86_64 targets. This calling convention is supported primarily for compatibility with existing code. Users seeking register parameters should use the `regparm` attribute, which does not require callee-cleanup. See the documentation for `__fastcall` on MSDN.

`ms_abi (gnu::ms_abi)`

Table 4.41: Supported Syntaxes

GNU	C++11	__declspec	Keyword	Pragma
X	X			

On non-Windows x86_64 targets, this attribute changes the calling convention of a function to match the default convention used on Windows x86_64. This attribute has no effect on Windows targets or non-x86_64 targets.

`pcs (gnu::pcs)`

Table 4.42: Supported Syntaxes

GNU	C++11	__declspec	Keyword	Pragma
X	X			

On ARM targets, this attribute can be used to select calling conventions similar to `stdcall` on x86. Valid parameter values are “aapcs” and “aapcs-vfp”.

`regparm (gnu::regparm)`

Table 4.43: Supported Syntaxes

GNU	C++11	__declspec	Keyword	Pragma
X	X			

On 32-bit x86 targets, the `regparm` attribute causes the compiler to pass the first three integer parameters in EAX, EDX, and ECX instead of on the stack. This attribute has no effect on variadic functions, and all parameters are passed via the stack as normal.

`stdcall (gnu::stdcall, __stdcall, _stdcall)`

Table 4.44: Supported Syntaxes

GNU	C++11	__declspec	Keyword	Pragma
X	X		X	

On 32-bit x86 targets, this attribute changes the calling convention of a function to clear parameters off of the stack on return. This convention does not support variadic calls or unprototyped functions in C, and has no effect on x86_64 targets. This calling convention is used widely by the Windows API and COM applications. See the documentation for `__stdcall` on MSDN.

thiscall (`gnu::thiscall`, `__thiscall`, `_thiscall`)

Table 4.45: Supported Syntaxes

GNU	C++11	__declspec	Keyword	Pragma
X	X		X	

On 32-bit x86 targets, this attribute changes the calling convention of a function to use ECX for the first parameter (typically the implicit `this` parameter of C++ methods) and clear parameters off of the stack on return. This convention does not support variadic calls or unprototyped functions in C, and has no effect on x86_64 targets. See the documentation for `__thiscall` on MSDN.

vectorcall (`__vectorcall`, `_vectorcall`)

Table 4.46: Supported Syntaxes

GNU	C++11	__declspec	Keyword	Pragma
X			X	

On 32-bit x86 *and* x86_64 targets, this attribute changes the calling convention of a function to pass vector parameters in SSE registers.

On 32-bit x86 targets, this calling convention is similar to `__fastcall`. The first two integer parameters are passed in ECX and EDX. Subsequent integer parameters are passed in memory, and callee clears the stack. On x86_64 targets, the callee does *not* clear the stack, and integer parameters are passed in RCX, RDX, R8, and R9 as is done for the default Windows x64 calling convention.

On both 32-bit x86 and x86_64 targets, vector and floating point arguments are passed in XMM0-XMM5. Homogeneous vector aggregates of up to four elements are passed in sequential SSE registers if enough are available. If AVX is enabled, 256 bit vectors are passed in YMM0-YMM5. Any vector or aggregate type that cannot be passed in registers for any reason is passed by reference, which allows the caller to align the parameter memory.

See the documentation for `__vectorcall` on MSDN for more details.

Consumed Annotation Checking

Clang supports additional attributes for checking basic resource management properties, specifically for unique objects that have a single owning reference. The following attributes are currently supported, although **the implementation for these annotations is currently in development and are subject to change**.

callable_when

Table 4.47: Supported Syntaxes

GNU	C++11	__declspec	Keyword	Pragma
X				

Use `__attribute__((callable_when(...)))` to indicate what states a method may be called in. Valid states are unconsumed, consumed, or unknown. Each argument to this attribute must be a quoted string. E.g.:

```
__attribute__((callable_when("unconsumed", "unknown")))
```

consumable

Table 4.48: Supported Syntaxes

GNU	C++11	__declspec	Keyword	Pragma
X				

Each `class` that uses any of the `typestate` annotations must first be marked using the `consumable` attribute. Failure to do so will result in a warning.

This attribute accepts a single parameter that must be one of the following: `unknown`, `consumed`, or `unconsumed`.

param_typestate

Table 4.49: Supported Syntaxes

GNU	C++11	__declspec	Keyword	Pragma
X				

This attribute specifies expectations about function parameters. Calls to an function with annotated parameters will issue a warning if the corresponding argument isn't in the expected state. The attribute is also used to set the initial state of the parameter when analyzing the function's body.

return_typestate

Table 4.50: Supported Syntaxes

GNU	C++11	__declspec	Keyword	Pragma
X				

The `return_typestate` attribute can be applied to functions or parameters. When applied to a function the attribute specifies the state of the returned value. The function's body is checked to ensure that it always returns a value in the specified state. On the caller side, values returned by the annotated function are initialized to the given state.

When applied to a function parameter it modifies the state of an argument after a call to the function returns. The function's body is checked to ensure that the parameter is in the expected state before returning.

set_typestate

Table 4.51: Supported Syntaxes

GNU	C++11	__declspec	Keyword	Pragma
X				

Annotate methods that transition an object into a new state with `__attribute__((set_typestate(new_state)))`. The new state must be `unconsumed`, `consumed`, or `unknown`.

test_tystate

Table 4.52: Supported Syntaxes

GNU	C++11	__declspec	Keyword	Pragma
X				

Use `__attribute__((test_tystate(tested_state)))` to indicate that a method returns true if the object is in the specified state..

Type Safety Checking

Clang supports additional attributes to enable checking type safety properties that can't be enforced by the C type system. Use cases include:

- MPI library implementations, where these attributes enable checking that the buffer type matches the passed `MPI_Datatype`;
- for HDF5 library there is a similar use case to MPI;
- checking types of variadic functions' arguments for functions like `fcntl()` and `ioctl()`.

You can detect support for these attributes with `__has_attribute()`. For example:

```
#if defined(__has_attribute)
# if __has_attribute(argument_with_type_tag) && \
    __has_attribute(pointer_with_type_tag) && \
    __has_attribute(type_tag_for_datatype)
#   define ATTR_MPI_PWT(buffer_idx, type_idx) __attribute__((pointer_with_type_
↪tag(mpi,buffer_idx,type_idx)))
/* ... other macros ... */
# endif
#endif

#if !defined(ATTR_MPI_PWT)
# define ATTR_MPI_PWT(buffer_idx, type_idx)
#endif

int MPI_Send(void *buf, int count, MPI_Datatype datatype /*, other args omitted */)
    ATTR_MPI_PWT(1,3);
```

argument_with_type_tag

Table 4.53: Supported Syntaxes

GNU	C++11	__declspec	Keyword	Pragma
X				

Use `__attribute__((argument_with_type_tag(arg_kind, arg_idx, type_tag_idx)))` on a function declaration to specify that the function accepts a type tag that determines the type of some other argument. `arg_kind` is an identifier that should be used when annotating all applicable type tags.

This attribute is primarily useful for checking arguments of variadic functions (`pointer_with_type_tag` can be used in most non-variadic cases).

For example:

```
int fcntl(int fd, int cmd, ...)
__attribute__((argument_with_type_tag(fcntl,3,2)));
```

pointer_with_type_tag

Table 4.54: Supported Syntaxes

GNU	C++11	__declspec	Keyword	Pragma
X				

Use `__attribute__((pointer_with_type_tag(ptr_kind, ptr_idx, type_tag_idx)))` on a function declaration to specify that the function accepts a type tag that determines the pointee type of some other pointer argument.

For example:

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype /*, other args omitted */)
__attribute__((pointer_with_type_tag(mpi,1,3)));
```

type_tag_for_datatype

Table 4.55: Supported Syntaxes

GNU	C++11	__declspec	Keyword	Pragma
X				

Clang supports annotating type tags of two forms.

- **Type tag that is an expression containing a reference to some declared identifier.** Use `__attribute__((type_tag_for_datatype(kind, type)))` on a declaration with that identifier:

```
extern struct mpi_datatype mpi_datatype_int
__attribute__((type_tag_for_datatype(mpi,int)));
#define MPI_INT ((MPI_Datatype) &mpi_datatype_int)
```

- **Type tag that is an integral literal.** Introduce a static const variable with a corresponding initializer value and attach `__attribute__((type_tag_for_datatype(kind, type)))` on that declaration, for example:

```
#define MPI_INT ((MPI_Datatype) 42)
static const MPI_Datatype mpi_datatype_int
__attribute__((type_tag_for_datatype(mpi,int))) = 42
```

The attribute also accepts an optional third argument that determines how the expression is compared to the type tag. There are two supported flags:

- `layout_compatible` will cause types to be compared according to layout-compatibility rules (C++11 [class.mem] p 17, 18). This is implemented to support annotating types like `MPI_DOUBLE_INT`.

For example:

```
/* In mpi.h */
struct internal_mpi_double_int { double d; int i; };
extern struct mpi_datatype mpi_datatype_double_int
__attribute__((type_tag_for_datatype(mpi, struct internal_mpi_double_int,
layout_compatible)));
```

```

#define MPI_DOUBLE_INT ((MPI_Datatype) &mpi_datatype_double_int)

/* In user code */
struct my_pair { double a; int b; };
struct my_pair *buffer;
MPI_Send(buffer, 1, MPI_DOUBLE_INT /*, ... */); // no warning

struct my_int_pair { int a; int b; }
struct my_int_pair *buffer2;
MPI_Send(buffer2, 1, MPI_DOUBLE_INT /*, ... */); // warning: actual buffer_
↪element                                         // type 'struct my_int_pair'
                                                // doesn't match specified MPI_
↪Datatype

```

- `must_be_null` specifies that the expression should be a null pointer constant, for example:

```

/* In mpi.h */
extern struct mpi_datatype mpi_datatype_null
    __attribute__(( type_tag_for_datatype(mpi, void, must_be_null) ));

#define MPI_DATATYPE_NULL ((MPI_Datatype) &mpi_datatype_null)

/* In user code */
MPI_Send(buffer, 1, MPI_DATATYPE_NULL /*, ... */); // warning: MPI_DATATYPE_NULL
                                                    // was specified but buffer
                                                    // is not a null pointer

```

OpenCL Address Spaces

The address space qualifier may be used to specify the region of memory that is used to allocate the object. OpenCL supports the following address spaces: `__generic(generic)`, `__global(global)`, `__local(local)`, `__private(private)`, `__constant(constant)`.

```

__constant int c = ...;

__generic int* foo(global int* g) {
    __local int* l;
    private int p;
    ...
    return l;
}

```

More details can be found in the OpenCL C language Spec v2.0, Section 6.5.

`__constant(constant)`

Table 4.56: Supported Syntaxes

GNU	C++11	<code>__declspec</code>	Keyword	Pragma
			X	

The constant address space attribute signals that an object is located in a constant (non-modifiable) memory region. It

is available to all work items. Any type can be annotated with the constant address space attribute. Objects with the constant address space qualifier can be declared in any scope and must have an initializer.

`__generic(generic)`

Table 4.57: Supported Syntaxes

GNU	C++11	__declspec	Keyword	Pragma
			X	

The generic address space attribute is only available with OpenCL v2.0 and later. It can be used with pointer types. Variables in global and local scope and function parameters in non-kernel functions can have the generic address space type attribute. It is intended to be a placeholder for any other address space except for ‘__constant’ in OpenCL code which can be used with multiple address spaces.

`__global(global)`

Table 4.58: Supported Syntaxes

GNU	C++11	__declspec	Keyword	Pragma
			X	

The global address space attribute specifies that an object is allocated in global memory, which is accessible by all work items. The content stored in this memory area persists between kernel executions. Pointer types to the global address space are allowed as function parameters or local variables. Starting with OpenCL v2.0, the global address space can be used with global (program scope) variables and static local variable as well.

`__local(local)`

Table 4.59: Supported Syntaxes

GNU	C++11	__declspec	Keyword	Pragma
			X	

The local address space specifies that an object is allocated in the local (work group) memory area, which is accessible to all work items in the same work group. The content stored in this memory region is not accessible after the kernel execution ends. In a kernel function scope, any variable can be in the local address space. In other scopes, only pointer types to the local address space are allowed. Local address space variables cannot have an initializer.

`__private(private)`

Table 4.60: Supported Syntaxes

GNU	C++11	__declspec	Keyword	Pragma
			X	

The private address space specifies that an object is allocated in the private (work item) memory. Other work items cannot access the same memory area and its content is destroyed after work item execution ends. Local variables can be declared in the private address space. Function arguments are always in the private address space. Kernel function arguments of a pointer or an array type cannot point to the private address space.

Nullability Attributes

Whether a particular pointer may be “null” is an important concern when working with pointers in the C family of languages. The various nullability attributes indicate whether a particular pointer can be null or not, which makes APIs more expressive and can help static analysis tools identify bugs involving null pointers. Clang supports several kinds of nullability attributes: the `nonnull` and `returns_nonnull` attributes indicate which function or method parameters and result types can never be null, while nullability type qualifiers indicate which pointer types can be null (`_Nullable`) or cannot be null (`_Nonnull`).

The nullability (type) qualifiers express whether a value of a given pointer type can be null (the `_Nullable` qualifier), doesn’t have a defined meaning for null (the `_Nonnull` qualifier), or for which the purpose of null is unclear (the `_Null_unspecified` qualifier). Because nullability qualifiers are expressed within the type system, they are more general than the `nonnull` and `returns_nonnull` attributes, allowing one to express (for example) a nullable pointer to an array of nonnull pointers. Nullability qualifiers are written to the right of the pointer to which they apply. For example:

```
// No meaningful result when 'ptr' is null (here, it happens to be undefined_
↪behavior).
int fetch(int * _Nonnull ptr) { return *ptr; }

// 'ptr' may be null.
int fetch_or_zero(int * _Nullable ptr) {
    return ptr ? *ptr : 0;
}

// A nullable pointer to non-null pointers to const characters.
const char *join_strings(const char * _Nonnull * _Nullable strings, unsigned_
↪n);
```

In Objective-C, there is an alternate spelling for the nullability qualifiers that can be used in Objective-C methods and properties using context-sensitive, non-underscored keywords. For example:

```
@interface NSView : NSResponder
- (nullable NSView *)ancestorSharedWithView:(nonnull NSView *)aView;
@property (assign, nullable) NSView *superview;
@property (readonly, nonnull) NSArray *subviews;
@end
```

nonnull

Table 4.61: Supported Syntaxes

GNU	C++11	__declspec	Keyword	Pragma
X	X			

The `nonnull` attribute indicates that some function parameters must not be null, and can be used in several different ways. It’s original usage (from GCC) is as a function (or Objective-C method) attribute that specifies which parameters of the function are nonnull in a comma-separated list. For example:

```
extern void * my_memcpy (void *dest, const void *src, size_t len)
    __attribute__((nonnull (1, 2)));
```

Here, the `nonnull` attribute indicates that parameters 1 and 2 cannot have a null value. Omitting the parenthesized list of parameter indices means that all parameters of pointer type cannot be null:

```
extern void * my_memcpy (void *dest, const void *src, size_t len)
    __attribute__((nonnull));
```

Clang also allows the `nonnull` attribute to be placed directly on a function (or Objective-C method) parameter, eliminating the need to specify the parameter index ahead of type. For example:

```
extern void * my_memcpy (void *dest __attribute__((nonnull)),
    const void *src __attribute__((nonnull)), size_t_
    len);
```

Note that the `nonnull` attribute indicates that passing null to a non-null parameter is undefined behavior, which the optimizer may take advantage of to, e.g., remove null checks. The `_Nonnull` type qualifier indicates that a pointer cannot be null in a more general manner (because it is part of the type system) and does not imply undefined behavior, making it more widely applicable.

returns_nonnull

Table 4.62: Supported Syntaxes

GNU	C++11	__declspec	Keyword	Pragma
X	X			

The `returns_nonnull` attribute indicates that a particular function (or Objective-C method) always returns a non-null pointer. For example, a particular system `malloc` might be defined to terminate a process when memory is not available rather than returning a null pointer:

```
extern void * malloc (size_t size) __attribute__((returns_nonnull));
```

The `returns_nonnull` attribute implies that returning a null pointer is undefined behavior, which the optimizer may take advantage of. The `_Nonnull` type qualifier indicates that a pointer cannot be null in a more general manner (because it is part of the type system) and does not imply undefined behavior, making it more widely applicable.

_Nonnull

Table 4.63: Supported Syntaxes

GNU	C++11	__declspec	Keyword	Pragma
			X	

The `_Nonnull` nullability qualifier indicates that null is not a meaningful value for a value of the `_Nonnull` pointer type. For example, given a declaration such as:

```
int fetch(int * _Nonnull ptr);
```

a caller of `fetch` should not provide a null value, and the compiler will produce a warning if it sees a literal null value passed to `fetch`. Note that, unlike the declaration attribute `nonnull`, the presence of `_Nonnull` does not imply that passing null is undefined behavior: `fetch` is free to consider null undefined behavior or (perhaps for backward-compatibility reasons) defensively handle null.

`_Null_unspecified`

Table 4.64: Supported Syntaxes

GNU	C++11	__declspec	Keyword	Pragma
			X	

The `_Null_unspecified` nullability qualifier indicates that neither the `_Nonnull` nor `_Nullable` qualifiers make sense for a particular pointer type. It is used primarily to indicate that the role of null with specific pointers in a nullability-annotated header is unclear, e.g., due to overly-complex implementations or historical factors with a long-lived API.

`_Nullable`

Table 4.65: Supported Syntaxes

GNU	C++11	__declspec	Keyword	Pragma
			X	

The `_Nullable` nullability qualifier indicates that a value of the `_Nullable` pointer type can be null. For example, given:

```
int fetch_or_zero(int * _Nullable ptr);
```

a caller of `fetch_or_zero` can provide null.

Cross-compilation using Clang

Introduction

This document will guide you in choosing the right Clang options for cross-compiling your code to a different architecture. It assumes you already know how to compile the code in question for the host architecture, and that you know how to choose additional include and library paths.

However, this document is *not* a “how to” and won’t help you setting your build system or Makefiles, nor choosing the right CMake options, etc. Also, it does not cover all the possible options, nor does it contain specific examples for specific architectures. For a concrete example, the [instructions for cross-compiling LLVM itself](#) may be of interest.

After reading this document, you should be familiar with the main issues related to cross-compilation, and what main compiler options Clang provides for performing cross-compilation.

Cross compilation issues

In GCC world, every host/target combination has its own set of binaries, headers, libraries, etc. So, it’s usually simple to download a package with all files in, unzip to a directory and point the build system to that compiler, that will know about its location and find all it needs to when compiling your code.

On the other hand, Clang/LLVM is natively a cross-compiler, meaning that one set of programs can compile to all targets by setting the `-target` option. That makes it a lot easier for programers wishing to compile to different platforms and architectures, and for compiler developers that only have to maintain one build system, and for OS distributions, that need only one set of main packages.

But, as is true to any cross-compiler, and given the complexity of different architectures, OS's and options, it's not always easy finding the headers, libraries or binutils to generate target specific code. So you'll need special options to help Clang understand what target you're compiling to, where your tools are, etc.

Another problem is that compilers come with standard libraries only (like `compiler-rt`, `libcxx`, `libgcc`, `libm`, etc), so you'll have to find and make available to the build system, every other library required to build your software, that is specific to your target. It's not enough to have your host's libraries installed.

Finally, not all toolchains are the same, and consequently, not every Clang option will work magically. Some options, like `--sysroot` (which effectively changes the logical root for headers and libraries), assume all your binaries and libraries are in the same directory, which may not true when your cross-compiler was installed by the distribution's package management. So, for each specific case, you may use more than one option, and in most cases, you'll end up setting include paths (`-I`) and library paths (`-L`) manually.

To sum up, different toolchains can:

- be host/target specific or more flexible
- be in a single directory, or spread out across your system
- have different sets of libraries and headers by default
- need special options, which your build system won't be able to figure out by itself

General Cross-Compilation Options in Clang

Target Triple

The basic option is to define the target architecture. For that, use `-target <triple>`. If you don't specify the target, CPU names won't match (since Clang assumes the host triple), and the compilation will go ahead, creating code for the host platform, which will break later on when assembling or linking.

The triple has the general format `<arch><sub>--<vendor>--<sys>--<abi>`, where:

- `arch` = `x86`, `arm`, `thumb`, `mips`, etc.
- `sub` = for ex. on ARM: `v5`, `v6m`, `v7a`, `v7m`, etc.
- `vendor` = `pc`, `apple`, `nvidia`, `ibm`, etc.
- `sys` = `none`, `linux`, `win32`, `darwin`, `cuda`, etc.
- `abi` = `eabi`, `gnu`, `android`, `macho`, `elf`, etc.

The sub-architecture options are available for their own architectures, of course, so “x86v7a” doesn't make sense. The vendor needs to be specified only if there's a relevant change, for instance between PC and Apple. Most of the time it can be omitted (and Unknown) will be assumed, which sets the defaults for the specified architecture. The system name is generally the OS (`linux`, `darwin`), but could be special like the bare-metal “none”.

When a parameter is not important, it can be omitted, or you can choose `unknown` and the defaults will be used. If you choose a parameter that Clang doesn't know, like `blerg`, it'll ignore and assume `unknown`, which is not always desired, so be careful.

Finally, the ABI option is something that will pick default CPU/FPU, define the specific behaviour of your code (PCS, extensions), and also choose the correct library calls, etc.

CPU, FPU, ABI

Once your target is specified, it's time to pick the hardware you'll be compiling to. For every architecture, a default set of CPU/FPU/ABI will be chosen, so you'll almost always have to change it via flags.

Typical flags include:

- `-mcpu=<cpu-name>`, like `x86-64`, `swift`, `cortex-a15`
- `-mfpu=<fpu-name>`, like `SSE3`, `NEON`, controlling the FP unit available
- `-mfloat-abi=<fabi>`, like `soft`, `hard`, controlling which registers to use for floating-point

The default is normally the common denominator, so that Clang doesn't generate code that breaks. But that also means you won't get the best code for your specific hardware, which may mean orders of magnitude slower than you expect.

For example, if your target is `arm-none-eabi`, the default CPU will be `arm7tdmi` using `soft` float, which is extremely slow on modern cores, whereas if your triple is `armv7a-none-eabi`, it'll be Cortex-A8 with NEON, but still using `soft-float`, which is much better, but still not great.

Toolchain Options

There are three main options to control access to your cross-compiler: `--sysroot`, `-I`, and `-L`. The two last ones are well known, but they're particularly important for additional libraries and headers that are specific to your target.

There are two main ways to have a cross-compiler:

1. When you have extracted your cross-compiler from a zip file into a directory, you have to use `--sysroot=<path>`. The path is the root directory where you have unpacked your file, and Clang will look for the directories `bin`, `lib`, `include` in there.

In this case, your setup should be pretty much done (if no additional headers or libraries are needed), as Clang will find all binaries it needs (assembler, linker, etc) in there.

2. When you have installed via a package manager (modern Linux distributions have cross-compiler packages available), make sure the target triple you set is *also* the prefix of your cross-compiler toolchain.

In this case, Clang will find the other binaries (assembler, linker), but not always where the target headers and libraries are. People add system-specific clues to Clang often, but as things change, it's more likely that it won't find than the other way around.

So, here, you'll be a lot safer if you specify the include/library directories manually (via `-I` and `-L`).

Target-Specific Libraries

All libraries that you compile as part of your build will be cross-compiled to your target, and your build system will probably find them in the right place. But all dependencies that are normally checked against (like `libxml` or `libz` etc) will match against the host platform, not the target.

So, if the build system is not aware that you want to cross-compile your code, it will get every dependency wrong, and your compilation will fail during build time, not configure time.

Also, finding the libraries for your target are not as easy as for your host machine. There aren't many cross-libraries available as packages to most OS's, so you'll have to either cross-compile them from source, or download the package for your target platform, extract the libraries and headers, put them in specific directories and add `-I` and `-L` pointing to them.

Also, some libraries have different dependencies on different targets, so configuration tools to find dependencies in the host can get the list wrong for the target platform. This means that the configuration of your build can get things wrong when setting their own library paths, and you'll have to augment it via additional flags (`configure`, `Make`, `CMake`, etc).

Multilibs

When you want to cross-compile to more than one configuration, for example hard-float-ARM and soft-float-ARM, you'll have to have multiple copies of your libraries and (possibly) headers.

Some Linux distributions have support for Multilib, which handle that for you in an easier way, but if you're not careful and, for instance, forget to specify `-ccc-gcc-name armv7l-linux-gnueabi-gcc` (which uses hard-float), Clang will pick the `armv7l-linux-gnueabi-ld` (which uses soft-float) and linker errors will happen.

The same is true if you're compiling for different ABIs, like `gnueabi` and `androideabi`, and might even link and run, but produce run-time errors, which are much harder to track down and fix.

Thread Safety Analysis

Introduction

Clang Thread Safety Analysis is a C++ language extension which warns about potential race conditions in code. The analysis is completely static (i.e. compile-time); there is no run-time overhead. The analysis is still under active development, but it is mature enough to be deployed in an industrial setting. It is being developed by Google, in collaboration with CERT/SEI, and is used extensively in Google's internal code base.

Thread safety analysis works very much like a type system for multi-threaded programs. In addition to declaring the *type* of data (e.g. `int`, `float`, etc.), the programmer can (optionally) declare how access to that data is controlled in a multi-threaded environment. For example, if `foo` is *guarded* by the mutex `mu`, then the analysis will issue a warning whenever a piece of code reads or writes to `foo` without first locking `mu`. Similarly, if there are particular routines that should only be called by the GUI thread, then the analysis will warn if other threads call those routines.

Getting Started

```
#include "mutex.h"

class BankAccount {
private:
    Mutex mu;
    int balance GUARDED_BY(mu);

    void depositImpl(int amount) {
        balance += amount;          // WARNING! Cannot write balance without locking mu.
    }

    void withdrawImpl(int amount) REQUIRES(mu) {
        balance -= amount;          // OK. Caller must have locked mu.
    }

public:
    void withdraw(int amount) {
        mu.Lock();
        withdrawImpl(amount);       // OK. We've locked mu.
    }                                // WARNING! Failed to unlock mu.

    void transferFrom(BankAccount& b, int amount) {
        mu.Lock();
        b.withdrawImpl(amount);     // WARNING! Calling withdrawImpl() requires locking b.
        ↪ mu.
    }
}
```

```
    depositImpl(amount);           // OK.  depositImpl() has no requirements.
    mu.Unlock();
}
};
```

This example demonstrates the basic concepts behind the analysis. The `GUARDED_BY` attribute declares that a thread must lock `mu` before it can read or write to `balance`, thus ensuring that the increment and decrement operations are atomic. Similarly, `REQUIRES` declares that the calling thread must lock `mu` before calling `withdrawImpl`. Because the caller is assumed to have locked `mu`, it is safe to modify `balance` within the body of the method.

The `depositImpl()` method does not have `REQUIRES`, so the analysis issues a warning. Thread safety analysis is not inter-procedural, so caller requirements must be explicitly declared. There is also a warning in `transferFrom()`, because although the method locks `this->mu`, it does not lock `b.mu`. The analysis understands that these are two separate mutexes, in two different objects.

Finally, there is a warning in the `withdraw()` method, because it fails to unlock `mu`. Every lock must have a corresponding unlock, and the analysis will detect both double locks, and double unlocks. A function is allowed to acquire a lock without releasing it, (or vice versa), but it must be annotated as such (using `ACQUIRE/RELEASE`).

Running The Analysis

To run the analysis, simply compile with the `-Wthread-safety` flag, e.g.

```
clang -c -Wthread-safety example.cpp
```

Note that this example assumes the presence of a suitably annotated `mutex.h` that declares which methods perform locking, unlocking, and so on.

Basic Concepts: Capabilities

Thread safety analysis provides a way of protecting *resources* with *capabilities*. A resource is either a data member, or a function/method that provides access to some underlying resource. The analysis ensures that the calling thread cannot access the *resource* (i.e. call the function, or read/write the data) unless it has the *capability* to do so.

Capabilities are associated with named C++ objects which declare specific methods to acquire and release the capability. The name of the object serves to identify the capability. The most common example is a mutex. For example, if `mu` is a mutex, then calling `mu.Lock()` causes the calling thread to acquire the capability to access data that is protected by `mu`. Similarly, calling `mu.Unlock()` releases that capability.

A thread may hold a capability either *exclusively* or *shared*. An exclusive capability can be held by only one thread at a time, while a shared capability can be held by many threads at the same time. This mechanism enforces a multiple-reader, single-writer pattern. Write operations to protected data require exclusive access, while read operations require only shared access.

At any given moment during program execution, a thread holds a specific set of capabilities (e.g. the set of mutexes that it has locked.) These act like keys or tokens that allow the thread to access a given resource. Just like physical security keys, a thread cannot make copy of a capability, nor can it destroy one. A thread can only release a capability to another thread, or acquire one from another thread. The annotations are deliberately agnostic about the exact mechanism used to acquire and release capabilities; it assumes that the underlying implementation (e.g. the Mutex implementation) does the handoff in an appropriate manner.

The set of capabilities that are actually held by a given thread at a given point in program execution is a run-time concept. The static analysis works by calculating an approximation of that set, called the *capability environment*. The capability environment is calculated for every program point, and describes the set of capabilities that are statically known to be held, or not held, at that particular point. This environment is a conservative approximation of the full set of capabilities that will actually held by a thread at run-time.

Reference Guide

The thread safety analysis uses attributes to declare threading constraints. Attributes must be attached to named declarations, such as classes, methods, and data members. Users are *strongly advised* to define macros for the various attributes; example definitions can be found in [mutex.h](#), below. The following documentation assumes the use of macros.

For historical reasons, prior versions of thread safety used macro names that were very lock-centric. These macros have since been renamed to fit a more general capability model. The prior names are still in use, and will be mentioned under the tag *previously* where appropriate.

GUARDED_BY(c) and PT_GUARDED_BY(c)

GUARDED_BY is an attribute on data members, which declares that the data member is protected by the given capability. Read operations on the data require shared access, while write operations require exclusive access.

PT_GUARDED_BY is similar, but is intended for use on pointers and smart pointers. There is no constraint on the data member itself, but the *data that it points to* is protected by the given capability.

```
Mutex mu;
int *p1          GUARDED_BY(mu);
int *p2          PT_GUARDED_BY(mu);
unique_ptr<int> p3  PT_GUARDED_BY(mu);

void test() {
    p1 = 0;          // Warning!

    *p2 = 42;        // Warning!
    p2 = new int;    // OK.

    *p3 = 42;        // Warning!
    p3.reset(new int); // OK.
}
```

REQUIRES(...), REQUIRES_SHARED(...)

Previously: EXCLUSIVE_LOCKS_REQUIRED, SHARED_LOCKS_REQUIRED

REQUIRES is an attribute on functions or methods, which declares that the calling thread must have exclusive access to the given capabilities. More than one capability may be specified. The capabilities must be held on entry to the function, *and must still be held on exit*.

REQUIRES_SHARED is similar, but requires only shared access.

```
Mutex mu1, mu2;
int a GUARDED_BY(mu1);
int b GUARDED_BY(mu2);

void foo() REQUIRES(mu1, mu2) {
    a = 0;
    b = 0;
}

void test() {
    mu1.Lock();
    foo();          // Warning! Requires mu2.
}
```

```
    mul.Unlock();  
}
```

ACQUIRE(...), ACQUIRE_SHARED(...), RELEASE(...), RELEASE_SHARED(...)

Previously: EXCLUSIVE_LOCK_FUNCTION, SHARED_LOCK_FUNCTION, UNLOCK_FUNCTION

ACQUIRE is an attribute on functions or methods, which declares that the function acquires a capability, but does not release it. The caller must not hold the given capability on entry, and it will hold the capability on exit. ACQUIRE_SHARED is similar.

RELEASE and RELEASE_SHARED declare that the function releases the given capability. The caller must hold the capability on entry, and will no longer hold it on exit. It does not matter whether the given capability is shared or exclusive.

```
Mutex mu;  
MyClass myObject GUARDED_BY(mu);  
  
void lockAndInit() ACQUIRE(mu) {  
    mu.Lock();  
    myObject.init();  
}  
  
void cleanupAndUnlock() RELEASE(mu) {  
    myObject.cleanup();  
}                                     // Warning! Need to unlock mu.  
  
void test() {  
    lockAndInit();  
    myObject.doSomething();  
    cleanupAndUnlock();  
    myObject.doSomething(); // Warning, mu is not locked.  
}
```

If no argument is passed to ACQUIRE or RELEASE, then the argument is assumed to be `this`, and the analysis will not check the body of the function. This pattern is intended for use by classes which hide locking details behind an abstract interface. For example:

```
template <class T>  
class CAPABILITY("mutex") Container {  
private:  
    Mutex mu;  
    T* data;  
  
public:  
    // Hide mu from public interface.  
    void Lock() ACQUIRE() { mu.Lock(); }  
    void Unlock() RELEASE() { mu.Unlock(); }  
  
    T& getElem(int i) { return data[i]; }  
};  
  
void test() {  
    Container<int> c;  
    c.Lock();  
    int i = c.getElem(0);  
}
```

```
c.Unlock();
}
```

EXCLUDES(...)

Previously: LOCKS_EXCLUDED

EXCLUDES is an attribute on functions or methods, which declares that the caller must *not* hold the given capabilities. This annotation is used to prevent deadlock. Many mutex implementations are not re-entrant, so deadlock can occur if the function acquires the mutex a second time.

```
Mutex mu;
int a GUARDED_BY(mu);

void clear() EXCLUDES(mu) {
    mu.Lock();
    a = 0;
    mu.Unlock();
}

void reset() {
    mu.Lock();
    clear();    // Warning! Caller cannot hold 'mu'.
    mu.Unlock();
}
```

Unlike REQUIRES, EXCLUDES is optional. The analysis will not issue a warning if the attribute is missing, which can lead to false negatives in some cases. This issue is discussed further in [Negative Capabilities](#).

NO_THREAD_SAFETY_ANALYSIS

NO_THREAD_SAFETY_ANALYSIS is an attribute on functions or methods, which turns off thread safety checking for that method. It provides an escape hatch for functions which are either (1) deliberately thread-unsafe, or (2) are thread-safe, but too complicated for the analysis to understand. Reasons for (2) will be described in the [Known Limitations](#), below.

```
class Counter {
    Mutex mu;
    int a GUARDED_BY(mu);

    void unsafeIncrement() NO_THREAD_SAFETY_ANALYSIS { a++; }
};
```

Unlike the other attributes, NO_THREAD_SAFETY_ANALYSIS is not part of the interface of a function, and should thus be placed on the function definition (in the .cc or .cpp file) rather than on the function declaration (in the header).

RETURN_CAPABILITY(c)

Previously: LOCK_RETURNED

RETURN_CAPABILITY is an attribute on functions or methods, which declares that the function returns a reference to the given capability. It is used to annotate getter methods that return mutexes.

```
class MyClass {
private:
    Mutex mu;
    int a GUARDED_BY(mu);

public:
    Mutex* getMu() RETURN_CAPABILITY(mu) { return &mu; }

    // analysis knows that getMu() == mu
    void clear() REQUIRES(getMu()) { a = 0; }
};
```

ACQUIRED_BEFORE(...), ACQUIRED_AFTER(...)

ACQUIRED_BEFORE and ACQUIRED_AFTER are attributes on member declarations, specifically declarations of mutexes or other capabilities. These declarations enforce a particular order in which the mutexes must be acquired, in order to prevent deadlock.

```
Mutex m1;
Mutex m2 ACQUIRED_AFTER(m1);

// Alternative declaration
// Mutex m2;
// Mutex m1 ACQUIRED_BEFORE(m2);

void foo() {
    m2.Lock();
    m1.Lock(); // Warning! m2 must be acquired after m1.
    m1.Unlock();
    m2.Unlock();
}
```

CAPABILITY(<string>)

Previously: LOCKABLE

CAPABILITY is an attribute on classes, which specifies that objects of the class can be used as a capability. The string argument specifies the kind of capability in error messages, e.g. "mutex". See the Container example given above, or the Mutex class in *mutex.h*.

SCOPED_CAPABILITY

Previously: SCOPED_LOCKABLE

SCOPED_CAPABILITY is an attribute on classes that implement RAII-style locking, in which a capability is acquired in the constructor, and released in the destructor. Such classes require special handling because the constructor and destructor refer to the capability via different names; see the MutexLocker class in *mutex.h*, below.

TRY_ACQUIRE(<bool>, ...), TRY_ACQUIRE_SHARED(<bool>, ...)

Previously: EXCLUSIVE_TRYLOCK_FUNCTION, SHARED_TRYLOCK_FUNCTION

These are attributes on a function or method that tries to acquire the given capability, and returns a boolean value indicating success or failure. The first argument must be `true` or `false`, to specify which return value indicates success, and the remaining arguments are interpreted in the same way as `ACQUIRE`. See [mutex.h](#), below, for example uses.

ASSERT_CAPABILITY(...) and ASSERT_SHARED_CAPABILITY(...)

Previously: `ASSERT_EXCLUSIVE_LOCK`, `ASSERT_SHARED_LOCK`

These are attributes on a function or method that does a run-time test to see whether the calling thread holds the given capability. The function is assumed to fail (no return) if the capability is not held. See [mutex.h](#), below, for example uses.

GUARDED_VAR and PT_GUARDED_VAR

Use of these attributes has been deprecated.

Warning flags

- `-Wthread-safety`: Umbrella flag which turns on the following three:
 - `-Wthread-safety-attributes`: Sanity checks on attribute syntax.
 - `-Wthread-safety-analysis`: The core analysis.
 - **`-Wthread-safety-precise`: Requires that mutex expressions match precisely.** This warning can be disabled for code which has a lot of aliases.
 - `-Wthread-safety-reference`: Checks when guarded members are passed by reference.

Negative Capabilities are an experimental feature, which are enabled with:

- `-Wthread-safety-negative`: Negative capabilities. Off by default.

When new features and checks are added to the analysis, they can often introduce additional warnings. Those warnings are initially released as *beta* warnings for a period of time, after which they are migrated into the standard analysis.

- `-Wthread-safety-beta`: New features. Off by default.

Negative Capabilities

Thread Safety Analysis is designed to prevent both race conditions and deadlock. The `GUARDED_BY` and `REQUIRES` attributes prevent race conditions, by ensuring that a capability is held before reading or writing to guarded data, and the `EXCLUDES` attribute prevents deadlock, by making sure that a mutex is *not* held.

However, `EXCLUDES` is an optional attribute, and does not provide the same safety guarantee as `REQUIRES`. In particular:

- A function which acquires a capability does not have to exclude it.
- A function which calls a function that excludes a capability does not have transitively exclude that capability.

As a result, `EXCLUDES` can easily produce false negatives:

```
class Foo {
    Mutex mu;

    void foo() {
```

```
mu.Lock();
bar();           // No warning.
baz();           // No warning.
mu.Unlock();
}

void bar() {      // No warning.  (Should have EXCLUDES(mu)).
    mu.Lock();
    // ...
    mu.Unlock();
}

void baz() {
    bif();        // No warning.  (Should have EXCLUDES(mu)).
}

void bif() EXCLUDES(mu);
};
```

Negative requirements are an alternative EXCLUDES that provide a stronger safety guarantee. A negative requirement uses the REQUIRES attribute, in conjunction with the `!` operator, to indicate that a capability should *not* be held.

For example, using `REQUIRES(!mu)` instead of `EXCLUDES(mu)` will produce the appropriate warnings:

```
class FooNeg {
    Mutex mu;

    void foo() REQUIRES(!mu) {    // foo() now requires !mu.
        mu.Lock();
        bar();
        baz();
        mu.Unlock();
    }

    void bar() {
        mu.Lock();               // WARNING! Missing REQUIRES(!mu).
        // ...
        mu.Unlock();
    }

    void baz() {
        bif();                   // WARNING! Missing REQUIRES(!mu).
    }

    void bif() REQUIRES(!mu);
};
```

Negative requirements are an experimental feature which is off by default, because it will produce many warnings in existing code. It can be enabled by passing `-Wthread-safety-negative`.

Frequently Asked Questions

17. Should I put attributes in the header file, or in the `.cc/.cpp/.cxx` file?

(A) Attributes are part of the formal interface of a function, and should always go in the header, where they are visible to anything that includes the header. Attributes in the `.cpp` file are not visible outside of the immediate translation unit, which leads to false negatives and false positives.

17. “*Mutex is not locked on every path through here?*” What does that mean?

1. See *No conditionally held locks.*, below.

Known Limitations

Lexical scope

Thread safety attributes contain ordinary C++ expressions, and thus follow ordinary C++ scoping rules. In particular, this means that mutexes and other capabilities must be declared before they can be used in an attribute. Use-before-declaration is okay within a single class, because attributes are parsed at the same time as method bodies. (C++ delays parsing of method bodies until the end of the class.) However, use-before-declaration is not allowed between classes, as illustrated below.

```
class Foo;

class Bar {
    void bar(Foo* f) REQUIRES(f->mu); // Error: mu undeclared.
};

class Foo {
    Mutex mu;
};
```

Private Mutexes

Good software engineering practice dictates that mutexes should be private members, because the locking mechanism used by a thread-safe class is part of its internal implementation. However, private mutexes can sometimes leak into the public interface of a class. Thread safety attributes follow normal C++ access restrictions, so if `mu` is a private member of `c`, then it is an error to write `c.mu` in an attribute.

One workaround is to (ab)use the `RETURN_CAPABILITY` attribute to provide a public *name* for a private mutex, without actually exposing the underlying mutex. For example:

```
class MyClass {
private:
    Mutex mu;

public:
    // For thread safety analysis only. Does not actually return mu.
    Mutex* getMu() RETURN_CAPABILITY(mu) { return 0; }

    void doSomething() REQUIRES(mu);
};

void doSomethingTwice(MyClass& c) REQUIRES(c.getMu()) {
    // The analysis thinks that c.getMu() == c.mu
    c.doSomething();
    c.doSomething();
}
```

In the above example, `doSomethingTwice()` is an external routine that requires `c.mu` to be locked, which cannot be declared directly because `mu` is private. This pattern is discouraged because it violates encapsulation, but it is sometimes necessary, especially when adding annotations to an existing code base. The workaround is to define `getMu()` as a fake getter method, which is provided only for the benefit of thread safety analysis.

No conditionally held locks.

The analysis must be able to determine whether a lock is held, or not held, at every program point. Thus, sections of code where a lock *might be held* will generate spurious warnings (false positives). For example:

```
void foo() {
    bool b = needsToLock();
    if (b) mu.Lock();
    ... // Warning! Mutex 'mu' is not held on every path through here.
    if (b) mu.Unlock();
}
```

No checking inside constructors and destructors.

The analysis currently does not do any checking inside constructors or destructors. In other words, every constructor and destructor is treated as if it was annotated with `NO_THREAD_SAFETY_ANALYSIS`. The reason for this is that during initialization, only one thread typically has access to the object which is being initialized, and it is thus safe (and common practice) to initialize guarded members without acquiring any locks. The same is true of destructors.

Ideally, the analysis would allow initialization of guarded members inside the object being initialized or destroyed, while still enforcing the usual access restrictions on everything else. However, this is difficult to enforce in practice, because in complex pointer-based data structures, it is hard to determine what data is owned by the enclosing object.

No inlining.

Thread safety analysis is strictly intra-procedural, just like ordinary type checking. It relies only on the declared attributes of a function, and will not attempt to inline any method calls. As a result, code such as the following will not work:

```
template<class T>
class AutoCleanup {
    T* object;
    void (T::*mp)();

public:
    AutoCleanup(T* obj, void (T::*imp)()) : object(obj), mp(imp) { }
    ~AutoCleanup() { (object->*mp)(); }
};

Mutex mu;
void foo() {
    mu.Lock();
    AutoCleanup<Mutex>(&mu, &Mutex::Unlock);
    // ...
} // Warning, mu is not unlocked.
```

In this case, the destructor of `Autocleanup` calls `mu.Unlock()`, so the warning is bogus. However, thread safety analysis cannot see the unlock, because it does not attempt to inline the destructor. Moreover, there is no way to annotate the destructor, because the destructor is calling a function that is not statically known. This pattern is simply not supported.

No alias analysis.

The analysis currently does not track pointer aliases. Thus, there can be false positives if two pointers both point to the same mutex.

```
class MutexUnlocker {
    Mutex* mu;

public:
    MutexUnlocker(Mutex* m) RELEASE(m) : mu(m) { mu->Unlock(); }
    ~MutexUnlocker() ACQUIRE(mu) { mu->Lock(); }
};

Mutex mutex;
void test() REQUIRES(mutex) {
    {
        MutexUnlocker munl(&mutex); // unlocks mutex
        doSomeIO();
    }                               // Warning: locks munl.mu
}
```

The MutexUnlocker class is intended to be the dual of the MutexLocker class, defined in [mutex.h](#). However, it doesn't work because the analysis doesn't know that `munl.mu == mutex`. The `SCOPED_CAPABILITY` attribute handles aliasing for MutexLocker, but does so only for that particular pattern.

ACQUIRED_BEFORE(...) and ACQUIRED_AFTER(...) are currently unimplemented.

To be fixed in a future update.

mutex.h

Thread safety analysis can be used with any threading library, but it does require that the threading API be wrapped in classes and methods which have the appropriate annotations. The following code provides `mutex.h` as an example; these methods should be filled in to call the appropriate underlying implementation.

```
#ifndef THREAD_SAFETY_ANALYSIS_MUTEX_H
#define THREAD_SAFETY_ANALYSIS_MUTEX_H

// Enable thread safety attributes only with clang.
// The attributes can be safely erased when compiling with other compilers.
#if defined(__clang__) && (!defined(SWIG))
#define THREAD_ANNOTATION_ATTRIBUTE__(x) __attribute__((x))
#else
#define THREAD_ANNOTATION_ATTRIBUTE__(x) // no-op
#endif

#define THREAD_ANNOTATION_ATTRIBUTE__(x) __attribute__((x))

#define CAPABILITY(x) \
    THREAD_ANNOTATION_ATTRIBUTE__(capability(x))

#define SCOPED_CAPABILITY \
    THREAD_ANNOTATION_ATTRIBUTE__(scoped_lockable)

#define GUARDED_BY(x) \
```

```
    THREAD_ANNOTATION_ATTRIBUTE__ (guarded_by(x))

#define PT_GUARDED_BY(x) \
    THREAD_ANNOTATION_ATTRIBUTE__ (pt_guarded_by(x))

#define ACQUIRED_BEFORE(...) \
    THREAD_ANNOTATION_ATTRIBUTE__ (acquired_before(__VA_ARGS__))

#define ACQUIRED_AFTER(...) \
    THREAD_ANNOTATION_ATTRIBUTE__ (acquired_after(__VA_ARGS__))

#define REQUIRES(...) \
    THREAD_ANNOTATION_ATTRIBUTE__ (requires_capability(__VA_ARGS__))

#define REQUIRES_SHARED(...) \
    THREAD_ANNOTATION_ATTRIBUTE__ (requires_shared_capability(__VA_ARGS__))

#define ACQUIRE(...) \
    THREAD_ANNOTATION_ATTRIBUTE__ (acquire_capability(__VA_ARGS__))

#define ACQUIRE_SHARED(...) \
    THREAD_ANNOTATION_ATTRIBUTE__ (acquire_shared_capability(__VA_ARGS__))

#define RELEASE(...) \
    THREAD_ANNOTATION_ATTRIBUTE__ (release_capability(__VA_ARGS__))

#define RELEASE_SHARED(...) \
    THREAD_ANNOTATION_ATTRIBUTE__ (release_shared_capability(__VA_ARGS__))

#define TRY_ACQUIRE(...) \
    THREAD_ANNOTATION_ATTRIBUTE__ (try_acquire_capability(__VA_ARGS__))

#define TRY_ACQUIRE_SHARED(...) \
    THREAD_ANNOTATION_ATTRIBUTE__ (try_acquire_shared_capability(__VA_ARGS__))

#define EXCLUDES(...) \
    THREAD_ANNOTATION_ATTRIBUTE__ (locks_excluded(__VA_ARGS__))

#define ASSERT_CAPABILITY(x) \
    THREAD_ANNOTATION_ATTRIBUTE__ (assert_capability(x))

#define ASSERT_SHARED_CAPABILITY(x) \
    THREAD_ANNOTATION_ATTRIBUTE__ (assert_shared_capability(x))

#define RETURN_CAPABILITY(x) \
    THREAD_ANNOTATION_ATTRIBUTE__ (lock_returned(x))

#define NO_THREAD_SAFETY_ANALYSIS \
    THREAD_ANNOTATION_ATTRIBUTE__ (no_thread_safety_analysis)

// Defines an annotated interface for mutexes.
// These methods can be implemented to use any internal mutex implementation.
class CAPABILITY("mutex") Mutex {
public:
    // Acquire/lock this mutex exclusively. Only one thread can have exclusive
    // access at any one time. Write operations to guarded data require an
    // exclusive lock.
```

```

void Lock() ACQUIRE();

// Acquire/lock this mutex for read operations, which require only a shared
// lock. This assumes a multiple-reader, single writer semantics. Multiple
// threads may acquire the mutex simultaneously as readers, but a writer
// must wait for all of them to release the mutex before it can acquire it
// exclusively.
void ReaderLock() ACQUIRE_SHARED();

// Release/unlock an exclusive mutex.
void Unlock() RELEASE();

// Release/unlock a shared mutex.
void ReaderUnlock() RELEASE_SHARED();

// Try to acquire the mutex. Returns true on success, and false on failure.
bool TryLock() TRY_ACQUIRE(true);

// Try to acquire the mutex for read operations.
bool ReaderTryLock() TRY_ACQUIRE_SHARED(true);

// Assert that this mutex is currently held by the calling thread.
void AssertHeld() ASSERT_CAPABILITY(this);

// Assert that is mutex is currently held for read operations.
void AssertReaderHeld() ASSERT_SHARED_CAPABILITY(this);

// For negative capabilities.
const Mutex& operator!() const { return *this; }
};

// MutexLocker is an RAII class that acquires a mutex in its constructor, and
// releases it in its destructor.
class SCOPED_CAPABILITY MutexLocker {
private:
    Mutex* mut;

public:
    MutexLocker(Mutex *mu) ACQUIRE(mu) : mut(mu) {
        mu->Lock();
    }
    ~MutexLocker() RELEASE() {
        mut->Unlock();
    }
};

#ifdef USE_LOCK_STYLE_THREAD_SAFETY_ATTRIBUTES
// The original version of thread safety analysis the following attribute
// definitions. These use a lock-based terminology. They are still in use
// by existing thread safety code, and will continue to be supported.

// Deprecated.
#define PT_GUARDED_VAR \
    THREAD_ANNOTATION_ATTRIBUTE__(pt_guarded)

// Deprecated.

```

```
#define GUARDED_VAR \
    THREAD_ANNOTATION_ATTRIBUTE__(guarded)

// Replaced by REQUIRES
#define EXCLUSIVE_LOCKS_REQUIRED(...) \
    THREAD_ANNOTATION_ATTRIBUTE__(exclusive_locks_required(__VA_ARGS__))

// Replaced by REQUIRES_SHARED
#define SHARED_LOCKS_REQUIRED(...) \
    THREAD_ANNOTATION_ATTRIBUTE__(shared_locks_required(__VA_ARGS__))

// Replaced by CAPABILITY
#define LOCKABLE \
    THREAD_ANNOTATION_ATTRIBUTE__(lockable)

// Replaced by SCOPED_CAPABILITY
#define SCOPED_LOCKABLE \
    THREAD_ANNOTATION_ATTRIBUTE__(scoped_lockable)

// Replaced by ACQUIRE
#define EXCLUSIVE_LOCK_FUNCTION(...) \
    THREAD_ANNOTATION_ATTRIBUTE__(exclusive_lock_function(__VA_ARGS__))

// Replaced by ACQUIRE_SHARED
#define SHARED_LOCK_FUNCTION(...) \
    THREAD_ANNOTATION_ATTRIBUTE__(shared_lock_function(__VA_ARGS__))

// Replaced by RELEASE and RELEASE_SHARED
#define UNLOCK_FUNCTION(...) \
    THREAD_ANNOTATION_ATTRIBUTE__(unlock_function(__VA_ARGS__))

// Replaced by TRY_ACQUIRE
#define EXCLUSIVE_TRYLOCK_FUNCTION(...) \
    THREAD_ANNOTATION_ATTRIBUTE__(exclusive_trylock_function(__VA_ARGS__))

// Replaced by TRY_ACQUIRE_SHARED
#define SHARED_TRYLOCK_FUNCTION(...) \
    THREAD_ANNOTATION_ATTRIBUTE__(shared_trylock_function(__VA_ARGS__))

// Replaced by ASSERT_CAPABILITY
#define ASSERT_EXCLUSIVE_LOCK(...) \
    THREAD_ANNOTATION_ATTRIBUTE__(assert_exclusive_lock(__VA_ARGS__))

// Replaced by ASSERT_SHARED_CAPABILITY
#define ASSERT_SHARED_LOCK(...) \
    THREAD_ANNOTATION_ATTRIBUTE__(assert_shared_lock(__VA_ARGS__))

// Replaced by EXCLUDE_CAPABILITY.
#define LOCKS_EXCLUDED(...) \
    THREAD_ANNOTATION_ATTRIBUTE__(locks_excluded(__VA_ARGS__))

// Replaced by RETURN_CAPABILITY
#define LOCK_RETURNED(x) \
    THREAD_ANNOTATION_ATTRIBUTE__(lock_returned(x))

#endif // USE_LOCK_STYLE_THREAD_SAFETY_ATTRIBUTES

#endif // THREAD_SAFETY_ANALYSIS_MUTEX_H
```

AddressSanitizer

- *Introduction*
- *How to build*
- *Usage*
- *Symbolizing the Reports*
- *Additional Checks*
 - *Initialization order checking*
 - *Memory leak detection*
- *Issue Suppression*
 - *Suppressing Reports in External Libraries*
 - *Conditional Compilation with `__has_feature(address_sanitizer)`*
 - *Disabling Instrumentation with `__attribute__((no_sanitize("address")))`*
 - *Suppressing Errors in Recompiled Code (Blacklist)*
- *Limitations*
- *Supported Platforms*
- *Current Status*
- *More Information*

Introduction

AddressSanitizer is a fast memory error detector. It consists of a compiler instrumentation module and a run-time library. The tool can detect the following types of bugs:

- Out-of-bounds accesses to heap, stack and globals
- Use-after-free
- Use-after-return (to some extent)
- Double-free, invalid free
- Memory leaks (experimental)

Typical slowdown introduced by AddressSanitizer is **2x**.

How to build

Build LLVM/Clang with [CMake](#).

Usage

Simply compile and link your program with `-fsanitize=address` flag. The AddressSanitizer run-time library should be linked to the final executable, so make sure to use `clang` (not `ld`) for the final link step. When linking

shared libraries, the AddressSanitizer run-time is not linked, so `-Wl,-z,defs` may cause link errors (don't use it with AddressSanitizer). To get a reasonable performance add `-O1` or higher. To get nicer stack traces in error messages add `-fno-omit-frame-pointer`. To get perfect stack traces you may need to disable inlining (just use `-O1`) and tail call elimination (`-fno-optimize-sibling-calls`).

```
% cat example_UseAfterFree.cc
int main(int argc, char **argv) {
    int *array = new int[100];
    delete [] array;
    return array[argc]; // BOOM
}

# Compile and link
% clang -O1 -g -fsanitize=address -fno-omit-frame-pointer example_UseAfterFree.cc
```

or:

```
# Compile
% clang -O1 -g -fsanitize=address -fno-omit-frame-pointer -c example_UseAfterFree.cc
# Link
% clang -g -fsanitize=address example_UseAfterFree.o
```

If a bug is detected, the program will print an error message to `stderr` and exit with a non-zero exit code. AddressSanitizer exits on the first detected error. This is by design:

- This approach allows AddressSanitizer to produce faster and smaller generated code (both by ~5%).
- Fixing bugs becomes unavoidable. AddressSanitizer does not produce false alarms. Once a memory corruption occurs, the program is in an inconsistent state, which could lead to confusing results and potentially misleading subsequent reports.

If your process is sandboxed and you are running on OS X 10.10 or earlier, you will need to set `DYLD_INSERT_LIBRARIES` environment variable and point it to the ASan library that is packaged with the compiler used to build the executable. (You can find the library by searching for dynamic libraries with `asan` in their name.) If the environment variable is not set, the process will try to re-exec. Also keep in mind that when moving the executable to another machine, the ASan library will also need to be copied over.

Symbolizing the Reports

To make AddressSanitizer symbolize its output you need to set the `ASAN_SYMBOLIZER_PATH` environment variable to point to the `llvm-symbolizer` binary (or make sure `llvm-symbolizer` is in your `$PATH`):

```
% ASAN_SYMBOLIZER_PATH=/usr/local/bin/llvm-symbolizer ./a.out
==9442== ERROR: AddressSanitizer heap-use-after-free on address 0x7f7ddab8c084 at pc_
↪0x403c8c bp 0x7fff87fb82d0 sp 0x7fff87fb82c8
READ of size 4 at 0x7f7ddab8c084 thread T0
    #0 0x403c8c in main example_UseAfterFree.cc:4
    #1 0x7f7ddab8c084 in __libc_start_main ??:0
0x7f7ddab8c084 is located 4 bytes inside of 400-byte region [0x7f7ddab8c080,
↪0x7f7ddab8c210)
freed by thread T0 here:
    #0 0x404704 in operator delete[](void*) ??:0
    #1 0x403c53 in main example_UseAfterFree.cc:4
    #2 0x7f7ddab8c084 in __libc_start_main ??:0
previously allocated by thread T0 here:
    #0 0x404544 in operator new[](unsigned long) ??:0
    #1 0x403c43 in main example_UseAfterFree.cc:2
```

```
#2 0x7f7ddabcac4d in __libc_start_main ??:0
==9442== ABORTING
```

If that does not work for you (e.g. your process is sandboxed), you can use a separate script to symbolize the result offline (online symbolization can be force disabled by setting `ASAN_OPTIONS=symbolize=0`):

```
% ASAN_OPTIONS=symbolize=0 ./a.out 2> log
% projects/compiler-rt/lib/asan/scripts/asan_symbolize.py / < log | c++filt
==9442== ERROR: AddressSanitizer heap-use-after-free on address 0x7f7ddab8c084 at pc_
↳ 0x403c8c bp 0x7fff87fb82d0 sp 0x7fff87fb82c8
READ of size 4 at 0x7f7ddab8c084 thread T0
    #0 0x403c8c in main example_UseAfterFree.cc:4
    #1 0x7f7ddabcac4d in __libc_start_main ??:0
...

```

Note that on OS X you may need to run `dsymutil` on your binary to have the file:line info in the AddressSanitizer reports.

Additional Checks

Initialization order checking

AddressSanitizer can optionally detect dynamic initialization order problems, when initialization of globals defined in one translation unit uses globals defined in another translation unit. To enable this check at runtime, you should set environment variable `ASAN_OPTIONS=check_initialization_order=1`.

Note that this option is not supported on OS X.

Memory leak detection

For more information on leak detector in AddressSanitizer, see [LeakSanitizer](#). The leak detection is turned on by default on Linux; however, it is not yet supported on other platforms.

Issue Suppression

AddressSanitizer is not expected to produce false positives. If you see one, look again; most likely it is a true positive!

Suppressing Reports in External Libraries

Runtime interposition allows AddressSanitizer to find bugs in code that is not being recompiled. If you run into an issue in external libraries, we recommend immediately reporting it to the library maintainer so that it gets addressed. However, you can use the following suppression mechanism to unblock yourself and continue on with the testing. This suppression mechanism should only be used for suppressing issues in external code; it does not work on code recompiled with AddressSanitizer. To suppress errors in external libraries, set the `ASAN_OPTIONS` environment variable to point to a suppression file. You can either specify the full path to the file or the path of the file relative to the location of your executable.

```
ASAN_OPTIONS=suppressions=MyASan.supp
```

Use the following format to specify the names of the functions or libraries you want to suppress. You can see these in the error report. Remember that the narrower the scope of the suppression, the more bugs you will be able to catch.

```
interceptor_via_fun:NameOfCFunctionToSuppress
interceptor_via_fun:-[ClassName objcMethodToSuppress:]
interceptor_via_lib:NameOfTheLibraryToSuppress
```

Conditional Compilation with `__has_feature(address_sanitizer)`

In some cases one may need to execute different code depending on whether AddressSanitizer is enabled. `__has_feature` can be used for this purpose.

```
#if defined(__has_feature)
#  if __has_feature(address_sanitizer)
// code that builds only under AddressSanitizer
#  endif
#endif
```

Disabling Instrumentation with `__attribute__((no_sanitize("address")))`

Some code should not be instrumented by AddressSanitizer. One may use the function attribute `__attribute__((no_sanitize("address")))` (which has deprecated synonyms *no_sanitize_address* and *no_address_safety_analysis*) to disable instrumentation of a particular function. This attribute may not be supported by other compilers, so we suggest to use it together with `__has_feature(address_sanitizer)`.

Suppressing Errors in Recompiled Code (Blacklist)

AddressSanitizer supports `src` and `fun` entity types in *Sanitizer special case list*, that can be used to suppress error reports in the specified source files or functions. Additionally, AddressSanitizer introduces `global` and `type` entity types that can be used to suppress error reports for out-of-bound access to globals with certain names and types (you may only specify class or struct types).

You may use an `init` category to suppress reports about initialization-order problems happening in certain source files or with certain global variables.

```
# Suppress error reports for code in a file or in a function:
src:bad_file.cpp
# Ignore all functions with names containing MyFooBar:
fun:*MyFooBar*
# Disable out-of-bound checks for global:
global:bad_array
# Disable out-of-bound checks for global instances of a given class ...
type:Namespace::BadClassName
# ... or a given struct. Use wildcard to deal with anonymous namespace.
type:Namespace2::*::BadStructName
# Disable initialization-order checks for globals:
global:bad_init_global=init
type:*BadInitClassSubstring*=init
src:bad/init/files/*=init
```

Limitations

- AddressSanitizer uses more real memory than a native run. Exact overhead depends on the allocations sizes. The smaller the allocations you make the bigger the overhead is.

- AddressSanitizer uses more stack memory. We have seen up to 3x increase.
- On 64-bit platforms AddressSanitizer maps (but not reserves) 16+ Terabytes of virtual address space. This means that tools like `ulimit` may not work as usually expected.
- Static linking is not supported.

Supported Platforms

AddressSanitizer is supported on:

- Linux i386/x86_64 (tested on Ubuntu 12.04)
- OS X 10.7 - 10.11 (i386/x86_64)
- iOS Simulator
- Android ARM
- FreeBSD i386/x86_64 (tested on FreeBSD 11-current)

Ports to various other platforms are in progress.

Current Status

AddressSanitizer is fully functional on supported platforms starting from LLVM 3.1. The test suite is integrated into CMake build and can be run with `make check-asan` command.

More Information

<http://code.google.com/p/address-sanitizer>

ThreadSanitizer

Introduction

ThreadSanitizer is a tool that detects data races. It consists of a compiler instrumentation module and a run-time library. Typical slowdown introduced by ThreadSanitizer is about **5x-15x**. Typical memory overhead introduced by ThreadSanitizer is about **5x-10x**.

How to build

Build LLVM/Clang with [CMake](#).

Supported Platforms

ThreadSanitizer is supported on Linux x86_64 (tested on Ubuntu 12.04). Support for other 64-bit architectures is possible, contributions are welcome. Support for 32-bit platforms is problematic and is not planned.

Usage

Simply compile and link your program with `-fsanitize=thread`. To get a reasonable performance add `-O1` or higher. Use `-g` to get file names and line numbers in the warning messages.

Example:

```
% cat projects/compiler-rt/lib/tsan/lit_tests/tiny_race.c
#include <pthread.h>
int Global;
void *Thread1(void *x) {
    Global = 42;
    return x;
}
int main() {
    pthread_t t;
    pthread_create(&t, NULL, Thread1, NULL);
    Global = 43;
    pthread_join(t, NULL);
    return Global;
}

$ clang -fsanitize=thread -g -O1 tiny_race.c
```

If a bug is detected, the program will print an error message to `stderr`. Currently, ThreadSanitizer symbolizes its output using an external `addr2line` process (this will be fixed in future).

```
% ./a.out
WARNING: ThreadSanitizer: data race (pid=19219)
  Write of size 4 at 0x7fcf47b21bc0 by thread T1:
    #0 Thread1 tiny_race.c:4 (exe+0x00000000a360)

  Previous write of size 4 at 0x7fcf47b21bc0 by main thread:
    #0 main tiny_race.c:10 (exe+0x00000000a3b4)

  Thread T1 (running) created at:
    #0 pthread_create tsan_interceptors.cc:705 (exe+0x00000000c790)
    #1 main tiny_race.c:9 (exe+0x00000000a3a4)
```

`__has_feature(thread_sanitizer)`

In some cases one may need to execute different code depending on whether ThreadSanitizer is enabled. `__has_feature` can be used for this purpose.

```
#if defined(__has_feature)
#   if __has_feature(thread_sanitizer)
// code that builds only under ThreadSanitizer
#   endif
#endif
```

`__attribute__((no_sanitize_thread))`

Some code should not be instrumented by ThreadSanitizer. One may use the function attribute `no_sanitize_thread` to disable instrumentation of plain (non-atomic) loads/stores in a particular function. ThreadSanitizer still instruments

such functions to avoid false positives and provide meaningful stack traces. This attribute may not be supported by other compilers, so we suggest to use it together with `__has_feature(thread_sanitizer)`.

Blacklist

ThreadSanitizer supports `src` and `fun` entity types in *Sanitizer special case list*, that can be used to suppress data race reports in the specified source files or functions. Unlike functions marked with *no_sanitize_thread* attribute, blacklisted functions are not instrumented at all. This can lead to false positives due to missed synchronization via atomic operations and missed stack frames in reports.

Limitations

- ThreadSanitizer uses more real memory than a native run. At the default settings the memory overhead is 5x plus 1Mb per each thread. Settings with 3x (less accurate analysis) and 9x (more accurate analysis) overhead are also available.
- ThreadSanitizer maps (but does not reserve) a lot of virtual address space. This means that tools like `ulimit` may not work as usually expected.
- Libc/libstdc++ static linking is not supported.
- Non-position-independent executables are not supported. Therefore, the `fsanitize=thread` flag will cause Clang to act as though the `-fPIE` flag had been supplied if compiling without `-fPIC`, and as though the `-pie` flag had been supplied if linking an executable.

Current Status

ThreadSanitizer is in beta stage. It is known to work on large C++ programs using `pthread`s, but we do not promise anything (yet). C++11 threading is supported with `llvm libc++`. The test suite is integrated into CMake build and can be run with `make check-tsan` command.

We are actively working on enhancing the tool — stay tuned. Any help, especially in the form of minimized standalone tests is more than welcome.

More Information

<http://code.google.com/p/thread-sanitizer>.

MemorySanitizer

- *Introduction*
- *How to build*
- *Usage*
 - `__has_feature(memory_sanitizer)`
 - `__attribute__((no_sanitize_memory))`
 - *Blacklist*

- *Report symbolization*
- *Origin Tracking*
- *Handling external code*
- *Supported Platforms*
- *Limitations*
- *Current Status*
- *More Information*

Introduction

MemorySanitizer is a detector of uninitialized reads. It consists of a compiler instrumentation module and a run-time library.

Typical slowdown introduced by MemorySanitizer is **3x**.

How to build

Build LLVM/Clang with [CMake](#).

Usage

Simply compile and link your program with `-fsanitize=memory` flag. The MemorySanitizer run-time library should be linked to the final executable, so make sure to use `clang` (not `ld`) for the final link step. When linking shared libraries, the MemorySanitizer run-time is not linked, so `-Wl,-z,defs` may cause link errors (don't use it with MemorySanitizer). To get a reasonable performance add `-O1` or higher. To get meaningful stack traces in error messages add `-fno-omit-frame-pointer`. To get perfect stack traces you may need to disable inlining (just use `-O1`) and tail call elimination (`-fno-optimize-sibling-calls`).

```
% cat umr.cc
#include <stdio.h>

int main(int argc, char** argv) {
    int* a = new int[10];
    a[5] = 0;
    if (a[argc])
        printf("xx\n");
    return 0;
}

% clang -fsanitize=memory -fno-omit-frame-pointer -g -O2 umr.cc
```

If a bug is detected, the program will print an error message to `stderr` and exit with a non-zero exit code. Currently, MemorySanitizer does not symbolize its output by default, so you may need to use a separate script to symbolize the result offline (this will be fixed in future).

```
% ./a.out
WARNING: MemorySanitizer: use-of-uninitialized-value
#0 0x7f45944b418a in main umr.cc:6
#1 0x7f45938b676c in __libc_start_main libc-start.c:226
```

By default, MemorySanitizer exits on the first detected error.

`__has_feature(memory_sanitizer)`

In some cases one may need to execute different code depending on whether MemorySanitizer is enabled. `__has_feature` can be used for this purpose.

```
#if defined(__has_feature)
#  if __has_feature(memory_sanitizer)
// code that builds only under MemorySanitizer
#  endif
#endif
```

`__attribute__((no_sanitize_memory))`

Some code should not be checked by MemorySanitizer. One may use the function attribute `no_sanitize_memory` to disable uninitialized checks in a particular function. MemorySanitizer may still instrument such functions to avoid false positives. This attribute may not be supported by other compilers, so we suggest to use it together with `__has_feature(memory_sanitizer)`.

Blacklist

MemorySanitizer supports `src` and `fun` entity types in *Sanitizer special case list*, that can be used to relax MemorySanitizer checks for certain source files and functions. All “Use of uninitialized value” warnings will be suppressed and all values loaded from memory will be considered fully initialized.

Report symbolization

MemorySanitizer uses an external symbolizer to print files and line numbers in reports. Make sure that `llvm-symbolizer` binary is in `PATH`, or set environment variable `MSAN_SYMBOLIZER_PATH` to point to it.

Origin Tracking

MemorySanitizer can track origins of uninitialized values, similar to Valgrind’s `-track-origins` option. This feature is enabled by `-fsanitize-memory-track-origins=2` (or simply `-fsanitize-memory-track-origins`) Clang option. With the code from the example above,

```
% cat umr2.cc
#include <stdio.h>

int main(int argc, char** argv) {
    int* a = new int[10];
    a[5] = 0;
    volatile int b = a[argc];
    if (b)
        printf("xx\n");
    return 0;
}

% clang -fsanitize=memory -fsanitize-memory-track-origins=2 -fno-omit-frame-pointer -
  ↪g -O2 umr2.cc
% ./a.out
```

```
WARNING: MemorySanitizer: use-of-uninitialized-value
#0 0x7f7893912f0b in main umr2.cc:7
#1 0x7f789249b76c in __libc_start_main libc-start.c:226

Uninitialized value was stored to memory at
#0 0x7f78938b5c25 in __msan_chain_origin msan.cc:484
#1 0x7f7893912ecd in main umr2.cc:6

Uninitialized value was created by a heap allocation
#0 0x7f7893901cbd in operator new[](unsigned long) msan_new_delete.cc:44
#1 0x7f7893912e06 in main umr2.cc:4
```

By default, MemorySanitizer collects both allocation points and all intermediate stores the uninitialized value went through. Origin tracking has proved to be very useful for debugging MemorySanitizer reports. It slows down program execution by a factor of 1.5x-2x on top of the usual MemorySanitizer slowdown.

Clang option `-fsanitize-memory-track-origins=1` enabled a slightly faster mode when MemorySanitizer collects only allocation points but not intermediate stores.

Handling external code

MemorySanitizer requires that all program code is instrumented. This also includes any libraries that the program depends on, even libc. Failing to achieve this may result in false reports.

Full MemorySanitizer instrumentation is very difficult to achieve. To make it easier, MemorySanitizer runtime library includes 70+ interceptors for the most common libc functions. They make it possible to run MemorySanitizer-instrumented programs linked with uninstrumented libc. For example, the authors were able to bootstrap MemorySanitizer-instrumented Clang compiler by linking it with self-built instrumented libc++ (as a replacement for libstdc++).

Supported Platforms

MemorySanitizer is supported on

- Linux x86_64 (tested on Ubuntu 12.04);

Limitations

- MemorySanitizer uses 2x more real memory than a native run, 3x with origin tracking.
- MemorySanitizer maps (but not reserves) 64 Terabytes of virtual address space. This means that tools like `ulimit` may not work as usually expected.
- Static linking is not supported.
- Non-position-independent executables are not supported. Therefore, the `fsanitize=memory` flag will cause Clang to act as though the `-fPIE` flag had been supplied if compiling without `-fPIC`, and as though the `-pie` flag had been supplied if linking an executable.
- Depending on the version of Linux kernel, running without ASLR may be not supported. Note that GDB disables ASLR by default. To debug instrumented programs, use “set disable-randomization off”.

Current Status

MemorySanitizer is an experimental tool. It is known to work on large real-world programs, like Clang/LLVM itself.

More Information

<http://code.google.com/p/memory-sanitizer>

DataFlowSanitizer

DataFlowSanitizer Design Document

This document sets out the design for DataFlowSanitizer, a general dynamic data flow analysis. Unlike other Sanitizer tools, this tool is not designed to detect a specific class of bugs on its own. Instead, it provides a generic dynamic data flow analysis framework to be used by clients to help detect application-specific issues within their own code.

DataFlowSanitizer is a program instrumentation which can associate a number of taint labels with any data stored in any memory region accessible by the program. The analysis is dynamic, which means that it operates on a running program, and tracks how the labels propagate through that program. The tool shall support a large (>100) number of labels, such that programs which operate on large numbers of data items may be analysed with each data item being tracked separately.

Use Cases

This instrumentation can be used as a tool to help monitor how data flows from a program's inputs (sources) to its outputs (sinks). This has applications from a privacy/security perspective in that one can audit how a sensitive data item is used within a program and ensure it isn't exiting the program anywhere it shouldn't be.

Interface

A number of functions are provided which will create taint labels, attach labels to memory regions and extract the set of labels associated with a specific memory region. These functions are declared in the header file `sanitizer/dfsan_interface.h`.

```
/// Creates and returns a base label with the given description and user data.
dfsan_label dfsan_create_label(const char *desc, void *userdata);

/// Sets the label for each address in [addr,addr+size) to \c label.
void dfsan_set_label(dfsan_label label, void *addr, size_t size);

/// Sets the label for each address in [addr,addr+size) to the union of the
/// current label for that address and \c label.
void dfsan_add_label(dfsan_label label, void *addr, size_t size);

/// Retrieves the label associated with the given data.
///
/// The type of 'data' is arbitrary. The function accepts a value of any type,
/// which can be truncated or extended (implicitly or explicitly) as necessary.
/// The truncation/extension operations will preserve the label of the original
/// value.
dfsan_label dfsan_get_label(long data);

/// Retrieves a pointer to the dfsan_label_info struct for the given label.
const struct dfsan_label_info *dfsan_get_label_info(dfsan_label label);

/// Returns whether the given label label contains the label elem.
int dfsan_has_label(dfsan_label label, dfsan_label elem);
```

```

/// If the given label label contains a label with the description desc, returns
/// that label, else returns 0.
dfsan_label dfsan_has_label_with_desc(dfsan_label label, const char *desc);

```

Taint label representation

As stated above, the tool must track a large number of taint labels. This poses an implementation challenge, as most multiple-label tainting systems assign one label per bit to shadow storage, and union taint labels using a bitwise or operation. This will not scale to clients which use hundreds or thousands of taint labels, as the label union operation becomes $O(n)$ in the number of supported labels, and data associated with it will quickly dominate the live variable set, causing register spills and hampering performance.

Instead, a low overhead approach is proposed which is best-case $O(\log_2 n)$ during execution. The underlying assumption is that the required space of label unions is sparse, which is a reasonable assumption to make given that we are optimizing for the case where applications mostly copy data from one place to another, without often invoking the need for an actual union operation. The representation of a taint label is a 16-bit integer, and new labels are allocated sequentially from a pool. The label identifier 0 is special, and means that the data item is unlabelled.

When a label union operation is requested at a join point (any arithmetic or logical operation with two or more operands, such as addition), the code checks whether a union is required, whether the same union has been requested before, and whether one union label subsumes the other. If so, it returns the previously allocated union label. If not, it allocates a new union label from the same pool used for new labels.

Specifically, the instrumentation pass will insert code like this to decide the union label `lu` for a pair of labels `l1` and `l2`:

```

if (l1 == l2)
    lu = l1;
else
    lu = __dfsan_union(l1, l2);

```

The equality comparison is outlined, to provide an early exit in the common cases where the program is processing unlabelled data, or where the two data items have the same label. `__dfsan_union` is a runtime library function which performs all other union computation.

Further optimizations are possible, for example if `l1` is known at compile time to be zero (e.g. it is derived from a constant), `l2` can be used for `lu`, and vice versa.

Memory layout and label management

The following is the current memory layout for Linux/x86_64:

Start	End	Use
0x700000008000	0x800000000000	application memory
0x200200000000	0x700000008000	unused
0x200000000000	0x200200000000	union table
0x000000010000	0x200000000000	shadow memory
0x000000000000	0x000000010000	reserved by kernel

Each byte of application memory corresponds to two bytes of shadow memory, which are used to store its taint label. As for LLVM SSA registers, we have not found it necessary to associate a label with each byte or bit of data, as some other tools do. Instead, labels are associated directly with registers. Loads will result in a union of all shadow labels corresponding to bytes loaded (which most of the time will be short circuited by the initial comparison) and stores will result in a copy of the label to the shadow of all bytes stored to.

Propagating labels through arguments

In order to propagate labels through function arguments and return values, DataFlowSanitizer changes the ABI of each function in the translation unit. There are currently two supported ABIs:

- **Args** – Argument and return value labels are passed through additional arguments and by modifying the return type.
- **TLS** – Argument and return value labels are passed through TLS variables `__dfsan_arg_tls` and `__dfsan_retval_tls`.

The main advantage of the TLS ABI is that it is more tolerant of ABI mismatches (TLS storage is not shared with any other form of storage, whereas extra arguments may be stored in registers which under the native ABI are not used for parameter passing and thus could contain arbitrary values). On the other hand the args ABI is more efficient and allows ABI mismatches to be more easily identified by checking for nonzero labels in nominally unlabelled programs.

Implementing the ABI list

The ABI list provides a list of functions which conform to the native ABI, each of which is callable from an instrumented program. This is implemented by replacing each reference to a native ABI function with a reference to a function which uses the instrumented ABI. Such functions are automatically-generated wrappers for the native functions. For example, given the ABI list example provided in the user manual, the following wrappers will be generated under the args ABI:

```
define linkonce_odr { i8*, i16 } @"dfsw$malloc"(i64 %0, i16 %1) {
entry:
    %2 = call i8* @malloc(i64 %0)
    %3 = insertvalue { i8*, i16 } undef, i8* %2, 0
    %4 = insertvalue { i8*, i16 } %3, i16 0, 1
    ret { i8*, i16 } %4
}

define linkonce_odr { i32, i16 } @"dfsw$tolower"(i32 %0, i16 %1) {
entry:
    %2 = call i32 @tolower(i32 %0)
    %3 = insertvalue { i32, i16 } undef, i32 %2, 0
    %4 = insertvalue { i32, i16 } %3, i16 %1, 1
    ret { i32, i16 } %4
}

define linkonce_odr { i8*, i16 } @"dfsw$memcpy"(i8* %0, i8* %1, i64 %2, i16 %3, i16
↪%4, i16 %5) {
entry:
    %labelreturn = alloca i16
    %6 = call i8* @__dfsw_memcpy(i8* %0, i8* %1, i64 %2, i16 %3, i16 %4, i16 %5, i16*
↪%labelreturn)
    %7 = load i16* %labelreturn
    %8 = insertvalue { i8*, i16 } undef, i8* %6, 0
    %9 = insertvalue { i8*, i16 } %8, i16 %7, 1
    ret { i8*, i16 } %9
}
```

As an optimization, direct calls to native ABI functions will call the native ABI function directly and the pass will compute the appropriate label internally. This has the advantage of reducing the number of union operations required when the return value label is known to be zero (i.e. discard functions, or functional functions with known unlabelled arguments).

Checking ABI Consistency

DFSan changes the ABI of each function in the module. This makes it possible for a function with the native ABI to be called with the instrumented ABI, or vice versa, thus possibly invoking undefined behavior. A simple way of statically detecting instances of this problem is to prepend the prefix “dfs\$” to the name of each instrumented-ABI function.

This will not catch every such problem; in particular function pointers passed across the instrumented-native barrier cannot be used on the other side. These problems could potentially be caught dynamically.

- *Introduction*
- *Usage*
 - *ABI List*
- *Example*
- *Current status*
- *Design*

Introduction

DataFlowSanitizer is a generalised dynamic data flow analysis.

Unlike other Sanitizer tools, this tool is not designed to detect a specific class of bugs on its own. Instead, it provides a generic dynamic data flow analysis framework to be used by clients to help detect application-specific issues within their own code.

Usage

With no program changes, applying DataFlowSanitizer to a program will not alter its behavior. To use DataFlowSanitizer, the program uses API functions to apply tags to data to cause it to be tracked, and to check the tag of a specific data item. DataFlowSanitizer manages the propagation of tags through the program according to its data flow.

The APIs are defined in the header file `sanitizer/dfsan_interface.h`. For further information about each function, please refer to the header file.

ABI List

DataFlowSanitizer uses a list of functions known as an ABI list to decide whether a call to a specific function should use the operating system’s native ABI or whether it should use a variant of this ABI that also propagates labels through function parameters and return values. The ABI list file also controls how labels are propagated in the former case. DataFlowSanitizer comes with a default ABI list which is intended to eventually cover the glibc library on Linux but it may become necessary for users to extend the ABI list in cases where a particular library or function cannot be instrumented (e.g. because it is implemented in assembly or another language which DataFlowSanitizer does not support) or a function is called from a library or function which cannot be instrumented.

DataFlowSanitizer’s ABI list file is a *Sanitizer special case list*. The pass treats every function in the `uninstrumented` category in the ABI list file as conforming to the native ABI. Unless the ABI list contains additional categories for those functions, a call to one of those functions will produce a warning message, as the labelling behavior of the function is unknown. The other supported categories are `discard`, `functional` and `custom`.

- `discard` – To the extent that this function writes to (user-accessible) memory, it also updates labels in shadow memory (this condition is trivially satisfied for functions which do not write to user-accessible memory). Its return value is unlabelled.
- `functional` – Like `discard`, except that the label of its return value is the union of the label of its arguments.
- `custom` – Instead of calling the function, a custom wrapper `__dfsw_F` is called, where `F` is the name of the function. This function may wrap the original function or provide its own implementation. This category is generally used for uninstrumentable functions which write to user-accessible memory or which have more complex label propagation behavior. The signature of `__dfsw_F` is based on that of `F` with each argument having a label of type `dfsan_label` appended to the argument list. If `F` is of non-void return type a final argument of type `dfsan_label *` is appended to which the custom function can store the label for the return value. For example:

```
void f(int x);
void __dfsw_f(int x, dfsan_label x_label);

void *memcpy(void *dest, const void *src, size_t n);
void *__dfsw_memcpy(void *dest, const void *src, size_t n,
                    dfsan_label dest_label, dfsan_label src_label,
                    dfsan_label n_label, dfsan_label *ret_label);
```

If a function defined in the translation unit being compiled belongs to the uninstrumented category, it will be compiled so as to conform to the native ABI. Its arguments will be assumed to be unlabelled, but it will propagate labels in shadow memory.

For example:

```
# main is called by the C runtime using the native ABI.
fun:main=uninstrumented
fun:main=discard

# malloc only writes to its internal data structures, not user-accessible memory.
fun:malloc=uninstrumented
fun:malloc=discard

# tolower is a pure function.
fun:tolower=uninstrumented
fun:tolower=functional

# memcpy needs to copy the shadow from the source to the destination region.
# This is done in a custom function.
fun:memcpy=uninstrumented
fun:memcpy=custom
```

Example

The following program demonstrates label propagation by checking that the correct labels are propagated.

```
#include <sanitizer/dfsan_interface.h>
#include <assert.h>

int main(void) {
    int i = 1;
    dfsan_label i_label = dfsan_create_label("i", 0);
    dfsan_set_label(i_label, &i, sizeof(i));
```

```
int j = 2;
dfsan_label j_label = dfsan_create_label("j", 0);
dfsan_set_label(j_label, &j, sizeof(j));

int k = 3;
dfsan_label k_label = dfsan_create_label("k", 0);
dfsan_set_label(k_label, &k, sizeof(k));

dfsan_label ij_label = dfsan_get_label(i + j);
assert(dfsan_has_label(ij_label, i_label));
assert(dfsan_has_label(ij_label, j_label));
assert(!dfsan_has_label(ij_label, k_label));

dfsan_label ijk_label = dfsan_get_label(i + j + k);
assert(dfsan_has_label(ijk_label, i_label));
assert(dfsan_has_label(ijk_label, j_label));
assert(dfsan_has_label(ijk_label, k_label));

return 0;
}
```

Current status

DataFlowSanitizer is a work in progress, currently under development for x86_64 Linux.

Design

Please refer to the *design document*.

LeakSanitizer

- *Introduction*
- *Current status*
- *More Information*

Introduction

LeakSanitizer is a run-time memory leak detector. It can be combined with *AddressSanitizer* to get both memory error and leak detection. LeakSanitizer does not introduce any additional slowdown when used in this mode. The LeakSanitizer runtime can also be linked in separately to get leak detection only, at a minimal performance cost.

Current status

LeakSanitizer is turned on by default, but it is only supported on x86_64 Linux.

The combined mode has been tested on fairly large software projects. The stand-alone mode has received much less testing.

There are plans to support LeakSanitizer in *MemorySanitizer* builds.

More Information

<https://code.google.com/p/address-sanitizer/wiki/LeakSanitizer>

SanitizerCoverage

- *Introduction*
- *How to build and run*
- *Postprocessing*
- *How good is the coverage?*
- *Edge coverage*
- *Bitset*
- *Caller-callee coverage*
- *Coverage counters*
- *Output directory*
- *Sudden death*
- *In-process fuzzing*
- *Performance*
- *Why another coverage?*

Introduction

Sanitizer tools have a very simple code coverage tool built in. It allows to get function-level, basic-block-level, and edge-level coverage at a very low cost.

How to build and run

SanitizerCoverage can be used with *AddressSanitizer*, *LeakSanitizer*, *MemorySanitizer*, and *UndefinedBehaviorSanitizer*. In addition to `-fsanitize=`, pass one of the following compile-time flags:

- `-fsanitize-coverage=func` for function-level coverage (very fast).
- `-fsanitize-coverage=bb` for basic-block-level coverage (may add up to 30% **extra** slowdown).
- `-fsanitize-coverage=edge` for edge-level coverage (up to 40% slowdown).

You may also specify `-fsanitize-coverage=indirect-calls` for additional *caller-callee coverage*.

At run time, pass `coverage=1` in `ASAN_OPTIONS`, `LSAN_OPTIONS`, `MSAN_OPTIONS` or `UBSAN_OPTIONS`, as appropriate.

To get *Coverage counters*, add `-fsanitize-coverage=8bit-counters` to one of the above compile-time flags. At runtime, use `*SAN_OPTIONS=coverage=1:coverage_counters=1`.

Example:

```
% cat -n cov.cc
 1  #include <stdio.h>
 2  __attribute__((noinline))
 3  void foo() { printf("foo\n"); }
 4
 5  int main(int argc, char **argv) {
 6      if (argc == 2)
 7          foo();
 8      printf("main\n");
 9  }

% clang++ -g cov.cc -fsanitize=address -fsanitize-coverage=func
% ASAN_OPTIONS=coverage=1 ./a.out; ls -l *sancov
main
-rw-r----- 1 kcc eng 4 Nov 27 12:21 a.out.22673.sancov
% ASAN_OPTIONS=coverage=1 ./a.out foo ; ls -l *sancov
foo
main
-rw-r----- 1 kcc eng 4 Nov 27 12:21 a.out.22673.sancov
-rw-r----- 1 kcc eng 8 Nov 27 12:21 a.out.22679.sancov
```

Every time you run an executable instrumented with SanitizerCoverage one `*.sancov` file is created during the process shutdown. If the executable is dynamically linked against instrumented DSOs, one `*.sancov` file will be also created for every DSO.

Postprocessing

The format of `*.sancov` files is very simple: the first 8 bytes is the magic, one of `0xC0BFFFFFFFFFFFFFFF64` and `0xC0BFFFFFFFFFFFFFFF32`. The last byte of the magic defines the size of the following offsets. The rest of the data is the offsets in the corresponding binary/DSO that were executed during the run.

A simple script `$LLVM/projects/compiler-rt/lib/sanitizer_common/scripts/sancov.py` is provided to dump these offsets.

```
% sancov.py print a.out.22679.sancov a.out.22673.sancov
sancov.py: read 2 PCs from a.out.22679.sancov
sancov.py: read 1 PCs from a.out.22673.sancov
sancov.py: 2 files merged; 2 PCs total
0x465250
0x4652a0
```

You can then filter the output of `sancov.py` through `addr2line --exe ObjectFile` or `llvm-symbolizer --obj ObjectFile` to get file names and line numbers:

```
% sancov.py print a.out.22679.sancov a.out.22673.sancov 2> /dev/null | llvm-
↳symbolizer --obj a.out
cov.cc:3
cov.cc:5
```

How good is the coverage?

It is possible to find out which PCs are not covered, by subtracting the covered set from the set of all instrumented PCs. The latter can be obtained by listing all callsites of `__sanitizer_cov()` in the binary. On Linux, `sancov.py` can do this for you. Just supply the path to binary and a list of covered PCs:

```
% sancov.py print a.out.12345.sancov > covered.txt
sancov.py: read 2 64-bit PCs from a.out.12345.sancov
sancov.py: 1 file merged; 2 PCs total
% sancov.py missing a.out < covered.txt
sancov.py: found 3 instrumented PCs in a.out
sancov.py: read 2 PCs from stdin
sancov.py: 1 PCs missing from coverage
0x4cc61c
```

Edge coverage

Consider this code:

```
void foo(int *a) {
    if (a)
        *a = 0;
}
```

It contains 3 basic blocks, let's name them A, B, C:

```
A
| \
|  \
|   B
|  /
| /
|/
C
```

If blocks A, B, and C are all covered we know for certain that the edges $A \Rightarrow B$ and $B \Rightarrow C$ were executed, but we still don't know if the edge $A \Rightarrow C$ was executed. Such edges of control flow graph are called **critical**. The edge-level coverage (`-fsanitize-coverage=edge`) simply splits all critical edges by introducing new dummy blocks and then instruments those blocks:

```
A
| \
|  \
D   B
|  /
| /
|/
C
```

Bitset

When `coverage_bitset=1` run-time flag is given, the coverage will also be dumped as a bitset (text file with 1 for blocks that have been executed and 0 for blocks that were not).

```
% clang++ -fsanitize=address -fsanitize-coverage=edge cov.cc
% ASAN_OPTIONS="coverage=1:coverage_bitset=1" ./a.out
```

```
main
% ASAN_OPTIONS="coverage=1:coverage_bitset=1" ./a.out 1
foo
main
% head *bitset*
==> a.out.38214.bitset-sancov <==
01101
==> a.out.6128.bitset-sancov <==
11011%
```

For a given executable the length of the bitset is always the same (well, unless `dlopen/dlclose` come into play), so the bitset coverage can be easily used for bitset-based corpus distillation.

Caller-callee coverage

(Experimental!) Every indirect function call is instrumented with a run-time function call that captures caller and callee. At the shutdown time the process dumps a separate file called `caller-callee.PID.sancov` which contains caller/callee pairs as pairs of lines (odd lines are callers, even lines are callees)

```
a.out 0x4a2e0c
a.out 0x4a6510
a.out 0x4a2e0c
a.out 0x4a87f0
```

Current limitations:

- Only the first 14 callees for every caller are recorded, the rest are silently ignored.
- The output format is not very compact since caller and callee may reside in different modules and we need to spell out the module names.
- The routine that dumps the output is not optimized for speed
- Only Linux x86_64 is tested so far.
- Sandboxes are not supported.

Coverage counters

This experimental feature is inspired by [AFL](#)'s coverage instrumentation. With additional compile-time and run-time flags you can get more sensitive coverage information. In addition to boolean values assigned to every basic block (edge) the instrumentation will collect imprecise counters. On exit, every counter will be mapped to a 8-bit bitset representing counter ranges: 1, 2, 3, 4-7, 8-15, 16-31, 32-127, 128+ and those 8-bit bitsets will be dumped to disk.

```
% clang++ -g cov.cc -fsanitize=address -fsanitize-coverage=edge,8bit-counters
% ASAN_OPTIONS="coverage=1:coverage_counters=1" ./a.out
% ls -l *counters-sancov
... a.out.17110.counters-sancov
% xxd *counters-sancov
00000000: 0001 0100 01
```

These counters may also be used for in-process coverage-guided fuzzers. See `include/sanitizer/coverage_interface.h`:


```
// The coverage instrumentation may optionally provide imprecise counters.
// Rather than exposing the counter values to the user we instead map
// the counters to a bitset.
// Every counter is associated with 8 bits in the bitset.
// We define 8 value ranges: 1, 2, 3, 4-7, 8-15, 16-31, 32-127, 128+
// The i-th bit is set to 1 if the counter value is in the i-th range.
// This counter-based coverage implementation is not thread-safe.

// Returns the number of registered coverage counters.
uintptr_t __sanitizer_get_number_of_counters();
// Updates the counter 'bitset', clears the counters and returns the number of
// new bits in 'bitset'.
// If 'bitset' is nullptr, only clears the counters.
// Otherwise 'bitset' should be at least
// __sanitizer_get_number_of_counters bytes long and 8-aligned.
uintptr_t
__sanitizer_update_counter_bitset_and_clear_counters(uint8_t *bitset);
```

Output directory

By default, .sancov files are created in the current working directory. This can be changed with `ASAN_OPTIONS=coverage_dir=/path`:

```
% ASAN_OPTIONS="coverage=1:coverage_dir=/tmp/cov" ./a.out foo
% ls -l /tmp/cov/*sancov
-rw-r----- 1 kcc eng 4 Nov 27 12:21 a.out.22673.sancov
-rw-r----- 1 kcc eng 8 Nov 27 12:21 a.out.22679.sancov
```

Sudden death

Normally, coverage data is collected in memory and saved to disk when the program exits (with an `atexit()` handler), when a `SIGSEGV` is caught, or when `__sanitizer_cov_dump()` is called.

If the program ends with a signal that ASan does not handle (or can not handle at all, like `SIGKILL`), coverage data will be lost. This is a big problem on Android, where `SIGKILL` is a normal way of evicting applications from memory.

With `ASAN_OPTIONS=coverage=1:coverage_direct=1` coverage data is written to a memory-mapped file as soon as it collected.

```
% ASAN_OPTIONS="coverage=1:coverage_direct=1" ./a.out
main
% ls
7036.sancov.map 7036.sancov.raw a.out
% sancov.py rawunpack 7036.sancov.raw
sancov.py: reading map 7036.sancov.map
sancov.py: unpacking 7036.sancov.raw
writing 1 PCs to a.out.7036.sancov
% sancov.py print a.out.7036.sancov
sancov.py: read 1 PCs from a.out.7036.sancov
sancov.py: 1 files merged; 1 PCs total
0x4b2bae
```

Note that on 64-bit platforms, this method writes 2x more data than the default, because it stores full PC values instead of 32-bit offsets.

In-process fuzzing

Coverage data could be useful for fuzzers and sometimes it is preferable to run a fuzzer in the same process as the code being fuzzed (in-process fuzzer).

You can use `__sanitizer_get_total_unique_coverage()` from `<sanitizer/coverage_interface.h>` which returns the number of currently covered entities in the program. This will tell the fuzzer if the coverage has increased after testing every new input.

If a fuzzer finds a bug in the ASan run, you will need to save the reproducer before exiting the process. Use `__asan_set_death_callback` from `<sanitizer/asan_interface.h>` to do that.

An example of such fuzzer can be found in [the LLVM tree](#).

Performance

This coverage implementation is **fast**. With function-level coverage (`-fsanitize-coverage=func`) the overhead is not measurable. With basic-block-level coverage (`-fsanitize-coverage=bb`) the overhead varies between 0 and 25%.

benchmark	cov0	cov1	diff 0-1	cov2	diff 0-2	diff 1-2
400.perlbench	1296.00	1307.00	1.01	1465.00	1.13	1.12
401.bzip2	858.00	854.00	1.00	1010.00	1.18	1.18
403.gcc	613.00	617.00	1.01	683.00	1.11	1.11
429.mcf	605.00	582.00	0.96	610.00	1.01	1.05
445.gobmk	896.00	880.00	0.98	1050.00	1.17	1.19
456.hmmmer	892.00	892.00	1.00	918.00	1.03	1.03
458.sjeng	995.00	1009.00	1.01	1217.00	1.22	1.21
462.libquantum	497.00	492.00	0.99	534.00	1.07	1.09
464.h264ref	1461.00	1467.00	1.00	1543.00	1.06	1.05
471.omnetpp	575.00	590.00	1.03	660.00	1.15	1.12
473.astar	658.00	652.00	0.99	715.00	1.09	1.10
483.xalancbmk	471.00	491.00	1.04	582.00	1.24	1.19
433.milc	616.00	627.00	1.02	627.00	1.02	1.00
444.namd	602.00	601.00	1.00	654.00	1.09	1.09
447.dealII	630.00	634.00	1.01	653.00	1.04	1.03
450.soplex	365.00	368.00	1.01	395.00	1.08	1.07
453.povray	427.00	434.00	1.02	495.00	1.16	1.14
470.lbm	357.00	375.00	1.05	370.00	1.04	0.99
482.sphinx3	927.00	928.00	1.00	1000.00	1.08	1.08

Why another coverage?

Why did we implement yet another code coverage?

- We needed something that is lightning fast, plays well with AddressSanitizer, and does not significantly increase the binary size.
- Traditional coverage implementations based in global counters [suffer from contention on counters](#).

Sanitizer special case list

- [Introduction](#)
- [Goal and usage](#)
- [Example](#)
- [Format](#)

Introduction

This document describes the way to disable or alter the behavior of sanitizer tools for certain source-level entities by providing a special file at compile-time.

Goal and usage

User of sanitizer tools, such as [AddressSanitizer](#), [ThreadSanitizer](#) or [MemorySanitizer](#) may want to disable or alter some checks for certain source-level entities to:

- speedup hot function, which is known to be correct;
- ignore a function that does some low-level magic (e.g. walks through the thread stack, bypassing the frame boundaries);
- ignore a known problem.

To achieve this, user may create a file listing the entities they want to ignore, and pass it to clang at compile-time using `-fsanitize-blacklist` flag. See [Clang Compiler User's Manual](#) for details.

Example

```
$ cat foo.c
#include <stdlib.h>
void bad_foo() {
    int *a = (int*)malloc(40);
    a[10] = 1;
}
int main() { bad_foo(); }
$ cat blacklist.txt
# Ignore reports from bad_foo function.
fun:bad_foo
$ clang -fsanitize=address foo.c ; ./a.out
# AddressSanitizer prints an error report.
$ clang -fsanitize=address -fsanitize-blacklist=blacklist.txt foo.c ; ./a.out
# No error report here.
```

Format

Each line contains an entity type, followed by a colon and a regular expression, specifying the names of the entities, optionally followed by an equals sign and a tool-specific category. Empty lines and lines starting with “#” are ignored. The meaning of `*` in regular expression for entity names is different - it is treated as in shell wildcarding. Two generic entity types are `src` and `fun`, which allow user to add, respectively, source files and functions to special case list. Some sanitizer tools may introduce custom entity types - refer to tool-specific docs.

```
# Lines starting with # are ignored.
# Turn off checks for the source file (use absolute path or path relative
# to the current working directory):
src:/path/to/source/file.c
# Turn off checks for a particular functions (use mangled names):
fun:MyFooBar
fun:_Z8MyFooBarv
# Extended regular expressions are supported:
fun:bad_(foo|bar)
src:bad_source[1-9].c
# Shell like usage of * is supported (* is treated as .*):
src:bad/sources/*
fun:*BadFunction*
# Specific sanitizer tools may introduce categories.
src:/special/path/*=special_sources
```

Control Flow Integrity

Control Flow Integrity Design Documentation

This page documents the design of the *Control Flow Integrity* schemes supported by Clang.

Forward-Edge CFI for Virtual Calls

This scheme works by allocating, for each static type used to make a virtual call, a region of read-only storage in the object file holding a bit vector that maps onto to the region of storage used for those virtual tables. Each set bit in the bit vector corresponds to the [address point](#) for a virtual table compatible with the static type for which the bit vector is being built.

For example, consider the following three C++ classes:

```
struct A {
    virtual void f1();
    virtual void f2();
    virtual void f3();
};

struct B : A {
    virtual void f1();
    virtual void f2();
    virtual void f3();
};

struct C : A {
    virtual void f1();
    virtual void f2();
    virtual void f3();
};
```

The scheme will cause the virtual tables for A, B and C to be laid out consecutively:

Table 4.66: Virtual Table Layout for A, B, C

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A::offset-to-top	&A::rtt	&A::f1	&A::f2	&A::f3	B::offset-to-top	&B::rtt	&B::f1	&B::f2	&B::f3	C::offset-to-top	&C::rtt	&C::f1	&C::f2	&C::f3

The bit vector for static types A, B and C will look like this:

Table 4.67: Bit Vectors for A, B, C

Class	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0
B	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
C	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0

Bit vectors are represented in the object file as byte arrays. By loading from indexed offsets into the byte array and applying a mask, a program can test bits from the bit set with a relatively short instruction sequence. Bit vectors may overlap so long as they use different bits. For the full details, see the [ByteArrayBuilder](#) class.

In this case, assuming A is laid out at offset 0 in bit 0, B at offset 0 in bit 1 and C at offset 0 in bit 2, the byte array would look like this:

```
char bits[] = { 0, 0, 1, 0, 0, 0, 3, 0, 0, 0, 0, 0, 5, 0, 0 };
```

To emit a virtual call, the compiler will assemble code that checks that the object's virtual table pointer is in-bounds and aligned and that the relevant bit is set in the bit vector.

For example on x86 a typical virtual call may look like this:

```
ca7fbb: 48 8b 0f          mov    (%rdi),%rcx
ca7fbe: 48 8d 15 c3 42 fb 07 lea    0x7fb42c3(%rip),%rdx
ca7fc5: 48 89 c8          mov    %rcx,%rax
ca7fc8: 48 29 d0          sub    %rdx,%rax
ca7fcb: 48 c1 c0 3d       rol    $0x3d,%rax
ca7fcf: 48 3d 7f 01 00 00  cmp    $0x17f,%rax
ca7fd5: 0f 87 36 05 00 00  ja     ca8511
ca7fdb: 48 8d 15 c0 0b f7 06 lea    0x6f70bc0(%rip),%rdx
ca7fe2: f6 04 10 10       testb  $0x10, (%rax,%rdx,1)
ca7fe6: 0f 84 25 05 00 00  je     ca8511
ca7fec: ff 91 98 00 00 00  callq  *0x98(%rcx)
[...]
```

```
ca8511: 0f 0b            ud2
```

The compiler relies on co-operation from the linker in order to assemble the bit vectors for the whole program. It currently does this using LLVM's [bit sets](#) mechanism together with link-time optimization.

Optimizations

The scheme as described above is the fully general variant of the scheme. Most of the time we are able to apply one or more of the following optimizations to improve binary size or performance.

In fact, if you try the above example with the current version of the compiler, you will probably find that it will not use the described virtual table layout or machine instructions. Some of the optimizations we are about to introduce cause the compiler to use a different layout or a different sequence of machine instructions.

Stripping Leading/Trailing Zeros in Bit Vectors

If a bit vector contains leading or trailing zeros, we can strip them from the vector. The compiler will emit code to check if the pointer is in range of the region covered by ones, and perform the bit vector check using a truncated version of the bit vector. For example, the bit vectors for our example class hierarchy will be emitted like this:

Table 4.68: Bit Vectors for A, B, C

Class	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A			1	0	0	0	0	1	0	0	0	0	1		
B								1							
C													1		

Short Inline Bit Vectors

If the vector is sufficiently short, we can represent it as an inline constant on x86. This saves us a few instructions when reading the correct element of the bit vector.

If the bit vector fits in 32 bits, the code looks like this:

```
dc2:      48 8b 03      mov    (%rbx),%rax
dc5:      48 8d 15 14 1e 00 00  lea    0x1e14(%rip),%rdx
dcc:      48 89 c1      mov    %rax,%rcx
dcf:      48 29 d1      sub    %rdx,%rcx
dd2:      48 c1 c1 3d      rol    $0x3d,%rcx
dd6:      48 83 f9 03      cmp    $0x3,%rcx
dda:      77 2f      ja     e0b <main+0x9b>
ddc:      ba 09 00 00 00      mov    $0x9,%edx
de1:      0f a3 ca      bt     %ecx,%edx
de4:      73 25      jae    e0b <main+0x9b>
de6:      48 89 df      mov    %rbx,%rdi
de9:      ff 10      callq  *(%rax)
[...]
```

```
e0b:      0f 0b      ud2
```

Or if the bit vector fits in 64 bits:

```
11a6:      48 8b 03      mov    (%rbx),%rax
11a9:      48 8d 15 d0 28 00 00  lea    0x28d0(%rip),%rdx
11b0:      48 89 c1      mov    %rax,%rcx
11b3:      48 29 d1      sub    %rdx,%rcx
11b6:      48 c1 c1 3d      rol    $0x3d,%rcx
11ba:      48 83 f9 2a      cmp    $0x2a,%rcx
11be:      77 35      ja     11f5 <main+0xb5>
11c0:      48 ba 09 00 00 00 00      movabs $0x40000000009,%rdx
11c7:      04 00 00
11ca:      48 0f a3 ca      bt     %rcx,%rdx
11ce:      73 25      jae    11f5 <main+0xb5>
11d0:      48 89 df      mov    %rbx,%rdi
11d3:      ff 10      callq  *(%rax)
[...]
```

```
11f5:      0f 0b      ud2
```

If the bit vector consists of a single bit, there is only one possible virtual table, and the check can consist of a single equality comparison:

9a2:	48 8b 03	mov	(%rbx),%rax
9a5:	48 8d 0d a4 13 00 00	lea	0x13a4(%rip),%rcx
9ac:	48 39 c8	cmp	%rcx,%rax
9af:	75 25	jne	9d6 <main+0x86>
9b1:	48 89 df	mov	%rbx,%rdi
9b4:	ff 10	callq	*(%rax)
[...]			
9d6:	0f 0b	ud2	

Virtual Table Layout

The compiler lays out classes of disjoint hierarchies in separate regions of the object file. At worst, bit vectors in disjoint hierarchies only need to cover their disjoint hierarchy. But the closer that classes in sub-hierarchies are laid out to each other, the smaller the bit vectors for those sub-hierarchies need to be (see “Stripping Leading/Trailing Zeros in Bit Vectors” above). The `GlobalLayoutBuilder` class is responsible for laying out the globals efficiently to minimize the sizes of the underlying bitsets.

Alignment

If all gaps between address points in a particular bit vector are multiples of powers of 2, the compiler can compress the bit vector by strengthening the alignment requirements of the virtual table pointer. For example, given this class hierarchy:

```
struct A {
    virtual void f1();
    virtual void f2();
};

struct B : A {
    virtual void f1();
    virtual void f2();
    virtual void f3();
    virtual void f4();
    virtual void f5();
    virtual void f6();
};

struct C : A {
    virtual void f1();
    virtual void f2();
};
```

The virtual tables will be laid out like this:

Table 4.69: Virtual Table Layout for A, B, C

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A::offset-to-top	&A::rtti	&A::f1	&A::f2	B::offset-to-top	&B::rtti	&B::f1	&B::f2	&B::f3	&B::f4	&B::f5	&B::f6	C::offset-to-top	&C::rtti	&C::f1	&C::f2

Notice that each address point for A is separated by 4 words. This lets us emit a compressed bit vector for A that looks like this:

2	6	10	14
1	1	0	1

At call sites, the compiler will strengthen the alignment requirements by using a different rotate count. For example, on a 64-bit machine where the address points are 4-word aligned (as in A from our example), the `rol` instruction may look like this:

<code>dd2:</code>	<code>48 c1 c1 3b</code>	<code>rol</code>	<code>\$0x3b,%rcx</code>
-------------------	--------------------------	------------------	--------------------------

Padding to Powers of 2

Of course, this alignment scheme works best if the address points are in fact aligned correctly. To make this more likely to happen, we insert padding between virtual tables that in many cases aligns address points to a power of 2. Specifically, our padding aligns virtual tables to the next highest power of 2 bytes; because address points for specific base classes normally appear at fixed offsets within the virtual table, this normally has the effect of aligning the address points as well.

This scheme introduces tradeoffs between decreased space overhead for instructions and bit vectors and increased overhead in the form of padding. We therefore limit the amount of padding so that we align to no more than 128 bytes. This number was found experimentally to provide a good tradeoff.

Eliminating Bit Vector Checks for All-Ones Bit Vectors

If the bit vector is all ones, the bit vector check is redundant; we simply need to check that the address is in range and well aligned. This is more likely to occur if the virtual tables are padded.

- *Introduction*
 - *Forward-Edge CFI for Virtual Calls*
 - * *Performance*
 - *Bad Cast Checking*
 - *Non-Virtual Member Function Call Checking*
 - * *Strictness*
 - *Design*
 - *Publications*

Introduction

Clang includes an implementation of a number of control flow integrity (CFI) schemes, which are designed to abort the program upon detecting certain forms of undefined behavior that can potentially allow attackers to subvert the program's control flow. These schemes have been optimized for performance, allowing developers to enable them in release builds.

To enable Clang's available CFI schemes, use the flag `-fsanitize=cfi`. As currently implemented, CFI relies on link-time optimization (LTO); so it is required to specify `-flto`, and the linker used must support LTO, for example via the [gold plugin](#). To allow the checks to be implemented efficiently, the program must be structured such that certain object files are compiled with CFI enabled, and are statically linked into the program. This may preclude the use of shared libraries in some cases.

Clang currently implements forward-edge CFI for member function calls and bad cast checking. More schemes are under development.

Forward-Edge CFI for Virtual Calls

This scheme checks that virtual calls take place using a vptr of the correct dynamic type; that is, the dynamic type of the called object must be a derived class of the static type of the object used to make the call. This CFI scheme can be enabled on its own using `-fsanitize=cfi-vcall`.

For this scheme to work, all translation units containing the definition of a virtual member function (whether inline or not) must be compiled with `-fsanitize=cfi-vcall` enabled and be statically linked into the program. Classes in the C++ standard library (under namespace `std`) are exempted from checking, and therefore programs may be linked against a pre-built standard library, but this may change in the future.

Performance

A performance overhead of less than 1% has been measured by running the Dromaeo benchmark suite against an instrumented version of the Chromium web browser. Another good performance benchmark for this mechanism is the virtual-call-heavy SPEC 2006 `xalancbnk`.

Note that this scheme has not yet been optimized for binary size; an increase of up to 15% has been observed for Chromium.

Bad Cast Checking

This scheme checks that pointer casts are made to an object of the correct dynamic type; that is, the dynamic type of the object must be a derived class of the pointee type of the cast. The checks are currently only introduced where the class being casted to is a polymorphic class.

Bad casts are not in themselves control flow integrity violations, but they can also create security vulnerabilities, and the implementation uses many of the same mechanisms.

There are two types of bad cast that may be forbidden: bad casts from a base class to a derived class (which can be checked with `-fsanitize=cfi-derived-cast`), and bad casts from a pointer of type `void*` or another unrelated type (which can be checked with `-fsanitize=cfi-unrelated-cast`).

The difference between these two types of casts is that the first is defined by the C++ standard to produce an undefined value, while the second is not in itself undefined behavior (it is well defined to cast the pointer back to its original type).

If a program as a matter of policy forbids the second type of cast, that restriction can normally be enforced. However it may in some cases be necessary for a function to perform a forbidden cast to conform with an external API (e.g. the `allocate` member function of a standard library allocator). Such functions may be blacklisted using a [Sanitizer special case list](#).

For this scheme to work, all translation units containing the definition of a virtual member function (whether inline or not) must be compiled with `-fsanitize=cfi-derived-cast` or `-fsanitize=cfi-unrelated-cast` enabled and be statically linked into the program. Classes in the C++ standard library (under namespace `std`) are exempted from checking, and therefore programs may be linked against a pre-built standard library, but this may change in the future.

Non-Virtual Member Function Call Checking

This scheme checks that non-virtual calls take place using an object of the correct dynamic type; that is, the dynamic type of the called object must be a derived class of the static type of the object used to make the call. The checks are currently only introduced where the object is of a polymorphic class type. This CFI scheme can be enabled on its own using `-fsanitize=cfi-nvcall`.

For this scheme to work, all translation units containing the definition of a virtual member function (whether inline or not) must be compiled with `-fsanitize=cfi-nvcall` enabled and be statically linked into the program. Classes in the C++ standard library (under namespace `std`) are exempted from checking, and therefore programs may be linked against a pre-built standard library, but this may change in the future.

Strictness

If a class has a single non-virtual base and does not introduce or override virtual member functions or fields other than an implicitly defined virtual destructor, it will have the same layout and virtual function semantics as its base. By default, casts to such classes are checked as if they were made to the least derived such class.

Casting an instance of a base class to such a derived class is technically undefined behavior, but it is a relatively common hack for introducing member functions on class instances with specific properties that works under most compilers and should not have security implications, so we allow it by default. It can be disabled with `-fsanitize=cfi-cast-strict`.

Design

Please refer to the *design document*.

Publications

Control-Flow Integrity: Principles, Implementations, and Applications. Martin Abadi, Mihai Budiu, Úlfar Erlingsson, Jay Ligatti.

Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, Geoff Pike.

SafeStack

- *Introduction*
 - *Performance*
 - *Compatibility*
 - * *Known compatibility limitations*
 - *Security*
 - * *Known security limitations*
- *Usage*
 - *Supported Platforms*
 - *Low-level API*
 - * `__has_feature(safe_stack)`
 - * `__attribute__((no_sanitize("safe-stack")))`
 - * `__builtin__get_unsafe_stack_ptr()`
 - * `__builtin__get_unsafe_stack_start()`

- *Design*
 - *Publications*

Introduction

SafeStack is an instrumentation pass that protects programs against attacks based on stack buffer overflows, without introducing any measurable performance overhead. It works by separating the program stack into two distinct regions: the safe stack and the unsafe stack. The safe stack stores return addresses, register spills, and local variables that are always accessed in a safe way, while the unsafe stack stores everything else. This separation ensures that buffer overflows on the unsafe stack cannot be used to overwrite anything on the safe stack.

SafeStack is a part of the [Code-Pointer Integrity \(CPI\) Project](#).

Performance

The performance overhead of the SafeStack instrumentation is less than 0.1% on average across a variety of benchmarks (see the [Code-Pointer Integrity](#) paper for details). This is mainly because most small functions do not have any variables that require the unsafe stack and, hence, do not need unsafe stack frames to be created. The cost of creating unsafe stack frames for large functions is amortized by the cost of executing the function.

In some cases, SafeStack actually improves the performance. Objects that end up being moved to the unsafe stack are usually large arrays or variables that are used through multiple stack frames. Moving such objects away from the safe stack increases the locality of frequently accessed values on the stack, such as register spills, return addresses, and small local variables.

Compatibility

Most programs, static libraries, or individual files can be compiled with SafeStack as is. SafeStack requires basic runtime support, which, on most platforms, is implemented as a compiler-rt library that is automatically linked in when the program is compiled with SafeStack.

Linking a DSO with SafeStack is not currently supported.

Known compatibility limitations

Certain code that relies on low-level stack manipulations requires adaption to work with SafeStack. One example is mark-and-sweep garbage collection implementations for C/C++ (e.g., Oilpan in chromium/blink), which must be changed to look for the live pointers on both safe and unsafe stacks.

SafeStack supports linking statically modules that are compiled with and without SafeStack. An executable compiled with SafeStack can load dynamic libraries that are not compiled with SafeStack. At the moment, compiling dynamic libraries with SafeStack is not supported.

Signal handlers that use `sigaltstack()` must not use the unsafe stack (see `__attribute__((no_sanitize("safe-stack")))` below).

Programs that use APIs from `ucontext.h` are not supported yet.

Security

SafeStack protects return addresses, spilled registers and local variables that are always accessed in a safe way by separating them in a dedicated safe stack region. The safe stack is automatically protected against stack-based buffer overflows, since it is disjoint from the unsafe stack in memory, and it itself is always accessed in a safe way. In the current implementation, the safe stack is protected against arbitrary memory write vulnerabilities through randomization and information hiding: the safe stack is allocated at a random address and the instrumentation ensures that no pointers to the safe stack are ever stored outside of the safe stack itself (see limitations below).

Known security limitations

A complete protection against control-flow hijack attacks requires combining SafeStack with another mechanism that enforces the integrity of code pointers that are stored on the heap or the unsafe stack, such as [CPI](#), or a forward-edge control flow integrity mechanism that enforces correct calling conventions at indirect call sites, such as [IFCC](#) with arity checks. Clang has control-flow integrity protection scheme for [C++ virtual calls](#), but not non-virtual indirect calls. With SafeStack alone, an attacker can overwrite a function pointer on the heap or the unsafe stack and cause a program to call arbitrary location, which in turn might enable stack pivoting and return-oriented programming.

In its current implementation, SafeStack provides precise protection against stack-based buffer overflows, but protection against arbitrary memory write vulnerabilities is probabilistic and relies on randomization and information hiding. The randomization is currently based on system-enforced ASLR and shares its known security limitations. The safe stack pointer hiding is not perfect yet either: system library functions such as `swapcontext`, exception handling mechanisms, intrinsics such as `__builtin_frame_address`, or low-level bugs in runtime support could leak the safe stack pointer. In the future, such leaks could be detected by static or dynamic analysis tools and prevented by adjusting such functions to either encrypt the stack pointer when storing it in the heap (as already done e.g., by `setjmp/longjmp` implementation in `glibc`), or store it in a safe region instead.

The [CPI paper](#) describes two alternative, stronger safe stack protection mechanisms, that rely on software fault isolation, or hardware segmentation (as available on x86-32 and some x86-64 CPUs).

At the moment, SafeStack assumes that the compiler's implementation is correct. This has not been verified except through manual code inspection, and could always regress in the future. It's therefore desirable to have a separate static or dynamic binary verification tool that would check the correctness of the SafeStack instrumentation in final binaries.

Usage

To enable SafeStack, just pass `-fsanitize=safe-stack` flag to both compile and link command lines.

Supported Platforms

SafeStack was tested on Linux, FreeBSD and MacOSX.

Low-level API

`__has_feature(safe_stack)`

In some rare cases one may need to execute different code depending on whether SafeStack is enabled. The macro `__has_feature(safe_stack)` can be used for this purpose.

```
#if __has_feature(safe_stack)
// code that builds only under SafeStack
#endif
```

`__attribute__((no_sanitize("safe-stack")))`

Use `__attribute__((no_sanitize("safe-stack")))` on a function declaration to specify that the safe stack instrumentation should not be applied to that function, even if enabled globally (see `-fsanitize=safe-stack` flag). This attribute may be required for functions that make assumptions about the exact layout of their stack frames.

All local variables in functions with this attribute will be stored on the safe stack. The safe stack remains unprotected against memory errors when accessing these variables, so extra care must be taken to manually ensure that all such accesses are safe. Furthermore, the addresses of such local variables should never be stored on the heap, as it would leak the location of the SafeStack.

`__builtin__get_unsafe_stack_ptr()`

This builtin function returns current unsafe stack pointer of the current thread.

`__builtin__get_unsafe_stack_start()`

This builtin function returns a pointer to the start of the unsafe stack of the current thread.

Design

Please refer to the [Code-Pointer Integrity](#) project page for more information about the design of the SafeStack and its related technologies.

Publications

[Code-Pointer Integrity](#). Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, R. Sekar, Dawn Song. USENIX Symposium on Operating Systems Design and Implementation ([OSDI](#)), Broomfield, CO, October 2014

Modules

- *Introduction*
 - *Problems with the current model*
 - *Semantic import*
 - *Problems modules do not solve*
- *Using Modules*
 - *Objective-C Import declaration*

- *Includes as imports*
 - *Module maps*
 - *Compilation model*
 - *Command-line parameters*
- *Module Semantics*
 - *Macros*
- *Module Map Language*
 - *Lexical structure*
 - *Module map file*
 - *Module declaration*
 - * *Requires declaration*
 - * *Header declaration*
 - * *Umbrella directory declaration*
 - * *Submodule declaration*
 - * *Export declaration*
 - * *Use declaration*
 - * *Link declaration*
 - * *Configuration macros declaration*
 - * *Conflict declarations*
 - *Attributes*
 - *Private Module Map Files*
- *Modularizing a Platform*
- *Future Directions*
- *Where To Learn More About Modules*

Introduction

Most software is built using a number of software libraries, including libraries supplied by the platform, internal libraries built as part of the software itself to provide structure, and third-party libraries. For each library, one needs to access both its interface (API) and its implementation. In the C family of languages, the interface to a library is accessed by including the appropriate header files(s):

```
#include <SomeLib.h>
```

The implementation is handled separately by linking against the appropriate library. For example, by passing `-lSomeLib` to the linker.

Modules provide an alternative, simpler way to use software libraries that provides better compile-time scalability and eliminates many of the problems inherent to using the C preprocessor to access the API of a library.

Problems with the current model

The `#include` mechanism provided by the C preprocessor is a very poor way to access the API of a library, for a number of reasons:

- **Compile-time scalability:** Each time a header is included, the compiler must preprocess and parse the text in that header and every header it includes, transitively. This process must be repeated for every translation unit in the application, which involves a huge amount of redundant work. In a project with N translation units and M headers included in each translation unit, the compiler is performing $M \times N$ work even though most of the M headers are shared among multiple translation units. C++ is particularly bad, because the compilation model for templates forces a huge amount of code into headers.
- **Fragility:** `#include` directives are treated as textual inclusion by the preprocessor, and are therefore subject to any active macro definitions at the time of inclusion. If any of the active macro definitions happens to collide with a name in the library, it can break the library API or cause compilation failures in the library header itself. For an extreme example, `#define std "The C++ Standard"` and then include a standard library header: the result is a horrific cascade of failures in the C++ Standard Library's implementation. More subtle real-world problems occur when the headers for two different libraries interact due to macro collisions, and users are forced to reorder `#include` directives or introduce `#undef` directives to break the (unintended) dependency.
- **Conventional workarounds:** C programmers have adopted a number of conventions to work around the fragility of the C preprocessor model. Include guards, for example, are required for the vast majority of headers to ensure that multiple inclusion doesn't break the compile. Macro names are written with `LONG_PREFIXED_UPPERCASE_IDENTIFIERS` to avoid collisions, and some library/framework developers even use `__underscored` names in headers to avoid collisions with "normal" names that (by convention) shouldn't even be macros. These conventions are a barrier to entry for developers coming from non-C languages, are boilerplate for more experienced developers, and make our headers far uglier than they should be.
- **Tool confusion:** In a C-based language, it is hard to build tools that work well with software libraries, because the boundaries of the libraries are not clear. Which headers belong to a particular library, and in what order should those headers be included to guarantee that they compile correctly? Are the headers C, C++, Objective-C++, or one of the variants of these languages? What declarations in those headers are actually meant to be part of the API, and what declarations are present only because they had to be written as part of the header file?

Semantic import

Modules improve access to the API of software libraries by replacing the textual preprocessor inclusion model with a more robust, more efficient semantic model. From the user's perspective, the code looks only slightly different, because one uses an `import` declaration rather than a `#include` preprocessor directive:

```
import std.io; // pseudo-code; see below for syntax discussion
```

However, this module import behaves quite differently from the corresponding `#include <stdio.h>`: when the compiler sees the module import above, it loads a binary representation of the `std.io` module and makes its API available to the application directly. Preprocessor definitions that precede the import declaration have no impact on the API provided by `std.io`, because the module itself was compiled as a separate, standalone module. Additionally, any linker flags required to use the `std.io` module will automatically be provided when the module is imported¹. This semantic import model addresses many of the problems of the preprocessor inclusion model:

- **Compile-time scalability:** The `std.io` module is only compiled once, and importing the module into a translation unit is a constant-time operation (independent of module system). Thus, the API of each software library is only parsed once, reducing the $M \times N$ compilation problem to an $M + N$ problem.
- **Fragility:** Each module is parsed as a standalone entity, so it has a consistent preprocessor environment. This completely eliminates the need for `__underscored` names and similarly defensive tricks. Moreover, the

¹ Automatic linking against the libraries of modules requires specific linker support, which is not widely available.

current preprocessor definitions when an import declaration is encountered are ignored, so one software library can not affect how another software library is compiled, eliminating include-order dependencies.

- **Tool confusion:** Modules describe the API of software libraries, and tools can reason about and present a module as a representation of that API. Because modules can only be built standalone, tools can rely on the module definition to ensure that they get the complete API for the library. Moreover, modules can specify which languages they work with, so, e.g., one can not accidentally attempt to load a C++ module into a C program.

Problems modules do not solve

Many programming languages have a module or package system, and because of the variety of features provided by these languages it is important to define what modules do *not* do. In particular, all of the following are considered out-of-scope for modules:

- **Rewrite the world's code:** It is not realistic to require applications or software libraries to make drastic or non-backward-compatible changes, nor is it feasible to completely eliminate headers. Modules must interoperate with existing software libraries and allow a gradual transition.
- **Versioning:** Modules have no notion of version information. Programmers must still rely on the existing versioning mechanisms of the underlying language (if any exist) to version software libraries.
- **Namespaces:** Unlike in some languages, modules do not imply any notion of namespaces. Thus, a struct declared in one module will still conflict with a struct of the same name declared in a different module, just as they would if declared in two different headers. This aspect is important for backward compatibility, because (for example) the mangled names of entities in software libraries must not change when introducing modules.
- **Binary distribution of modules:** Headers (particularly C++ headers) expose the full complexity of the language. Maintaining a stable binary module format across architectures, compiler versions, and compiler vendors is technically infeasible.

Using Modules

To enable modules, pass the command-line flag `-fmodules`. This will make any modules-enabled software libraries available as modules as well as introducing any modules-specific syntax. Additional *command-line parameters* are described in a separate section later.

Objective-C Import declaration

Objective-C provides syntax for importing a module via an *@import declaration*, which imports the named module:

```
@import std;
```

The `@import` declaration above imports the entire contents of the `std` module (which would contain, e.g., the entire C or C++ standard library) and make its API available within the current translation unit. To import only part of a module, one may use dot syntax to specific a particular submodule, e.g.,

```
@import std.io;
```

Redundant import declarations are ignored, and one is free to import modules at any point within the translation unit, so long as the import declaration is at global scope.

At present, there is no C or C++ syntax for import declarations. Clang will track the modules proposal in the C++ committee. See the section *Includes as imports* to see how modules get imported today.

Includes as imports

The primary user-level feature of modules is the import operation, which provides access to the API of software libraries. However, today's programs make extensive use of `#include`, and it is unrealistic to assume that all of this code will change overnight. Instead, modules automatically translate `#include` directives into the corresponding module import. For example, the include directive

```
#include <stdio.h>
```

will be automatically mapped to an import of the module `std.io`. Even with specific import syntax in the language, this particular feature is important for both adoption and backward compatibility: automatic translation of `#include` to `import` allows an application to get the benefits of modules (for all modules-enabled libraries) without any changes to the application itself. Thus, users can easily use modules with one compiler while falling back to the preprocessor-inclusion mechanism with other compilers.

Note: The automatic mapping of `#include` to `import` also solves an implementation problem: importing a module with a definition of some entity (say, a `struct Point`) and then parsing a header containing another definition of `struct Point` would cause a redefinition error, even if it is the same `struct Point`. By mapping `#include` to `import`, the compiler can guarantee that it always sees just the already-parsed definition from the module.

While building a module, `#include_next` is also supported, with one caveat. The usual behavior of `#include_next` is to search for the specified filename in the list of include paths, starting from the path *after* the one in which the current file was found. Because files listed in module maps are not found through include paths, a different strategy is used for `#include_next` directives in such files: the list of include paths is searched for the specified header name, to find the first include path that would refer to the current file. `#include_next` is interpreted as if the current file had been found in that path. If this search finds a file named by a module map, the `#include_next` directive is translated into an import, just like for a `#include` directive.“

Module maps

The crucial link between modules and headers is described by a *module map*, which describes how a collection of existing headers maps on to the (logical) structure of a module. For example, one could imagine a module `std` covering the C standard library. Each of the C standard library headers (`<stdio.h>`, `<stdlib.h>`, `<math.h>`, etc.) would contribute to the `std` module, by placing their respective APIs into the corresponding submodule (`std.io`, `std.lib`, `std.math`, etc.). Having a list of the headers that are part of the `std` module allows the compiler to build the `std` module as a standalone entity, and having the mapping from header names to (sub)modules allows the automatic translation of `#include` directives to module imports.

Module maps are specified as separate files (each named `module.modulemap`) alongside the headers they describe, which allows them to be added to existing software libraries without having to change the library headers themselves (in most cases²). The actual *Module map language* is described in a later section.

Note: To actually see any benefits from modules, one first has to introduce module maps for the underlying C standard library and the libraries and headers on which it depends. The section *Modularizing a Platform* describes the steps one must take to write these module maps.

One can use module maps without modules to check the integrity of the use of header files. To do this, use the `-fimplicit-module-maps` option instead of the `-fmodules` option, or use `-fmodule-map-file=` option to explicitly specify the module map files to load.

² There are certain anti-patterns that occur in headers, particularly system headers, that cause problems for modules. The section *Modularizing a Platform* describes some of them.

Compilation model

The binary representation of modules is automatically generated by the compiler on an as-needed basis. When a module is imported (e.g., by an `#include` of one of the module's headers), the compiler will spawn a second instance of itself³, with a fresh preprocessing context⁴, to parse just the headers in that module. The resulting Abstract Syntax Tree (AST) is then persisted into the binary representation of the module that is then loaded into translation unit where the module import was encountered.

The binary representation of modules is persisted in the *module cache*. Imports of a module will first query the module cache and, if a binary representation of the required module is already available, will load that representation directly. Thus, a module's headers will only be parsed once per language configuration, rather than once per translation unit that uses the module.

Modules maintain references to each of the headers that were part of the module build. If any of those headers changes, or if any of the modules on which a module depends change, then the module will be (automatically) recompiled. The process should never require any user intervention.

Command-line parameters

- fmodules** Enable the modules feature.
- fimplicit-module-maps** Enable implicit search for module map files named `module.modulemap` and similar. This option is implied by `-fmodules`. If this is disabled with `-fno-implicit-module-maps`, module map files will only be loaded if they are explicitly specified via `-fmodule-map-file` or transitively used by another module map file.
- fmodules-cache-path=<directory>** Specify the path to the modules cache. If not provided, Clang will select a system-appropriate default.
- fno-autolink** Disable automatic linking against the libraries associated with imported modules.
- fmodules-ignore-macro=macroname** Instruct modules to ignore the named macro when selecting an appropriate module variant. Use this for macros defined on the command line that don't affect how modules are built, to improve sharing of compiled module files.
- fmodules-prune-interval=seconds** Specify the minimum delay (in seconds) between attempts to prune the module cache. Module cache pruning attempts to clear out old, unused module files so that the module cache itself does not grow without bound. The default delay is large (604,800 seconds, or 7 days) because this is an expensive operation. Set this value to 0 to turn off pruning.
- fmodules-prune-after=seconds** Specify the minimum time (in seconds) for which a file in the module cache must be unused (according to access time) before module pruning will remove it. The default delay is large (2,678,400 seconds, or 31 days) to avoid excessive module rebuilding.
- module-file-info <module file name>** Debugging aid that prints information about a given module file (with a `.pcm` extension), including the language and preprocessor options that particular module variant was built with.
- fmodules-decluse** Enable checking of module `use` declarations.
- fmodule-name=module-id** Consider a source file as a part of the given module.
- fmodule-map-file=<file>** Load the given module map file if a header from its directory or one of its subdirectories is loaded.

³ The second instance is actually a new thread within the current process, not a separate process. However, the original compiler instance is blocked on the execution of this thread.

⁴ The preprocessing context in which the modules are parsed is actually dependent on the command-line options provided to the compiler, including the language dialect and any `-D` options. However, the compiled modules for different command-line options are kept distinct, and any preprocessor directives that occur within the translation unit are ignored. See the section on the *Configuration macros declaration* for more information.

-fmodules-search-all If a symbol is not found, search modules referenced in the current module maps but not imported for symbols, so the error message can reference the module by name. Note that if the global module index has not been built before, this might take some time as it needs to build all the modules. Note that this option doesn't apply in module builds, to avoid the recursion.

-fno-implicit-modules All modules used by the build must be specified with `-fmodule-file`.

-fmodule-file=<file> Load the given precompiled module file.

Module Semantics

Modules are modeled as if each submodule were a separate translation unit, and a module import makes names from the other translation unit visible. Each submodule starts with a new preprocessor state and an empty translation unit.

Note: This behavior is currently only approximated when building a module with submodules. Entities within a submodule that has already been built are visible when building later submodules in that module. This can lead to fragile modules that depend on the build order used for the submodules of the module, and should not be relied upon. This behavior is subject to change.

As an example, in C, this implies that if two structs are defined in different submodules with the same name, those two types are distinct types (but may be *compatible* types if their definitions match. In C++, two structs defined with the same name in different submodules are the *same* type, and must be equivalent under C++'s One Definition Rule.

Note: Clang currently only performs minimal checking for violations of the One Definition Rule.

If any submodule of a module is imported into any part of a program, the entire top-level module is considered to be part of the program. As a consequence of this, Clang may diagnose conflicts between an entity declared in an unimported submodule and an entity declared in the current translation unit, and Clang may inline or devirtualize based on knowledge from unimported submodules.

Macros

The C and C++ preprocessor assumes that the input text is a single linear buffer, but with modules this is not the case. It is possible to import two modules that have conflicting definitions for a macro (or where one `#defines` a macro and the other `#undefines` it). The rules for handling macro definitions in the presence of modules are as follows:

- Each definition and undefinition of a macro is considered to be a distinct entity.
- Such entities are *visible* if they are from the current submodule or translation unit, or if they were exported from a submodule that has been imported.
- A `#define X` or `#undef X` directive *overrides* all definitions of X that are visible at the point of the directive.
- A `#define` or `#undef` directive is *active* if it is visible and no visible directive overrides it.
- A set of macro directives is *consistent* if it consists of only `#undef` directives, or if all `#define` directives in the set define the macro name to the same sequence of tokens (following the usual rules for macro redefinitions).
- If a macro name is used and the set of active directives is not consistent, the program is ill-formed. Otherwise, the (unique) meaning of the macro name is used.

For example, suppose:

- `<stdio.h>` defines a macro `getc` (and exports its `#define`)
- `<cstdio>` imports the `<stdio.h>` module and undefines the macro (and exports its `#undef`)

The `#undef` overrides the `#define`, and a source file that imports both modules *in any order* will not see `getc` defined as a macro.

Module Map Language

Warning: The module map language is not currently guaranteed to be stable between major revisions of Clang.

The module map language describes the mapping from header files to the logical structure of modules. To enable support for using a library as a module, one must write a `module.modulemap` file for that library. The `module.modulemap` file is placed alongside the header files themselves, and is written in the module map language described below.

Note: For compatibility with previous releases, if a module map file named `module.modulemap` is not found, Clang will also search for a file named `module.map`. This behavior is deprecated and we plan to eventually remove it.

As an example, the module map file for the C standard library might look a bit like this:

```
module std [system] [extern_c] {
  module assert {
    textual header "assert.h"
    header "bits/assert-decls.h"
    export *
  }

  module complex {
    header "complex.h"
    export *
  }

  module ctype {
    header "ctype.h"
    export *
  }

  module errno {
    header "errno.h"
    header "sys/errno.h"
    export *
  }

  module fenv {
    header "fenv.h"
    export *
  }

  // ...more headers follow...
}
```

Here, the top-level module `std` encompasses the whole C standard library. It has a number of submodules containing different parts of the standard library: `complex` for complex numbers, `ctype` for character types, etc. Each submodule lists one or more headers that provide the contents for that submodule. Finally, the `export *` command specifies that anything included by that submodule will be automatically re-exported.

Lexical structure

Module map files use a simplified form of the C99 lexer, with the same rules for identifiers, tokens, string literals, `/*` and `/**` comments. The module map language has the following reserved words; all other C identifiers are valid identifiers.

```
config_macros  export      private
conflict       framework  requires
exclude        header     textual
explicit       link       umbrella
extern         module     use
```

Module map file

A module map file consists of a series of module declarations:

```
module-map-file:
    module-declaration*
```

Within a module map file, modules are referred to by a *module-id*, which uses periods to separate each part of a module's name:

```
module-id:
    identifier ('.' identifier)*
```

Module declaration

A module declaration describes a module, including the headers that contribute to that module, its submodules, and other aspects of the module.

```
module-declaration:
    explicit$ _opt$ framework$ _opt$ module module-id attributes$ _opt$ '{' module-
    ↪member* '}'
    extern module module-id string-literal
```

The *module-id* should consist of only a single *identifier*, which provides the name of the module being defined. Each module shall have a single definition.

The `explicit` qualifier can only be applied to a submodule, i.e., a module that is nested within another module. The contents of explicit submodules are only made available when the submodule itself was explicitly named in an import declaration or was re-exported from an imported module.

The `framework` qualifier specifies that this module corresponds to a Darwin-style framework. A Darwin-style framework (used primarily on Mac OS X and iOS) is contained entirely in directory `Name.framework`, where `Name` is the name of the framework (and, therefore, the name of the module). That directory has the following layout:

<code>Name.framework/</code>	
<code>Modules/module.modulemap</code>	Module map for the framework
<code>Headers/</code>	Subdirectory containing framework headers
<code>Frameworks/</code>	Subdirectory containing embedded frameworks
<code>Resources/</code>	Subdirectory containing additional resources
<code>Name</code>	Symbolic link to the shared library for the framework

The `system` attribute specifies that the module is a system module. When a system module is rebuilt, all of the module's headers will be considered system headers, which suppresses warnings. This is equivalent to placing `#pragma GCC system_header` in each of the module's headers. The form of attributes is described in the section [Attributes](#), below.

The `extern_c` attribute specifies that the module contains C code that can be used from within C++. When such a module is built for use in C++ code, all of the module's headers will be treated as if they were contained within an implicit `extern "C"` block. An import for a module with this attribute can appear within an `extern "C"` block. No other restrictions are lifted, however: the module currently cannot be imported within an `extern "C"` block in a namespace.

Modules can have a number of different kinds of members, each of which is described below:

```
module-member:
  requires-declaration
  header-declaration
  umbrella-dir-declaration
  submodule-declaration
  export-declaration
  use-declaration
  link-declaration
  config-macros-declaration
  conflict-declaration
```

An `extern` module references a module defined by the *module-id* in a file given by the *string-literal*. The file can be referenced either by an absolute path or by a path relative to the current map file.

Requires declaration

A *requires-declaration* specifies the requirements that an importing translation unit must satisfy to use the module.

```
requires-declaration:
  requires feature-list

feature-list:
  feature (',' feature)*

feature:
  !$_opt$ identifier
```

The requirements clause allows specific modules or submodules to specify that they are only accessible with certain language dialects or on certain platforms. The feature list is a set of identifiers, defined below. If any of the features is not available in a given translation unit, that translation unit shall not import the module. The optional `!` indicates that a feature is incompatible with the module.

The following features are defined:

altivec The target supports AltiVec.

blocks The “blocks” language feature is available.

cplusplus C++ support is available.

cplusplus11 C++11 support is available.

objc Objective-C support is available.

objc_arc Objective-C Automatic Reference Counting (ARC) is available

opencl OpenCL is available

tls Thread local storage is available.

target feature A specific target feature (e.g., `sse4`, `avx`, `neon`) is available.

Example: The `std` module can be extended to also include C++ and C++11 headers using a *requires-declaration*:

```

module std {
    // C standard library...

    module vector {
        requires cplusplus
        header "vector"
    }

    module type_traits {
        requires cplusplus11
        header "type_traits"
    }
}

```

Header declaration

A header declaration specifies that a particular header is associated with the enclosing module.

```

header-declaration:
    private$_opt$ textual$_opt$ header string-literal
    umbrella header string-literal
    exclude header string-literal

```

A header declaration that does not contain `exclude` nor `textual` specifies a header that contributes to the enclosing module. Specifically, when the module is built, the named header will be parsed and its declarations will be (logically) placed into the enclosing submodule.

A header with the `umbrella` specifier is called an umbrella header. An umbrella header includes all of the headers within its directory (and any subdirectories), and is typically used (in the `#include` world) to easily access the full API provided by a particular library. With modules, an umbrella header is a convenient shortcut that eliminates the need to write out header declarations for every library header. A given directory can only contain a single umbrella header.

Note: Any headers not included by the umbrella header should have explicit header declarations. Use the `-Wincomplete-umbrella` warning option to ask Clang to complain about headers not covered by the umbrella header or the module map.

A header with the `private` specifier may not be included from outside the module itself.

A header with the `textual` specifier will not be compiled when the module is built, and will be textually included if it is named by a `#include` directive. However, it is considered to be part of the module for the purpose of checking *use-declarations*, and must still be a lexically-valid header file. In the future, we intend to pre-tokenize such headers and include the token sequence within the prebuilt module representation.

A header with the `exclude` specifier is excluded from the module. It will not be included when the module is built, nor will it be considered to be part of the module, even if an umbrella header or directory would otherwise make it part of the module.

Example: The C header `assert.h` is an excellent candidate for a textual header, because it is meant to be included multiple times (possibly with different `NDEBUG` settings). However, declarations within it should typically be split into a separate modular header.

```

module std [system] {
    textual header "assert.h"
}

```

A given header shall not be referenced by more than one *header-declaration*.

Umbrella directory declaration

An umbrella directory declaration specifies that all of the headers in the specified directory should be included within the module.

```
umbrella-dir-declaration:  
    umbrella string-literal
```

The *string-literal* refers to a directory. When the module is built, all of the header files in that directory (and its subdirectories) are included in the module.

An *umbrella-dir-declaration* shall not refer to the same directory as the location of an umbrella *header-declaration*. In other words, only a single kind of umbrella can be specified for a given directory.

Note: Umbrella directories are useful for libraries that have a large number of headers but do not have an umbrella header.

Submodule declaration

Submodule declarations describe modules that are nested within their enclosing module.

```
submodule-declaration:  
    module-declaration  
    inferred-submodule-declaration
```

A *submodule-declaration* that is a *module-declaration* is a nested module. If the *module-declaration* has a `framework` specifier, the enclosing module shall have a `framework` specifier; the submodule's contents shall be contained within the subdirectory `Frameworks/SubName.framework`, where `SubName` is the name of the submodule.

A *submodule-declaration* that is an *inferred-submodule-declaration* describes a set of submodules that correspond to any headers that are part of the module but are not explicitly described by a *header-declaration*.

```
inferred-submodule-declaration:  
    explicit$_opt$ framework$_opt$ module '*' attributes$_opt$ '{' inferred-  
    ↳ submodule-member* '}'
```

```
inferred-submodule-member:  
    export '*'
```

A module containing an *inferred-submodule-declaration* shall have either an umbrella header or an umbrella directory. The headers to which the *inferred-submodule-declaration* applies are exactly those headers included by the umbrella header (transitively) or included in the module because they reside within the umbrella directory (or its subdirectories).

For each header included by the umbrella header or in the umbrella directory that is not named by a *header-declaration*, a module declaration is implicitly generated from the *inferred-submodule-declaration*. The module will:

- Have the same name as the header (without the file extension)
- Have the `explicit` specifier, if the *inferred-submodule-declaration* has the `explicit` specifier
- Have the `framework` specifier, if the *inferred-submodule-declaration* has the `framework` specifier
- Have the attributes specified by the *inferred-submodule-declaration*

- Contain a single *header-declaration* naming that header
- Contain a single *export-declaration* `export *`, if the *inferred-submodule-declaration* contains the *inferred-submodule-member* `export *`

Example: If the subdirectory “MyLib” contains the headers `A.h` and `B.h`, then the following module map:

```
module MyLib {
  umbrella "MyLib"
  explicit module * {
    export *
  }
}
```

is equivalent to the (more verbose) module map:

```
module MyLib {
  explicit module A {
    header "A.h"
    export *
  }

  explicit module B {
    header "B.h"
    export *
  }
}
```

Export declaration

An *export-declaration* specifies which imported modules will automatically be re-exported as part of a given module’s API.

```
export-declaration:
  export wildcard-module-id
```

```
wildcard-module-id:
  identifier
  '*'
  identifier '.' wildcard-module-id
```

The *export-declaration* names a module or a set of modules that will be re-exported to any translation unit that imports the enclosing module. Each imported module that matches the *wildcard-module-id* up to, but not including, the first `*` will be re-exported.

Example: In the following example, importing `MyLib.Derived` also provides the API for `MyLib.Base`:

```
module MyLib {
  module Base {
    header "Base.h"
  }

  module Derived {
    header "Derived.h"
    export Base
  }
}
```

Note that, if `Derived.h` includes `Base.h`, one can simply use a wildcard export to re-export everything `Derived.h` includes:

```
module MyLib {
  module Base {
    header "Base.h"
  }

  module Derived {
    header "Derived.h"
    export *
  }
}
```

Note: The wildcard export syntax `export *` re-exports all of the modules that were imported in the actual header file. Because `#include` directives are automatically mapped to module imports, `export *` provides the same transitive-inclusion behavior provided by the C preprocessor, e.g., importing a given module implicitly imports all of the modules on which it depends. Therefore, liberal use of `export *` provides excellent backward compatibility for programs that rely on transitive inclusion (i.e., all of them).

Use declaration

A *use-declaration* specifies another module that the current top-level module intends to use. When the option `-fmodules-decluse` is specified, a module can only use other modules that are explicitly specified in this way.

```
use-declaration:
  use module-id
```

Example: In the following example, use of `A` from `C` is not declared, so will trigger a warning.

```
module A {
  header "a.h"
}

module B {
  header "b.h"
}

module C {
  header "c.h"
  use B
}
```

When compiling a source file that implements a module, use the option `-fmodule-name=module-id` to indicate that the source file is logically part of that module.

The compiler at present only applies restrictions to the module directly being built.

Link declaration

A *link-declaration* specifies a library or framework against which a program should be linked if the enclosing module is imported in any translation unit in that program.

```
link-declaration:
```

```
link framework$_opt$ string-literal
```

The *string-literal* specifies the name of the library or framework against which the program should be linked. For example, specifying “clangBasic” would instruct the linker to link with `-lclangBasic` for a Unix-style linker.

A *link-declaration* with the `framework` specifies that the linker should link against the named framework, e.g., with `-framework MyFramework`.

Note: Automatic linking with the `link` directive is not yet widely implemented, because it requires support from both the object file format and the linker. The notion is similar to Microsoft Visual Studio’s `#pragma comment(lib, ..)`.

Configuration macros declaration

The *config-macros-declaration* specifies the set of configuration macros that have an effect on the API of the enclosing module.

```
config-macros-declaration:
  config_macros attributes$_opt$ config-macro-list$_opt$

config-macro-list:
  identifier (',' identifier)*
```

Each *identifier* in the *config-macro-list* specifies the name of a macro. The compiler is required to maintain different variants of the given module for differing definitions of any of the named macros.

A *config-macros-declaration* shall only be present on a top-level module, i.e., a module that is not nested within an enclosing module.

The `exhaustive` attribute specifies that the list of macros in the *config-macros-declaration* is exhaustive, meaning that no other macro definition is intended to have an effect on the API of that module.

Note: The `exhaustive` attribute implies that any macro definitions for macros not listed as configuration macros should be ignored completely when building the module. As an optimization, the compiler could reduce the number of unique module variants by not considering these non-configuration macros. This optimization is not yet implemented in Clang.

A translation unit shall not import the same module under different definitions of the configuration macros.

Note: Clang implements a weak form of this requirement: the definitions used for configuration macros are fixed based on the definitions provided by the command line. If an import occurs and the definition of any configuration macro has changed, the compiler will produce a warning (under the control of `-Wconfig-macros`).

Example: A logging library might provide different API (e.g., in the form of different definitions for a logging macro) based on the `NDEBUG` macro setting:

```
module MyLogger {
  umbrella header "MyLogger.h"
  config_macros [exhaustive] NDEBUG
}
```

Conflict declarations

A *conflict-declaration* describes a case where the presence of two different modules in the same translation unit is likely to cause a problem. For example, two modules may provide similar-but-incompatible functionality.

```
conflict-declaration:
    conflict module-id ',' string-literal
```

The *module-id* of the *conflict-declaration* specifies the module with which the enclosing module conflicts. The specified module shall not have been imported in the translation unit when the enclosing module is imported.

The *string-literal* provides a message to be provided as part of the compiler diagnostic when two modules conflict.

Note: Clang emits a warning (under the control of `-Wmodule-conflict`) when a module conflict is discovered.

Example:

```
module Conflicts {
    explicit module A {
        header "conflict_a.h"
        conflict B, "we just don't like B"
    }

    module B {
        header "conflict_b.h"
    }
}
```

Attributes

Attributes are used in a number of places in the grammar to describe specific behavior of other declarations. The format of attributes is fairly simple.

```
attributes:
    attribute attributes$_opt$_
```

```
attribute:
    '[' identifier '']'
```

Any *identifier* can be used as an attribute, and each declaration specifies what attributes can be applied to it.

Private Module Map Files

Module map files are typically named `module.modulemap` and live either alongside the headers they describe or in a parent directory of the headers they describe. These module maps typically describe all of the API for the library.

However, in some cases, the presence or absence of particular headers is used to distinguish between the “public” and “private” APIs of a particular library. For example, a library may contain the headers `Foo.h` and `Foo_Private.h`, providing public and private APIs, respectively. Additionally, `Foo_Private.h` may only be available on some versions of library, and absent in others. One cannot easily express this with a single module map file in the library:

```
module Foo {
    header "Foo.h"

    explicit module Private {
```

```

    header "Foo_Private.h"
}
}

```

because the header `Foo_Private.h` won't always be available. The module map file could be customized based on whether `Foo_Private.h` is available or not, but doing so requires custom build machinery.

Private module map files, which are named `module.private.modulemap` (or, for backward compatibility, `module_private.map`), allow one to augment the primary module map file with an additional submodule. For example, we would split the module map file above into two module map files:

```

/* module.modulemap */
module Foo {
    header "Foo.h"
}

/* module.private.modulemap */
explicit module Foo.Private {
    header "Foo_Private.h"
}

```

When a `module.private.modulemap` file is found alongside a `module.modulemap` file, it is loaded after the `module.modulemap` file. In our example library, the `module.private.modulemap` file would be available when `Foo_Private.h` is available, making it easier to split a library's public and private APIs along header boundaries.

Modularizing a Platform

To get any benefit out of modules, one needs to introduce module maps for software libraries starting at the bottom of the stack. This typically means introducing a module map covering the operating system's headers and the C standard library headers (in `/usr/include`, for a Unix system).

The module maps will be written using the *module map language*, which provides the tools necessary to describe the mapping between headers and modules. Because the set of headers differs from one system to the next, the module map will likely have to be somewhat customized for, e.g., a particular distribution and version of the operating system. Moreover, the system headers themselves may require some modification, if they exhibit any anti-patterns that break modules. Such common patterns are described below.

Macro-guarded copy-and-pasted definitions System headers vend core types such as `size_t` for users. These types are often needed in a number of system headers, and are almost trivial to write. Hence, it is fairly common to see a definition such as the following copy-and-pasted throughout the headers:

```

#ifndef __SIZE_T
#define __SIZE_T
typedef __SIZE_TYPE__ size_t;
#endif

```

Unfortunately, when modules compiles all of the C library headers together into a single module, only the first actual type definition of `size_t` will be visible, and then only in the submodule corresponding to the lucky first header. Any other headers that have copy-and-pasted versions of this pattern will *not* have a definition of `size_t`. Importing the submodule corresponding to one of those headers will therefore not yield `size_t` as part of the API, because it wasn't there when the header was parsed. The fix for this problem is either to pull the copied declarations into a common header that gets included everywhere `size_t` is part of the API, or to eliminate the `#ifndef` and redefine the `size_t` type. The latter works for C++ headers and C11, but will cause an error for non-modules C90/C99, where redefinition of typedefs is not permitted.

Conflicting definitions Different system headers may provide conflicting definitions for various macros, functions, or types. These conflicting definitions don't tend to cause problems in a pre-modules world unless someone happens to include both headers in one translation unit. Since the fix is often simply “don't do that”, such problems persist. Modules requires that the conflicting definitions be eliminated or that they be placed in separate modules (the former is generally the better answer).

Missing includes Headers are often missing `#include` directives for headers that they actually depend on. As with the problem of conflicting definitions, this only affects unlucky users who don't happen to include headers in the right order. With modules, the headers of a particular module will be parsed in isolation, so the module may fail to build if there are missing includes.

Headers that vend multiple APIs at different times Some systems have headers that contain a number of different kinds of API definitions, only some of which are made available with a given include. For example, the header may vend `size_t` only when the macro `__need_size_t` is defined before that header is included, and also vend `wchar_t` only when the macro `__need_wchar_t` is defined. Such headers are often included many times in a single translation unit, and will have no include guards. There is no sane way to map this header to a submodule. One can either eliminate the header (e.g., by splitting it into separate headers, one per actual API) or simply exclude it in the module map.

To detect and help address some of these problems, the `clang-tools-extra` repository contains a `modularize` tool that parses a set of given headers and attempts to detect these problems and produce a report. See the tool's in-source documentation for information on how to check your system or library headers.

Future Directions

Modules support is under active development, and there are many opportunities remaining to improve it. Here are a few ideas:

Detect unused module imports Unlike with `#include` directives, it should be fairly simple to track whether a directly-imported module has ever been used. By doing so, Clang can emit `unused import` or `unused #include` diagnostics, including Fix-Its to remove the useless imports/includes.

Fix-Its for missing imports It's fairly common for one to make use of some API while writing code, only to get a compiler error about “unknown type” or “no function named” because the corresponding header has not been included. Clang can detect such cases and auto-import the required module, but should provide a Fix-It to add the import.

Improve modularize The `modularize` tool is both extremely important (for deployment) and extremely crude. It needs better UI, better detection of problems (especially for C++), and perhaps an assistant mode to help write module maps for you.

Where To Learn More About Modules

The Clang source code provides additional information about modules:

`clang/lib/Headers/module.modulemap` Module map for Clang's compiler-specific header files.

`clang/test/Modules/` Tests specifically related to modules functionality.

`clang/include/clang/Basic/Module.h` The `Module` class in this header describes a module, and is used throughout the compiler to implement modules.

`clang/include/clang/Lex/ModuleMap.h` The `ModuleMap` class in this header describes the full module map, consisting of all of the module map files that have been parsed, and providing facilities for looking up module maps and mapping between modules and headers (in both directions).

PCHInternals Information about the serialized AST format used for precompiled headers and modules. The actual implementation is in the `clangSerialization` library.

MSVC compatibility

When Clang compiles C++ code for Windows, it attempts to be compatible with MSVC. There are multiple dimensions to compatibility.

First, Clang attempts to be ABI-compatible, meaning that Clang-compiled code should be able to link against MSVC-compiled code successfully. However, C++ ABIs are particularly large and complicated, and Clang's support for MSVC's C++ ABI is a work in progress. If you don't require MSVC ABI compatibility or don't want to use Microsoft's C and C++ runtimes, the mingw32 toolchain might be a better fit for your project.

Second, Clang implements many MSVC language extensions, such as `__declspec(dllexport)` and a handful of pragmas. These are typically controlled by `-fms-extensions`.

Third, MSVC accepts some C++ code that Clang will typically diagnose as invalid. When these constructs are present in widely included system headers, Clang attempts to recover and continue compiling the user's program. Most parsing and semantic compatibility tweaks are controlled by `-fms-compatibility` and `-fdelayed-template-parsing`, and they are a work in progress.

Finally, there is *clang-cl*, a driver program for clang that attempts to be compatible with MSVC's `cl.exe`.

ABI features

The status of major ABI-impacting C++ features:

- Record layout: Complete. We've tested this with a fuzzer and have fixed all known bugs.
- Class inheritance: Mostly complete. This covers all of the standard OO features you would expect: virtual method inheritance, multiple inheritance, and virtual inheritance. Every so often we uncover a bug where our tables are incompatible, but this is pretty well in hand. This feature has also been fuzz tested.
- Name mangling: Ongoing. Every new C++ feature generally needs its own mangling. For example, member pointer template arguments have an interesting and distinct mangling. Fortunately, incorrect manglings usually do not result in runtime errors. Non-inline functions with incorrect manglings usually result in link errors, which are relatively easy to diagnose. Incorrect manglings for inline functions and templates result in multiple copies in the final image. The C++ standard requires that those addresses be equal, but few programs rely on this.
- Member pointers: Mostly complete. Standard C++ member pointers are fully implemented and should be ABI compatible. Both `#pragma pointers_to_members` and the `/vm` flags are supported. However, MSVC supports an extension to allow creating a *pointer to a member of a virtual base class*. Clang does not yet support this.
- Debug info: Minimal. Clang emits both CodeView line tables (similar to what MSVC emits when given the `/Z7` flag) and DWARF debug information into the object file. Microsoft's `link.exe` will transform the CodeView line tables into a PDB, enabling stack traces in all modern Windows debuggers. Clang does not emit any CodeView-compatible type info or description of variable layout. Binaries linked with either `binutils' ld` or LLVM's `lld` should be usable with GDB however sophisticated C++ expressions are likely to fail.
- RTTI: Complete. Generation of RTTI data structures has been finished, along with support for the `/GR` flag.
- Exceptions and SEH: Partial. C++ exceptions (`try / catch / throw`) and structured exceptions (`__try / __except / __finally`) mostly work on x64. 32-bit exception handling support is being worked on. LLVM does not model asynchronous exceptions, so it is currently impossible to catch an asynchronous exception generated in the same frame as the catching `__try`. C++ exception specifications are ignored, but this is *consistent with Visual C++*.
- Thread-safe initialization of local statics: Complete. MSVC 2015 added support for thread-safe initialization of such variables by taking an ABI break. We are ABI compatible with both the MSVC 2013 and 2015 ABI for static local variables.

- Lambdas: Mostly complete. Clang is compatible with Microsoft's implementation of lambdas except for providing overloads for conversion to function pointer for different calling conventions. However, Microsoft's extension is non-conforming.

Template instantiation and name lookup

MSVC allows many invalid constructs in class templates that Clang has historically rejected. In order to parse widely distributed headers for libraries such as the Active Template Library (ATL) and Windows Runtime Library (WRL), some template rules have been relaxed or extended in Clang on Windows.

The first major semantic difference is that MSVC appears to defer all parsing and analysis of inline method bodies in class templates until instantiation time. By default on Windows, Clang attempts to follow suit. This behavior is controlled by the `-fdelayed-template-parsing` flag. While Clang delays parsing of method bodies, it still parses the bodies *before* template argument substitution, which is not what MSVC does. The following compatibility tweaks are necessary to parse the template in those cases.

MSVC allows some name lookup into dependent base classes. Even on other platforms, this has been a [frequently asked question](#) for Clang users. A dependent base class is a base class that depends on the value of a template parameter. Clang cannot see any of the names inside dependent bases while it is parsing your template, so the user is sometimes required to use the `typename` keyword to assist the parser. On Windows, Clang attempts to follow the normal lookup rules, but if lookup fails, it will assume that the user intended to find the name in a dependent base. While parsing the following program, Clang will recover as if the user had written the commented-out code:

```
template <typename T>
struct Foo : T {
    void f() {
        /*typename*/ T::UnknownType x = /*this->*/unknownMember;
    }
};
```

After recovery, Clang warns the user that this code is non-standard and issues a hint suggesting how to fix the problem.

As of this writing, Clang is able to compile a simple ATL hello world application. There are still issues parsing WRL headers for modern Windows 8 apps, but they should be addressed soon.

Clang “man” pages

The following documents are command descriptions for all of the Clang tools. These pages describe how to use the Clang commands and what their options are. Note that these pages do not describe all of the options available for all tools. To get a complete listing, pass the `--help` (general options) or `--help-hidden` (general and debugging options) arguments to the tool you are interested in.

Basic Commands

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

SYNOPSIS

```
clang [options] filename ...
```


DESCRIPTION

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

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

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

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

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

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

Assembler This stage runs the target assembler to translate the output of the compiler into a target object file. The output of this stage is typically called a ".o" file or "object" file.

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

Clang Static Analyzer

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

OPTIONS

Stage Selection Options

-E

Run the preprocessor stage.

-fsyntax-only

Run the preprocessor, parser and type checking stages.

-S

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

-c

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

no stage selection option

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

Language Selection and Mode Options

-x <language>

Treat subsequent input files as having type language.

-std=<language>

Specify the language standard to compile for.

-stdlib=<library>

Specify the C++ standard library to use; supported options are libstdc++ and libc++.

-ansi

Same as -std=c89.

-ObjC, **-ObjC++**

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

-trigraphs

Enable trigraphs.

-ffreestanding

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

-fno-builtin

Disable special handling and optimizations of builtin functions like `strlen()` and `malloc()`.

-fmath-errno

Indicate that math functions should be treated as updating `errno`.

-fpascal-strings

Enable support for Pascal-style strings with “\pfoo”.

-fms-extensions

Enable support for Microsoft extensions.

-fmsc-version=

Set `_MSC_VER`. Defaults to 1300 on Windows. Not set otherwise.

-fborland-extensions

Enable support for Borland extensions.

-fwritable-strings

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

-flax-vector-conversions

Allow loose type checking rules for implicit vector conversions.

-fblocks

Enable the “Blocks” language feature.

-fobjc-gc-only

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

-fobjc-gc

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

-fobjc-abi-version=version

Select the Objective-C ABI version to use. Available versions are 1 (legacy “fragile” ABI), 2 (non-fragile ABI 1), and 3 (non-fragile ABI 2).

-fobjc-nonfragile-abi-version=<version>

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

-fobjc-nonfragile-abi

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

Target Selection Options

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

-arch <architecture>

Specify the architecture to build for.

-mmacosx-version-min=<version>

When building for Mac OS X, specify the minimum version supported by your application.

-miphoneos-version-min

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

-march=<cpu>

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

Code Generation Options

-O0, -O1, -O2, -O3, -Ofast, -Os, -Oz, -O, -O4

Specify which optimization level to use:

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

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

-O2 Moderate level of optimization which enables most optimizations.

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

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

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

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

-O Equivalent to *-O2*.

-O4 and higher

Currently equivalent to *-O3*

-g

Generate debug information. Note that Clang debug information works best at -O0.

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

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

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

-fexceptions

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

-ftrapv

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

-fvisibility

This flag sets the default visibility level.

-fcommon

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

-ftls-model=<model>

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

-flto, -emit-llvm

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

Driver Options

-###

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

--help

Display available options.

-Qunused-arguments

Do not emit any warnings for unused driver arguments.

-Wa, <args>

Pass the comma separated arguments in args to the assembler.

-Wl, <args>

Pass the comma separated arguments in args to the linker.

-Wp, <args>

Pass the comma separated arguments in args to the preprocessor.

- Xanalyzer** <arg>
Pass arg to the static analyzer.
- Xassembler** <arg>
Pass arg to the assembler.
- Xlinker** <arg>
Pass arg to the linker.
- Xpreprocessor** <arg>
Pass arg to the preprocessor.
- o** <file>
Write output to file.
- print-file-name**=<file>
Print the full library path of file.
- print-libgcc-file-name**
Print the library path for “libgcc.a”.
- print-prog-name**=<name>
Print the full program path of name.
- print-search-dirs**
Print the paths used for finding libraries and programs.
- save-temps**
Save intermediate compilation results.
- integrated-as, -no-integrated-as**
Used to enable and disable, respectively, the use of the integrated assembler. Whether the integrated assembler is on by default is target dependent.
- time**
Time individual commands.
- ftime-report**
Print timing summary of each stage of compilation.
- v**
Show commands to run and use verbose output.

Diagnostics Options

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

Preprocessor Options

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

-I<directory>

Add the specified directory to the search path for include files.

-F<directory>

Add the specified directory to the search path for framework include files.

-nostdinc

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

-nostdlibinc

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

-nobuiltininc

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

ENVIRONMENT

TMPDIR, TEMP, TMP

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

CPATH

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

Empty components in the environment variable are ignored.

C_INCLUDE_PATH, OBJC_INCLUDE_PATH, CPLUS_INCLUDE_PATH, OBJCPLUS_INCLUDE_PATH

These environment variables specify additional paths, as for `CPATH`, which are only used when processing the appropriate language.

MACOSX_DEPLOYMENT_TARGET

If `-mmacosx-version-min` is unspecified, the default deployment target is read from this environment variable. This option only affects Darwin targets.

BUGS

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

SEE ALSO

as(1), ld(1)

Frequently Asked Questions (FAQ)

- *Driver*
 - *I run `clang -ccl ...` and get weird errors about missing headers*
 - *I get errors about some headers being missing (`stddef.h`, `stdarg.h`)*

Driver

I run `clang -cc1 ...` and get weird errors about missing headers

Given this source file:

```
#include <stdio.h>

int main() {
    printf("Hello world\n");
}
```

If you run:

```
$ clang -cc1 hello.c
hello.c:1:10: fatal error: 'stdio.h' file not found
#include <stdio.h>
      ^
1 error generated.
```

`clang -cc1` is the frontend, `clang` is the *driver*. The driver invokes the frontend with options appropriate for your system. To see these options, run:

```
$ clang -### -c hello.c
```

Some clang command line options are driver-only options, some are frontend-only options. Frontend-only options are intended to be used only by clang developers. Users should not run `clang -cc1` directly, because `-cc1` options are not guaranteed to be stable.

If you want to use a frontend-only option (“a `-cc1` option”), for example `-ast-dump`, then you need to take the `clang -cc1` line generated by the driver and add the option you need. Alternatively, you can run `clang -Xclang <option> ...` to force the driver pass `<option>` to `clang -cc1`.

I get errors about some headers being missing (`stddef.h`, `stdarg.h`)

Some header files (`stddef.h`, `stdarg.h`, and others) are shipped with Clang — these are called builtin includes. Clang searches for them in a directory relative to the location of the `clang` binary. If you moved the `clang` binary, you need to move the builtin headers, too.

More information can be found in the *Builtin includes* section.

Using Clang as a Library

Choosing the Right Interface for Your Application

Clang provides infrastructure to write tools that need syntactic and semantic information about a program. This document will give a short introduction of the different ways to write clang tools, and their pros and cons.

LibClang

LibClang is a stable high level C interface to clang. When in doubt LibClang is probably the interface you want to use. Consider the other interfaces only when you have a good reason not to use LibClang.

Canonical examples of when to use LibClang:

- Xcode
- Clang Python Bindings

Use LibClang when you...:

- want to interface with clang from other languages than C++
- need a stable interface that takes care to be backwards compatible
- want powerful high-level abstractions, like iterating through an AST with a cursor, and don't want to learn all the nitty gritty details of Clang's AST.

Do not use LibClang when you...:

- want full control over the Clang AST

Clang Plugins

Clang Plugins allow you to run additional actions on the AST as part of a compilation. Plugins are dynamic libraries that are loaded at runtime by the compiler, and they're easy to integrate into your build environment.

Canonical examples of when to use Clang Plugins:

- special lint-style warnings or errors for your project
- creating additional build artifacts from a single compile step

Use Clang Plugins when you...:

- need your tool to rerun if any of the dependencies change
- want your tool to make or break a build
- need full control over the Clang AST

Do not use Clang Plugins when you...:

- want to run tools outside of your build environment
- want full control on how Clang is set up, including mapping of in-memory virtual files
- need to run over a specific subset of files in your project which is not necessarily related to any changes which would trigger rebuilds

LibTooling

LibTooling is a C++ interface aimed at writing standalone tools, as well as integrating into services that run clang tools. Canonical examples of when to use LibTooling:

- a simple syntax checker
- refactoring tools

Use LibTooling when you...:

- want to run tools over a single file, or a specific subset of files, independently of the build system
- want full control over the Clang AST
- want to share code with Clang Plugins

Do not use LibTooling when you...:

- want to run as part of the build triggered by dependency changes
- want a stable interface so you don't need to change your code when the AST API changes
- want high level abstractions like cursors and code completion out of the box
- do not want to write your tools in C++

Clang tools are a collection of specific developer tools built on top of the LibTooling infrastructure as part of the Clang project. They are targeted at automating and improving core development activities of C/C++ developers.

Examples of tools we are building or planning as part of the Clang project:

- Syntax checking (**clang-check**)
- Automatic fixing of compile errors (**clang-fixit**)
- Automatic code formatting (**clang-format**)
- Migration tools for new features in new language standards
- Core refactoring tools

External Clang Examples

Introduction

This page provides some examples of the kinds of things that people have done with Clang that might serve as useful guides (or starting points) from which to develop your own tools. They may be helpful even for something as banal (but necessary) as how to set up your build to integrate Clang.

Clang’s library-based design is deliberately aimed at facilitating use by external projects, and we are always interested in improving Clang to better serve our external users. Some typical categories of applications where Clang is used are:

- Static analysis.
- Documentation/cross-reference generation.

If you know of (or wrote!) a tool or project using Clang, please send an email to Clang’s [development discussion mailing list](#) to have it added. (or if you are already a Clang contributor, feel free to directly commit additions). Since the primary purpose of this page is to provide examples that can help developers, generally they must have code available.

List of projects and tools

<https://github.com/Andersbakken/rtags/> “RTags is a client/server application that indexes c/c++ code and keeps a persistent in-memory database of references, symbolnames, completions etc.”

<http://rprichard.github.com/sourceweb/> “A C/C++ source code indexer and navigator”

<https://github.com/etaoins/qconnectlint> “qconnectlint is a Clang tool for statically verifying the consistency of signal and slot connections made with Qt’s `QObject::connect`.”

https://github.com/woboq/woboq_codebrowser “The Woboq Code Browser is a web-based code browser for C/C++ projects. Check out <http://code.woboq.org/> for an example!”

<https://github.com/mozilla/dxr> “DXR is a source code cross-reference tool that uses static analysis data collected by instrumented compilers.”

<https://github.com/eschulte/clang-mutate> “This tool performs a number of operations on C-language source files.”

<https://github.com/gmarpons/Crisp> “A coding rule validation add-on for LLVM/clang. Crisp rules are written in Prolog. A high-level declarative DSL to easily write new rules is under development. It will be called CRISP, an acronym for *Coding Rules in Sugared Prolog*.”

<https://github.com/drothlis/clang-ctags> “Generate tag file for C++ source code.”

https://github.com/exclipy/clang_indexer “This is an indexer for C and C++ based on the libclang library.”

<https://github.com/holtgrewe/linty> “Linty - C/C++ Style Checking with Python & libclang.”

<https://github.com/axw/cmonster> “cmonster is a Python wrapper for the Clang C++ parser.”

<https://github.com/rizotto/Constantine> “Constantine is a toy project to learn how to write clang plugin. Implements pseudo const analysis. Generates warnings about variables, which were declared without const qualifier.”

<https://github.com/jessevd/cldoc> “cldoc is a Clang based documentation generator for C and C++. cldoc tries to solve the issue of writing C/C++ software documentation with a modern, non-intrusive and robust approach.”

<https://github.com/AlexDenisov/ToyClangPlugin> “The simplest Clang plugin implementing a semantic check for Objective-C. This example shows how to use the `DiagnosticsEngine` (emit warnings, errors, fixit hints). See also http://l.rw.rw/clang_plugin for step-by-step instructions.”

Introduction to the Clang AST

This document gives a gentle introduction to the mysteries of the Clang AST. It is targeted at developers who either want to contribute to Clang, or use tools that work based on Clang’s AST, like the AST matchers.

[Slides](#)

Introduction

Clang’s AST is different from ASTs produced by some other compilers in that it closely resembles both the written C++ code and the C++ standard. For example, parenthesis expressions and compile time constants are available in an unreduced form in the AST. This makes Clang’s AST a good fit for refactoring tools.

Documentation for all Clang AST nodes is available via the generated [Doxygen](#). The doxygen online documentation is also indexed by your favorite search engine, which will make a search for clang and the AST node’s class name usually turn up the doxygen of the class you’re looking for (for example, search for: clang ParenExpr).

Examining the AST

A good way to familiarize yourself with the Clang AST is to actually look at it on some simple example code. Clang has a builtin AST-dump mode, which can be enabled with the flag `-ast-dump`.

Let’s look at a simple example AST:

```
$ cat test.cc
int f(int x) {
    int result = (x / 42);
    return result;
}

# Clang by default is a frontend for many tools; -Xclang is used to pass
# options directly to the C++ frontend.
$ clang -Xclang -ast-dump -fsyntax-only test.cc
TranslationUnitDecl 0x5aea0d0 <<invalid sloc>>
... cutting out internal declarations of clang ...
^-FunctionDecl 0x5aeab50 <test.cc:1:1, line:4:1> f 'int (int)'
  |-ParmVarDecl 0x5aeaa90 <line:1:7, col:11> x 'int'
  ^-CompoundStmt 0x5aead88 <col:14, line:4:1>
    |-DeclStmt 0x5aead10 <line:2:3, col:24>
      | ^-VarDecl 0x5aeac10 <col:3, col:23> result 'int'
      |   ^-ParenExpr 0x5aeacf0 <col:16, col:23> 'int'
      |     ^-BinaryOperator 0x5aeacc8 <col:17, col:21> 'int' '/'
      |       |-ImplicitCastExpr 0x5aeacb0 <col:17> 'int' <LValueToRValue>
      |         | ^-DeclRefExpr 0x5aeac68 <col:17> 'int' lvalue ParmVar 0x5aeaa90 'x' 'int'
      |         ↪
      |           ^-IntegerLiteral 0x5aeac90 <col:21> 'int' 42
    ^-ReturnStmt 0x5aead68 <line:3:3, col:10>
      ^-ImplicitCastExpr 0x5aead50 <col:10> 'int' <LValueToRValue>
        ^-DeclRefExpr 0x5aead28 <col:10> 'int' lvalue Var 0x5aeac10 'result' 'int'
```

The toplevel declaration in a translation unit is always the [translation unit declaration](#). In this example, our first user written declaration is the [function declaration](#) of “f”. The body of “f” is a [compound statement](#), whose child nodes are a [declaration statement](#) that declares our result variable, and the [return statement](#).

AST Context

All information about the AST for a translation unit is bundled up in the class `ASTContext`. It allows traversal of the whole translation unit starting from `getTranslationUnitDecl`, or to access Clang's [table of identifiers](#) for the parsed translation unit.

AST Nodes

Clang's AST nodes are modeled on a class hierarchy that does not have a common ancestor. Instead, there are multiple larger hierarchies for basic node types like `Decl` and `Stmt`. Many important AST nodes derive from `Type`, `Decl`, `DeclContext` or `Stmt`, with some classes deriving from both `Decl` and `DeclContext`.

There are also a multitude of nodes in the AST that are not part of a larger hierarchy, and are only reachable from specific other nodes, like `CXXBaseSpecifier`.

Thus, to traverse the full AST, one starts from the `TranslationUnitDecl` and then recursively traverses everything that can be reached from that node - this information has to be encoded for each specific node type. This algorithm is encoded in the `RecursiveASTVisitor`. See the [RecursiveASTVisitor tutorial](#).

The two most basic nodes in the Clang AST are statements (`Stmt`) and declarations (`Decl`). Note that expressions (`Expr`) are also statements in Clang's AST.

LibTooling

LibTooling is a library to support writing standalone tools based on Clang. This document will provide a basic walkthrough of how to write a tool using LibTooling.

For the information on how to setup Clang Tooling for LLVM see [How To Setup Clang Tooling For LLVM](#)

Introduction

Tools built with LibTooling, like Clang Plugins, run `FrontendActions` over code.

In this tutorial, we'll demonstrate the different ways of running Clang's `SyntaxOnlyAction`, which runs a quick syntax check, over a bunch of code.

Parsing a code snippet in memory

If you ever wanted to run a `FrontendAction` over some sample code, for example to unit test parts of the Clang AST, `runToolOnCode` is what you looked for. Let me give you an example:

```
#include "clang/Tooling/Tooling.h"

TEST(runToolOnCode, CanSyntaxCheckCode) {
    // runToolOnCode returns whether the action was correctly run over the
    // given code.
    EXPECT_TRUE(runToolOnCode(new clang::SyntaxOnlyAction, "class X {};"));
}
```

Writing a standalone tool

Once you unit tested your `FrontendAction` to the point where it cannot possibly break, it's time to create a standalone tool. For a standalone tool to run clang, it first needs to figure out what command line arguments to use for a specified file. To that end we create a `CompilationDatabase`. There are different ways to create a compilation database, and we need to support all of them depending on command-line options. There's the `CommonOptionsParser` class that takes the responsibility to parse command-line parameters related to compilation databases and inputs, so that all tools share the implementation.

Parsing common tools options

`CompilationDatabase` can be read from a build directory or the command line. Using `CommonOptionsParser` allows for explicit specification of a compile command line, specification of build path using the `-p` command-line option, and automatic location of the compilation database using source files paths.

```
#include "clang/Tooling/CommonOptionsParser.h"
#include "llvm/Support/CommandLine.h"

using namespace clang::tooling;

// Apply a custom category to all command-line options so that they are the
// only ones displayed.
static llvm::cl::OptionCategory MyToolCategory("my-tool options");

int main(int argc, const char **argv) {
    // CommonOptionsParser constructor will parse arguments and create a
    // CompilationDatabase. In case of error it will terminate the program.
    CommonOptionsParser OptionsParser(argc, argv, MyToolCategory);

    // Use OptionsParser.getCompilations() and OptionsParser.getSourcePathList()
    // to retrieve CompilationDatabase and the list of input file paths.
}
```

Creating and running a ClangTool

Once we have a `CompilationDatabase`, we can create a `ClangTool` and run our `FrontendAction` over some code. For example, to run the `SyntaxOnlyAction` over the files “a.cc” and “b.cc” one would write:

```
// A clang tool can run over a number of sources in the same process...
std::vector<std::string> Sources;
Sources.push_back("a.cc");
Sources.push_back("b.cc");

// We hand the CompilationDatabase we created and the sources to run over into
// the tool constructor.
ClangTool Tool(OptionsParser.getCompilations(), Sources);

// The ClangTool needs a new FrontendAction for each translation unit we run
// on. Thus, it takes a FrontendActionFactory as parameter. To create a
// FrontendActionFactory from a given FrontendAction type, we call
// newFrontendActionFactory<clang::SyntaxOnlyAction>().
int result = Tool.run(newFrontendActionFactory<clang::SyntaxOnlyAction>().get());
```

Putting it together — the first tool

Now we combine the two previous steps into our first real tool. A more advanced version of this example tool is also checked into the clang tree at `tools/clang-check/ClangCheck.cpp`.

```
// Declares clang::SyntaxOnlyAction.
#include "clang/Frontend/FrontendActions.h"
#include "clang/Tooling/CommonOptionsParser.h"
#include "clang/Tooling/Tooling.h"
// Declares llvm::cl::extrahelp.
#include "llvm/Support/CommandLine.h"

using namespace clang::tooling;
using namespace llvm;

// Apply a custom category to all command-line options so that they are the
// only ones displayed.
static cl::OptionCategory MyToolCategory("my-tool options");

// CommonOptionsParser declares HelpMessage with a description of the common
// command-line options related to the compilation database and input files.
// It's nice to have this help message in all tools.
static cl::extrahelp CommonHelp(CommonOptionsParser::HelpMessage);

// A help message for this specific tool can be added afterwards.
static cl::extrahelp MoreHelp("\nMore help text...");

int main(int argc, const char **argv) {
    CommonOptionsParser OptionsParser(argc, argv, MyToolCategory);
    ClangTool Tool(OptionsParser.getCompilations(),
                  OptionsParser.getSourcePathList());
    return Tool.run(newFrontendActionFactory<clang::SyntaxOnlyAction>().get());
}
```

Running the tool on some code

When you check out and build clang, clang-check is already built and available to you in `bin/clang-check` inside your build directory.

You can run clang-check on a file in the llvm repository by specifying all the needed parameters after a “--” separator:

```
$ cd /path/to/source/llvm
$ export BD=/path/to/build/llvm
$ $BD/bin/clang-check tools/clang/tools/clang-check/ClangCheck.cpp -- \
    clang++ -D__STDC_CONSTANT_MACROS -D__STDC_LIMIT_MACROS \
    -Itools/clang/include -I$BD/include -Iinclude \
    -Itools/clang/lib/Headers -c
```

As an alternative, you can also configure cmake to output a compile command database into its build directory:

```
# Alternatively to calling cmake, use ccmake, toggle to advanced mode and
# set the parameter CMAKE_EXPORT_COMPILE_COMMANDS from the UI.
$ cmake -DCMAKE_EXPORT_COMPILE_COMMANDS=ON .
```

This creates a file called `compile_commands.json` in the build directory. Now you can run **clang-check** over files in the project by specifying the build path as first argument and some source files as further positional arguments:

```
$ cd /path/to/source/llvm
$ export BD=/path/to/build/llvm
$ $BD/bin/clang-check -p $BD tools/clang/tools/clang-check/ClangCheck.cpp
```

Builtin includes

Clang tools need their builtin headers and search for them the same way Clang does. Thus, the default location to look for builtin headers is in a path `$(dirname /path/to/tool)/../lib/clang/3.3/include` relative to the tool binary. This works out-of-the-box for tools running from LLVM's toplevel binary directory after building clang-headers, or if the tool is running from the binary directory of a clang install next to the clang binary.

Tips: if your tool fails to find `stddef.h` or similar headers, call the tool with `-v` and look at the search paths it looks through.

Linking

For a list of libraries to link, look at one of the tools' Makefiles (for example `clang-check/Makefile`).

LibFormat

LibFormat is a library that implements automatic source code formatting based on Clang. This document describes the LibFormat interface and design as well as some basic style discussions.

If you just want to use *clang-format* as a tool or integrated into an editor, checkout *ClangFormat*.

Design

FIXME: Write up design.

Interface

The core routine of LibFormat is `reformat()`:

```
tooling::Replacements reformat(const FormatStyle &Style, Lexer &Lex,
                               SourceManager &SourceMgr,
                               std::vector<CharSourceRange> Ranges);
```

This reads a token stream out of the lexer `Lex` and reformats all the code ranges in `Ranges`. The `FormatStyle` controls basic decisions made during formatting. A list of options can be found under *Style Options*.

Style Options

The style options describe specific formatting options that can be used in order to make *ClangFormat* comply with different style guides. Currently, two style guides are hard-coded:

```
/// \brief Returns a format style complying with the LLVM coding standards:
/// http://llvm.org/docs/CodingStandards.html.
FormatStyle getLLVMStyle();
```



```
/// \brief Returns a format style complying with Google's C++ style guide:
/// http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml.
FormatStyle getGoogleStyle();
```

These options are also exposed in the *standalone tools* through the `-style` option.

In the future, we plan on making this configurable.

Clang Plugins

Clang Plugins make it possible to run extra user defined actions during a compilation. This document will provide a basic walkthrough of how to write and run a Clang Plugin.

Introduction

Clang Plugins run `FrontendActions` over code. See the *FrontendAction tutorial* on how to write a `FrontendAction` using the `RecursiveASTVisitor`. In this tutorial, we'll demonstrate how to write a simple clang plugin.

Writing a `PluginASTAction`

The main difference from writing normal `FrontendActions` is that you can handle plugin command line options. The `PluginASTAction` base class declares a `ParseArgs` method which you have to implement in your plugin.

```
bool ParseArgs(const CompilerInstance &CI,
               const std::vector<std::string>& args) {
    for (unsigned i = 0, e = args.size(); i != e; ++i) {
        if (args[i] == "-some-arg") {
            // Handle the command line argument.
        }
    }
    return true;
}
```

Registering a plugin

A plugin is loaded from a dynamic library at runtime by the compiler. To register a plugin in a library, use `FrontendPluginRegistry::Add<>`:

```
static FrontendPluginRegistry::Add<MyPlugin> X("my-plugin-name", "my plugin_
↳description");
```

Putting it all together

Let's look at an example plugin that prints top-level function names. This example is checked into the clang repository; please take a look at the [latest version of PrintFunctionNames.cpp](#).

Running the plugin

To run a plugin, the dynamic library containing the plugin registry must be loaded via the `-load` command line option. This will load all plugins that are registered, and you can select the plugins to run by specifying the `-plugin` option. Additional parameters for the plugins can be passed with `-plugin-arg-`.

Note that those options must reach clang's `cc1` process. There are two ways to do so:

- Directly call the parsing process by using the `-cc1` option; this has the downside of not configuring the default header search paths, so you'll need to specify the full system path configuration on the command line.
- Use clang as usual, but prefix all arguments to the `cc1` process with `-Xclang`.

For example, to run the `print-function-names` plugin over a source file in clang, first build the plugin, and then call clang with the plugin from the source tree:

```
$ export BD=/path/to/build/directory
$ (cd $BD && make PrintFunctionNames )
$ clang++ -D_GNU_SOURCE -D_DEBUG -D__STDC_CONSTANT_MACROS \
          -D__STDC_FORMAT_MACROS -D__STDC_LIMIT_MACROS -D_GNU_SOURCE \
          -I$BD/tools/clang/include -Itools/clang/include -I$BD/include -Iinclude \
          tools/clang/tools/clang-check/ClangCheck.cpp -fsyntax-only \
          -Xclang -load -Xclang $BD/lib/PrintFunctionNames.so -Xclang \
          -plugin -Xclang print-fns
```

Also see the `print-function-name` plugin example's [README](#)

How to write RecursiveASTVisitor based ASTFrontendActions.

Introduction

In this tutorial you will learn how to create a `FrontendAction` that uses a `RecursiveASTVisitor` to find `CXXRecordDecl` AST nodes with a specified name.

Creating a FrontendAction

When writing a clang based tool like a Clang Plugin or a standalone tool based on LibTooling, the common entry point is the `FrontendAction`. `FrontendAction` is an interface that allows execution of user specific actions as part of the compilation. To run tools over the AST clang provides the convenience interface `ASTFrontendAction`, which takes care of executing the action. The only part left is to implement the `CreateASTConsumer` method that returns an `ASTConsumer` per translation unit.

```
class FindNamedClassAction : public clang::ASTFrontendAction {
public:
    virtual std::unique_ptr<clang::ASTConsumer> CreateASTConsumer(
        clang::CompilerInstance &Compiler, llvm::StringRef InFile) {
        return std::unique_ptr<clang::ASTConsumer>(
            new FindNamedClassConsumer);
    }
};
```

Creating an ASTConsumer

ASTConsumer is an interface used to write generic actions on an AST, regardless of how the AST was produced. ASTConsumer provides many different entry points, but for our use case the only one needed is `HandleTranslationUnit`, which is called with the `ASTContext` for the translation unit.

```
class FindNamedClassConsumer : public clang::ASTConsumer {
public:
    virtual void HandleTranslationUnit(clang::ASTContext &Context) {
        // Traversing the translation unit decl via a RecursiveASTVisitor
        // will visit all nodes in the AST.
        Visitor.TraverseDecl(Context.getTranslationUnitDecl());
    }
private:
    // A RecursiveASTVisitor implementation.
    FindNamedClassVisitor Visitor;
};
```

Using the RecursiveASTVisitor

Now that everything is hooked up, the next step is to implement a `RecursiveASTVisitor` to extract the relevant information from the AST.

The `RecursiveASTVisitor` provides hooks of the form `bool VisitNodeType(NodeType *)` for most AST nodes; the exception are `TypeLoc` nodes, which are passed by-value. We only need to implement the methods for the relevant node types.

Let's start by writing a `RecursiveASTVisitor` that visits all `CXXRecordDecl`'s.

```
class FindNamedClassVisitor
: public RecursiveASTVisitor<FindNamedClassVisitor> {
public:
    bool VisitCXXRecordDecl(CXXRecordDecl *Declaration) {
        // For debugging, dumping the AST nodes will show which nodes are already
        // being visited.
        Declaration->dump();

        // The return value indicates whether we want the visitation to proceed.
        // Return false to stop the traversal of the AST.
        return true;
    }
};
```

In the methods of our `RecursiveASTVisitor` we can now use the full power of the Clang AST to drill through to the parts that are interesting for us. For example, to find all class declaration with a certain name, we can check for a specific qualified name:

```
bool VisitCXXRecordDecl(CXXRecordDecl *Declaration) {
    if (Declaration->getQualifiedNameAsString() == "n::m::C")
        Declaration->dump();
    return true;
}
```

Accessing the SourceManager and ASTContext

Some of the information about the AST, like source locations and global identifier information, are not stored in the AST nodes themselves, but in the `ASTContext` and its associated source manager. To retrieve them we need to hand the `ASTContext` into our `RecursiveASTVisitor` implementation.

The `ASTContext` is available from the `CompilerInstance` during the call to `CreateASTConsumer`. We can thus extract it there and hand it into our freshly created `FindNamedClassConsumer`:

```
virtual std::unique_ptr<clang::ASTConsumer> CreateASTConsumer(
    clang::CompilerInstance &Compiler, llvm::StringRef InFile) {
    return std::unique_ptr<clang::ASTConsumer>(
        new FindNamedClassConsumer(&Compiler.getASTContext()));
}
```

Now that the `ASTContext` is available in the `RecursiveASTVisitor`, we can do more interesting things with AST nodes, like looking up their source locations:

```
bool VisitCXXRecordDecl(CXXRecordDecl *Declaration) {
    if (Declaration->getQualifiedNameAsString() == "n::m::C") {
        // getFullLoc uses the ASTContext's SourceManager to resolve the source
        // location and break it up into its line and column parts.
        FullSourceLoc FullLocation = Context->getFullLoc(Declaration->getLocStart());
        if (FullLocation.isValid())
            llvm::outs() << "Found declaration at "
                        << FullLocation.getSpellingLineNumber() << ":"
                        << FullLocation.getSpellingColumnNumber() << "\n";
    }
    return true;
}
```

Putting it all together

Now we can combine all of the above into a small example program:

```
#include "clang/AST/ASTConsumer.h"
#include "clang/AST/RecursiveASTVisitor.h"
#include "clang/Frontend/CompilerInstance.h"
#include "clang/Frontend/FrontendAction.h"
#include "clang/Tooling/Tooling.h"

using namespace clang;

class FindNamedClassVisitor
    : public RecursiveASTVisitor<FindNamedClassVisitor> {
public:
    explicit FindNamedClassVisitor(ASTContext *Context)
        : Context(Context) {}

    bool VisitCXXRecordDecl(CXXRecordDecl *Declaration) {
        if (Declaration->getQualifiedNameAsString() == "n::m::C") {
            FullSourceLoc FullLocation = Context->getFullLoc(Declaration->getLocStart());
            if (FullLocation.isValid())
                llvm::outs() << "Found declaration at "
                            << FullLocation.getSpellingLineNumber() << ":"
                            << FullLocation.getSpellingColumnNumber() << "\n";
        }
    }
}
```

```

    return true;
}

private:
    ASTContext *Context;
};

class FindNamedClassConsumer : public clang::ASTConsumer {
public:
    explicit FindNamedClassConsumer(ASTContext *Context)
        : Visitor(Context) {}

    virtual void HandleTranslationUnit(clang::ASTContext &Context) {
        Visitor.TraverseDecl(Context.getTranslationUnitDecl());
    }
private:
    FindNamedClassVisitor Visitor;
};

class FindNamedClassAction : public clang::ASTFrontendAction {
public:
    virtual std::unique_ptr<clang::ASTConsumer> CreateASTConsumer(
        clang::CompilerInstance &Compiler, llvm::StringRef InFile) {
        return std::unique_ptr<clang::ASTConsumer>(
            new FindNamedClassConsumer(&Compiler.getASTContext()));
    }
};

int main(int argc, char **argv) {
    if (argc > 1) {
        clang::tooling::runToolOnCode(new FindNamedClassAction, argv[1]);
    }
}

```

We store this into a file called `FindClassDecls.cpp` and create the following `CMakeLists.txt` to link it:

```

set(LLVM_USED_LIBS clangTooling)

add_clang_executable(find-class-decls FindClassDecls.cpp)

```

When running this tool over a small code snippet it will output all declarations of a class `n::m::C` it found:

```

$ ./bin/find-class-decls "namespace n { namespace m { class C {}; } }"
Found declaration at 1:29

```

Tutorial for building tools using LibTooling and LibASTMatchers

This document is intended to show how to build a useful source-to-source translation tool based on Clang's LibTooling. It is explicitly aimed at people who are new to Clang, so all you should need is a working knowledge of C++ and the command line.

In order to work on the compiler, you need some basic knowledge of the abstract syntax tree (AST). To this end, the reader is encouraged to skim the *[Introduction to the Clang AST](#)*

Step 0: Obtaining Clang

As Clang is part of the LLVM project, you'll need to download LLVM's source code first. Both Clang and LLVM are maintained as Subversion repositories, but we'll be accessing them through the git mirror. For further information, see the [getting started guide](#).

```
mkdir ~/clang-llvm && cd ~/clang-llvm
git clone http://llvm.org/git/llvm.git
cd llvm/tools
git clone http://llvm.org/git/clang.git
cd clang/tools
git clone http://llvm.org/git/clang-tools-extra.git extra
```

Next you need to obtain the CMake build system and Ninja build tool. You may already have CMake installed, but current binary versions of CMake aren't built with Ninja support.

```
cd ~/clang-llvm
git clone https://github.com/martine/ninja.git
cd ninja
git checkout release
./bootstrap.py
sudo cp ninja /usr/bin/

cd ~/clang-llvm
git clone git://cmake.org/stage/cmake.git
cd cmake
git checkout next
./bootstrap
make
sudo make install
```

Okay. Now we'll build Clang!

```
cd ~/clang-llvm
mkdir build && cd build
cmake -G Ninja ../llvm -DLLVM_BUILD_TESTS=ON # Enable tests; default is off.
ninja
ninja check # Test LLVM only.
ninja clang-test # Test Clang only.
ninja install
```

And we're live.

All of the tests should pass, though there is a (very) small chance that you can catch LLVM and Clang out of sync. Running 'git svn rebase' in both the llvm and clang directories should fix any problems.

Finally, we want to set Clang as its own compiler.

```
cd ~/clang-llvm/build
ccmake ../llvm
```

The second command will bring up a GUI for configuring Clang. You need to set the entry for CMAKE_CXX_COMPILER. Press 't' to turn on advanced mode. Scroll down to CMAKE_CXX_COMPILER, and set it to /usr/bin/clang++, or wherever you installed it. Press 'c' to configure, then 'g' to generate CMake's files.

Finally, run ninja one last time, and you're done.

Step 1: Create a ClangTool

Now that we have enough background knowledge, it's time to create the simplest productive ClangTool in existence: a syntax checker. While this already exists as `clang-check`, it's important to understand what's going on.

First, we'll need to create a new directory for our tool and tell CMake that it exists. As this is not going to be a core clang tool, it will live in the `tools/extra` repository.

```
cd ~/clang-llvm/llvm/tools/clang
mkdir tools/extra/loop-convert
echo 'add_subdirectory(loop-convert)' >> tools/extra/CMakeLists.txt
vim tools/extra/loop-convert/CMakeLists.txt
```

CMakeLists.txt should have the following contents:

```
set(LLVM_LINK_COMPONENTS support)
set(LLVM_USED_LIBS clangTooling clangBasic clangAST)

add_clang_executable(loop-convert
  LoopConvert.cpp
)
target_link_libraries(loop-convert
  clangTooling
  clangBasic
  clangASTMatchers
)
```

With that done, Ninja will be able to compile our tool. Let's give it something to compile! Put the following into `tools/extra/loop-convert/LoopConvert.cpp`. A detailed explanation of why the different parts are needed can be found in the LibTooling documentation.

```
// Declares clang::SyntaxOnlyAction.
#include "clang/Frontend/FrontendActions.h"
#include "clang/Tooling/CommonOptionsParser.h"
#include "clang/Tooling/Tooling.h"
// Declares llvm::cl::extrahelp.
#include "llvm/Support/CommandLine.h"

using namespace clang::tooling;
using namespace llvm;

// Apply a custom category to all command-line options so that they are the
// only ones displayed.
static llvm::cl::OptionCategory MyToolCategory("my-tool options");

// CommonOptionsParser declares HelpMessage with a description of the common
// command-line options related to the compilation database and input files.
// It's nice to have this help message in all tools.
static cl::extrahelp CommonHelp(CommonOptionsParser::HelpMessage);

// A help message for this specific tool can be added afterwards.
static cl::extrahelp MoreHelp("\nMore help text...");

int main(int argc, const char **argv) {
  CommonOptionsParser OptionsParser(argc, argv, MyToolCategory);
  ClangTool Tool(OptionsParser.getCompilations(),
    OptionsParser.getSourcePathList());
  return Tool.run(newFrontendActionFactory<clang::SyntaxOnlyAction>().get());
}
```

```
}
```

And that's it! You can compile our new tool by running `ninja` from the `build` directory.

```
cd ~/clang-llvm/build
ninja
```

You should now be able to run the syntax checker, which is located in `~/clang-llvm/build/bin`, on any source file. Try it!

```
echo "int main() { return 0; }" > test.cpp
bin/loop-convert test.cpp --
```

Note the two dashes after we specify the source file. The additional options for the compiler are passed after the dashes rather than loading them from a compilation database - there just aren't any options needed right now.

Intermezzo: Learn AST matcher basics

Clang recently introduced the *ASTMatcher library* to provide a simple, powerful, and concise way to describe specific patterns in the AST. Implemented as a DSL powered by macros and templates (see `ASTMatchers.h` if you're curious), matchers offer the feel of algebraic data types common to functional programming languages.

For example, suppose you wanted to examine only binary operators. There is a matcher to do exactly that, conveniently named `binaryOperator`. I'll give you one guess what this matcher does:

```
binaryOperator(hasOperatorName("+"), hasLHS(integerLiteral(equals(0))))
```

Shockingly, it will match against addition expressions whose left hand side is exactly the literal 0. It will not match against other forms of 0, such as `'\0'` or `NULL`, but it will match against macros that expand to 0. The matcher will also not match against calls to the overloaded operator `'+'`, as there is a separate `operatorCallExpr` matcher to handle overloaded operators.

There are AST matchers to match all the different nodes of the AST, narrowing matchers to only match AST nodes fulfilling specific criteria, and traversal matchers to get from one kind of AST node to another. For a complete list of AST matchers, take a look at the [AST Matcher References](#)

All matcher that are nouns describe entities in the AST and can be bound, so that they can be referred to whenever a match is found. To do so, simply call the method `bind` on these matchers, e.g.:

```
variable(hasType(isInteger())) .bind("intvar")
```

Step 2: Using AST matchers

Okay, on to using matchers for real. Let's start by defining a matcher which will capture all `for` statements that define a new variable initialized to zero. Let's start with matching all `for` loops:

```
forStmt()
```

Next, we want to specify that a single variable is declared in the first portion of the loop, so we can extend the matcher to

```
forStmt(hasLoopInit(declStmt(hasSingleDecl(varDecl()))))
```

Finally, we can add the condition that the variable is initialized to zero.


```
forStmt (hasLoopInit (declStmt (hasSingleDecl (varDecl (
  hasInitializer (integerLiteral (equals(0))))))))))
```

It is fairly easy to read and understand the matcher definition (“match loops whose init portion declares a single variable which is initialized to the integer literal 0”), but deciding that every piece is necessary is more difficult. Note that this matcher will not match loops whose variables are initialized to ‘\0’, 0.0, NULL, or any form of zero besides the integer 0.

The last step is giving the matcher a name and binding the `ForStmt` as we will want to do something with it:

```
StatementMatcher LoopMatcher =
  forStmt (hasLoopInit (declStmt (hasSingleDecl (varDecl (
    hasInitializer (integerLiteral (equals(0))))))))).bind("forLoop");
```

Once you have defined your matchers, you will need to add a little more scaffolding in order to run them. Matchers are paired with a `MatchCallback` and registered with a `MatchFinder` object, then run from a `ClangTool`. More code!

Add the following to `LoopConvert.cpp`:

```
#include "clang/ASTMatchers/ASTMatchers.h"
#include "clang/ASTMatchers/ASTMatchFinder.h"

using namespace clang;
using namespace clang::ast_matchers;

StatementMatcher LoopMatcher =
  forStmt (hasLoopInit (declStmt (hasSingleDecl (varDecl (
    hasInitializer (integerLiteral (equals(0))))))))).bind("forLoop");

class LoopPrinter : public MatchFinder::MatchCallback {
public:
  virtual void run(const MatchFinder::MatchResult &Result) {
    if (const ForStmt *FS = Result.Nodes.getNodeAs<clang::ForStmt>("forLoop"))
      FS->dump();
  }
};
```

And change `main()` to:

```
int main(int argc, const char **argv) {
  CommonOptionsParser OptionsParser(argc, argv, MyToolCategory);
  ClangTool Tool(OptionsParser.getCompilations(),
    OptionsParser.getSourcePathList());

  LoopPrinter Printer;
  MatchFinder Finder;
  Finder.addMatcher(LoopMatcher, &Printer);

  return Tool.run(newFrontendActionFactory(&Finder).get());
}
```

Now, you should be able to recompile and run the code to discover for loops. Create a new file with a few examples, and test out our new handiwork:

```
cd ~/clang-llvm/llvm/llvm_build/
ninja loop-convert
```

```
vim ~/test-files/simple-loops.cc
bin/loop-convert ~/test-files/simple-loops.cc
```

Step 3.5: More Complicated Matchers

Our simple matcher is capable of discovering for loops, but we would still need to filter out many more ourselves. We can do a good portion of the remaining work with some cleverly chosen matchers, but first we need to decide exactly which properties we want to allow.

How can we characterize for loops over arrays which would be eligible for translation to range-based syntax? Range based loops over arrays of size N that:

- start at index 0
- iterate consecutively
- end at index $N-1$

We already check for (1), so all we need to add is a check to the loop's condition to ensure that the loop's index variable is compared against N and another check to ensure that the increment step just increments this same variable. The matcher for (2) is straightforward: require a pre- or post-increment of the same variable declared in the init portion.

Unfortunately, such a matcher is impossible to write. Matchers contain no logic for comparing two arbitrary AST nodes and determining whether or not they are equal, so the best we can do is matching more than we would like to allow, and punting extra comparisons to the callback.

In any case, we can start building this sub-matcher. We can require that the increment step be a unary increment like this:

```
hasIncrement (unaryOperator (hasOperatorName ("++")) )
```

Specifying what is incremented introduces another quirk of Clang's AST: Usages of variables are represented as `DeclRefExpr`'s ("declaration reference expressions") because they are expressions which refer to variable declarations. To find a `unaryOperator` that refers to a specific declaration, we can simply add a second condition to it:

```
hasIncrement (unaryOperator (
  hasOperatorName ("++"),
  hasUnaryOperand (declRefExpr ()) ) )
```

Furthermore, we can restrict our matcher to only match if the incremented variable is an integer:

```
hasIncrement (unaryOperator (
  hasOperatorName ("++"),
  hasUnaryOperand (declRefExpr (to (varDecl (hasType (isInteger ())))) ) ) )
```

And the last step will be to attach an identifier to this variable, so that we can retrieve it in the callback:

```
hasIncrement (unaryOperator (
  hasOperatorName ("++"),
  hasUnaryOperand (declRefExpr (to (
    varDecl (hasType (isInteger ())) .bind ("incrementVariable") ) ) ) ) )
```

We can add this code to the definition of `LoopMatcher` and make sure that our program, outfitted with the new matcher, only prints out loops that declare a single variable initialized to zero and have an increment step consisting of a unary increment of some variable.

Now, we just need to add a matcher to check if the condition part of the `for` loop compares a variable against the size of the array. There is only one problem - we don't know which array we're iterating over without looking at the body of the loop! We are again restricted to approximating the result we want with matchers, filling in the details in the callback. So we start with:

```
hasCondition(binaryOperator(hasOperatorName("<")))
```

It makes sense to ensure that the left-hand side is a reference to a variable, and that the right-hand side has integer type.

```
hasCondition(binaryOperator(
  hasOperatorName("<"),
  hasLHS(declRefExpr(to(varDecl(hasType(isInteger()))))),
  hasRHS(expr(hasType(isInteger())))))
```

Why? Because it doesn't work. Of the three loops provided in `test-files/simple.cpp`, zero of them have a matching condition. A quick look at the AST dump of the first `for` loop, produced by the previous iteration of `loop-convert`, shows us the answer:

```
(ForStmt 0x173b240
 (DeclStmt 0x173afc8
  0x173af50 "int i =
    (IntegerLiteral 0x173afa8 'int' 0)")
 <<>>
 (BinaryOperator 0x173b060 '_Bool' '<'
  (ImplicitCastExpr 0x173b030 'int'
   (DeclRefExpr 0x173afe0 'int' lvalue Var 0x173af50 'i' 'int'))
  (ImplicitCastExpr 0x173b048 'int'
   (DeclRefExpr 0x173b008 'const int' lvalue Var 0x170fa80 'N' 'const int'))
 (UnaryOperator 0x173b0b0 'int' lvalue prefix '++'
  (DeclRefExpr 0x173b088 'int' lvalue Var 0x173af50 'i' 'int'))
 (CompoundStatement ...
```

We already know that the declaration and increments both match, or this loop wouldn't have been dumped. The culprit lies in the implicit cast applied to the first operand (i.e. the LHS) of the less-than operator, an L-value to R-value conversion applied to the expression referencing `i`. Thankfully, the matcher library offers a solution to this problem in the form of `ignoringParenImpCasts`, which instructs the matcher to ignore implicit casts and parentheses before continuing to match. Adjusting the condition operator will restore the desired match.

```
hasCondition(binaryOperator(
  hasOperatorName("<"),
  hasLHS(ignoringParenImpCasts(declRefExpr(
    to(varDecl(hasType(isInteger()))))),
  hasRHS(expr(hasType(isInteger())))))
```

After adding binds to the expressions we wished to capture and extracting the identifier strings into variables, we have array-step-2 completed.

Step 4: Retrieving Matched Nodes

So far, the matcher callback isn't very interesting: it just dumps the loop's AST. At some point, we will need to make changes to the input source code. Next, we'll work on using the nodes we bound in the previous step.

The `MatchFinder::run()` callback takes a `MatchFinder::MatchResult&` as its parameter. We're most interested in its `Context` and `Nodes` members. Clang uses the `ASTContext` class to represent contextual information about the AST, as the name implies, though the most functionally important detail is that several operations

require an `ASTContext*` parameter. More immediately useful is the set of matched nodes, and how we retrieve them.

Since we bind three variables (identified by `ConditionVarName`, `InitVarName`, and `IncrementVarName`), we can obtain the matched nodes by using the `getNodeAs()` member function.

In `LoopConvert.cpp` add

```
#include "clang/AST/ASTContext.h"
```

Change `LoopMatcher` to

```
StatementMatcher LoopMatcher =
    forStmt (hasLoopInit (declStmt (
        hasSingleDecl (varDecl (hasInitializer (integerLiteral (equals (0))))
            .bind("initVarName"))),
        hasIncrement (unaryOperator (
            hasOperatorName ("++"),
            hasUnaryOperand (declRefExpr (
                to (varDecl (hasType (isInteger ())).bind("incVarName"))))),
        hasCondition (binaryOperator (
            hasOperatorName ("<"),
            hasLHS (ignoringParenImpCasts (declRefExpr (
                to (varDecl (hasType (isInteger ())).bind("condVarName"))))),
            hasRHS (expr (hasType (isInteger ())))).bind("forLoop");
```

And change `LoopPrinter::run` to

```
void LoopPrinter::run(const MatchFinder::MatchResult &Result) {
    ASTContext *Context = Result.Context;
    const ForStmt *FS = Result.Nodes.getStmtAs<ForStmt>("forLoop");
    // We do not want to convert header files!
    if (!FS || !Context->getSourceManager().isFromMainFile(FS->getForLoc()))
        return;
    const VarDecl *IncVar = Result.Nodes.getNodeAs<VarDecl>("incVarName");
    const VarDecl *CondVar = Result.Nodes.getNodeAs<VarDecl>("condVarName");
    const VarDecl *InitVar = Result.Nodes.getNodeAs<VarDecl>("initVarName");

    if (!areSameVariable(IncVar, CondVar) || !areSameVariable(IncVar, InitVar))
        return;
    llvm::outs() << "Potential array-based loop discovered.\n";
}
```

Clang associates a `VarDecl` with each variable to represent the variable's declaration. Since the "canonical" form of each declaration is unique by address, all we need to do is make sure neither `ValueDecl` (base class of `VarDecl`) is `NULL` and compare the canonical `Decls`.

```
static bool areSameVariable(const ValueDecl *First, const ValueDecl *Second) {
    return First && Second &&
        First->getCanonicalDecl() == Second->getCanonicalDecl();
}
```

If execution reaches the end of `LoopPrinter::run()`, we know that the loop shell that looks like

```
for (int i= 0; i < expr(); ++i) { ... }
```

For now, we will just print a message explaining that we found a loop. The next section will deal with recursively traversing the AST to discover all changes needed.

As a side note, it's not as trivial to test if two expressions are the same, though Clang has already done the hard work for us by providing a way to canonicalize expressions:

```
static bool areSameExpr(ASTContext *Context, const Expr *First,
                      const Expr *Second) {
    if (!First || !Second)
        return false;
    llvm::FoldingSetNodeID FirstID, SecondID;
    First->Profile(FirstID, *Context, true);
    Second->Profile(SecondID, *Context, true);
    return FirstID == SecondID;
}
```

This code relies on the comparison between two `llvm::FoldingSetNodeIDs`. As the documentation for `Stmt::Profile()` indicates, the `Profile()` member function builds a description of a node in the AST, based on its properties, along with those of its children. `FoldingSetNodeID` then serves as a hash we can use to compare expressions. We will need `areSameExpr` later. Before you run the new code on the additional loops added to `test-files/simple.cpp`, try to figure out which ones will be considered potentially convertible.

Matching the Clang AST

This document explains how to use Clang's `LibASTMatchers` to match interesting nodes of the AST and execute code that uses the matched nodes. Combined with [LibTooling](#), `LibASTMatchers` helps to write code-to-code transformation tools or query tools.

We assume basic knowledge about the Clang AST. See the [Introduction to the Clang AST](#) if you want to learn more about how the AST is structured.

Introduction

`LibASTMatchers` provides a domain specific language to create predicates on Clang's AST. This DSL is written in and can be used from C++, allowing users to write a single program to both match AST nodes and access the node's C++ interface to extract attributes, source locations, or any other information provided on the AST level.

AST matchers are predicates on nodes in the AST. Matchers are created by calling creator functions that allow building up a tree of matchers, where inner matchers are used to make the match more specific.

For example, to create a matcher that matches all class or union declarations in the AST of a translation unit, you can call `recordDecl()`. To narrow the match down, for example to find all class or union declarations with the name "Foo", insert a `hasName` matcher: the call `recordDecl(hasName("Foo"))` returns a matcher that matches classes or unions that are named "Foo", in any namespace. By default, matchers that accept multiple inner matchers use an implicit `allOf()`. This allows further narrowing down the match, for example to match all classes that are derived from "Bar": `recordDecl(hasName("Foo"), isDerivedFrom("Bar"))`.

How to create a matcher

With more than a thousand classes in the Clang AST, one can quickly get lost when trying to figure out how to create a matcher for a specific pattern. This section will teach you how to use a rigorous step-by-step pattern to build the matcher you are interested in. Note that there will always be matchers missing for some part of the AST. See the section about [how to write your own AST matchers](#) later in this document.

The precondition to using the matchers is to understand how the AST for what you want to match looks like. The [Introduction to the Clang AST](#) teaches you how to dump a translation unit's AST into a human readable format.

In general, the strategy to create the right matchers is:

1. Find the outermost class in Clang's AST you want to match.
2. Look at the AST Matcher Reference for matchers that either match the node you're interested in or narrow down attributes on the node.
3. Create your outer match expression. Verify that it works as expected.
4. Examine the matchers for what the next inner node you want to match is.
5. Repeat until the matcher is finished.

Binding nodes in match expressions

Matcher expressions allow you to specify which parts of the AST are interesting for a certain task. Often you will want to then do something with the nodes that were matched, like building source code transformations.

To that end, matchers that match specific AST nodes (so called node matchers) are bindable; for example, `recordDecl(hasName("MyClass")).bind("id")` will bind the matched `recordDecl` node to the string "id", to be later retrieved in the [match callback](#).

Writing your own matchers

There are multiple different ways to define a matcher, depending on its type and flexibility.

`VariadicDynCastAllOfMatcher<Base, Derived>`

Those match all nodes of type *Base* if they can be dynamically casted to *Derived*. The names of those matchers are nouns, which closely resemble *Derived*. `VariadicDynCastAllOfMatchers` are the backbone of the matcher hierarchy. Most often, your match expression will start with one of them, and you can *bind* the node they represent to ids for later processing.

`VariadicDynCastAllOfMatchers` are callable classes that model variadic template functions in C++03. They take an arbitrary number of `Matcher<Derived>` and return a `Matcher<Base>`.

`AST_MATCHER_P(Type, Name, ParamType, Param)`

Most matcher definitions use the matcher creation macros. Those define both the matcher of type `Matcher<Type>` itself, and a matcher-creation function named *Name* that takes a parameter of type *ParamType* and returns the corresponding matcher.

There are multiple matcher definition macros that deal with polymorphic return values and different parameter counts. See [ASTMatchersMacros.h](#).

Matcher creation functions

Matchers are generated by nesting calls to matcher creation functions. Most of the time those functions are either created by using `VariadicDynCastAllOfMatcher` or the matcher creation macros (see below). The free-standing functions are an indication that this matcher is just a combination of other matchers, as is for example the case with `callee`.

How To Setup Clang Tooling For LLVM

Clang Tooling provides infrastructure to write tools that need syntactic and semantic information about a program. This term also relates to a set of specific tools using this infrastructure (e.g. `clang-check`). This document provides information on how to set up and use Clang Tooling for the LLVM source code.

Introduction

Clang Tooling needs a compilation database to figure out specific build options for each file. Currently it can create a compilation database from the `compile_commands.json` file, generated by CMake. When invoking clang tools, you can either specify a path to a build directory using a command line parameter `-p` or let Clang Tooling find this file in your source tree. In either case you need to configure your build using CMake to use clang tools.

Setup Clang Tooling Using CMake and Make

If you intend to use make to build LLVM, you should have CMake 2.8.6 or later installed (can be found [here](#)).

First, you need to generate Makefiles for LLVM with CMake. You need to make a build directory and run CMake from it:

```
$ mkdir your/build/directory
$ cd your/build/directory
$ cmake -DCMAKE_EXPORT_COMPILE_COMMANDS=ON path/to/llvm/sources
```

If you want to use clang instead of GCC, you can add `-DCMAKE_C_COMPILER=/path/to/clang` `-DCMAKE_CXX_COMPILER=/path/to/clang++`. You can also use `ccmake`, which provides a curses interface to configure CMake variables for lazy people.

As a result, the new `compile_commands.json` file should appear in the current directory. You should link it to the LLVM source tree so that Clang Tooling is able to use it:

```
$ ln -s $PWD/compile_commands.json path/to/llvm/source/
```

Now you are ready to build and test LLVM using make:

```
$ make check-all
```

Using Clang Tools

After you completed the previous steps, you are ready to run clang tools. If you have a recent clang installed, you should have `clang-check` in `$PATH`. Try to run it on any `.cpp` file inside the LLVM source tree:

```
$ clang-check tools/clang/lib/Tooling/CompilationDatabase.cpp
```

If you're using vim, it's convenient to have clang-check integrated. Put this into your `.vimrc`:

```
function! ClangCheckImpl(cmd)
  if &autowrite | wall | endif
  echo "Running " . a:cmd . " ..."
  let l:output = system(a:cmd)
  cexpr l:output
  cwindow
  let w:quickfix_title = a:cmd
endfunction
```

```
if v:shell_error != 0
    cc
endif
let g:clang_check_last_cmd = a:cmd
endfunction

function! ClangCheck()
    let l:filename = expand('%')
    if l:filename =~ '\. \(cpp\|cxx\|cc\|c\) $'
        call ClangCheckImpl("clang-check " . l:filename)
    elseif exists("g:clang_check_last_cmd")
        call ClangCheckImpl(g:clang_check_last_cmd)
    else
        echo "Can't detect file's compilation arguments and no previous clang-check
↪ invocation!"
    endif
endfunction

nmap <silent> <F5> :call ClangCheck()<CR><CR>
```

When editing a .cpp/.cxx/.cc/.c file, hit F5 to reparse the file. In case the current file has a different extension (for example, .h), F5 will re-run the last clang-check invocation made from this vim instance (if any). The output will go into the error window, which is opened automatically when clang-check finds errors, and can be re-opened with :cope.

Other clang-check options that can be useful when working with clang AST:

- `-ast-print` — Build ASTs and then pretty-print them.
- `-ast-dump` — Build ASTs and then debug dump them.
- `-ast-dump-filter=<string>` — Use with `-ast-dump` or `-ast-print` to dump/print only AST declaration nodes having a certain substring in a qualified name. Use `-ast-list` to list all filterable declaration node names.
- `-ast-list` — Build ASTs and print the list of declaration node qualified names.

Examples:

```
$ clang-check tools/clang/tools/clang-check/ClangCheck.cpp -ast-dump -ast-dump-filter_
↪ ActionFactory::newASTConsumer
Processing: tools/clang/tools/clang-check/ClangCheck.cpp.
Dumping ::ActionFactory::newASTConsumer:
clang::ASTConsumer *newASTConsumer() (CompoundStmt 0x44da290 </home/alex/fh/local/llvm/
↪ tools/clang/tools/clang-check/ClangCheck.cpp:64:40, line:72:3>
  (IfStmt 0x44d97c8 <line:65:5, line:66:45>
    <<<NULL>>>
    (ImplicitCastExpr 0x44d96d0 <line:65:9> '_Bool': '_Bool' <UserDefinedConversion>
...
$ clang-check tools/clang/tools/clang-check/ClangCheck.cpp -ast-print -ast-dump-
↪ filter ActionFactory::newASTConsumer
Processing: tools/clang/tools/clang-check/ClangCheck.cpp.
Printing <anonymous namespace>::ActionFactory::newASTConsumer:
clang::ASTConsumer *newASTConsumer() {
    if (this->ASTList.operator _Bool())
        return clang::CreateASTDeclNodeLister();
    if (this->ASTDump.operator _Bool())
        return clang::CreateASTDumper(this->ASTDumpFilter);
    if (this->ASTPrint.operator _Bool())
        return clang::CreateASTPrinter(&llvm::outs(), this->ASTDumpFilter);
```



```
return new clang::ASTConsumer();
}
```

(Experimental) Using Ninja Build System

Optionally you can use the [Ninja](#) build system instead of make. It is aimed at making your builds faster. Currently this step will require building Ninja from sources.

To take advantage of using Clang Tools along with Ninja build you need at least CMake 2.8.9.

Clone the Ninja git repository and build Ninja from sources:

```
$ git clone git://github.com/martine/ninja.git
$ cd ninja/
$ ./bootstrap.py
```

This will result in a single binary `ninja` in the current directory. It doesn't require installation and can just be copied to any location inside `$PATH`, say `/usr/local/bin/`:

```
$ sudo cp ninja /usr/local/bin/
$ sudo chmod a+rx /usr/local/bin/ninja
```

After doing all of this, you'll need to generate Ninja build files for LLVM with CMake. You need to make a build directory and run CMake from it:

```
$ mkdir your/build/directory
$ cd your/build/directory
$ cmake -G Ninja -DCMAKE_EXPORT_COMPILE_COMMANDS=ON path/to/llvm/sources
```

If you want to use clang instead of GCC, you can add `-DCMAKE_C_COMPILER=/path/to/clang` `-DCMAKE_CXX_COMPILER=/path/to/clang++`. You can also use `ccmake`, which provides a curses interface to configure CMake variables in an interactive manner.

As a result, the new `compile_commands.json` file should appear in the current directory. You should link it to the LLVM source tree so that Clang Tooling is able to use it:

```
$ ln -s $PWD/compile_commands.json path/to/llvm/source/
```

Now you are ready to build and test LLVM using Ninja:

```
$ ninja check-all
```

Other target names can be used in the same way as with make.

JSON Compilation Database Format Specification

This document describes a format for specifying how to replay single compilations independently of the build system.

Background

Tools based on the C++ Abstract Syntax Tree need full information how to parse a translation unit. Usually this information is implicitly available in the build system, but running tools as part of the build system is not necessarily the best solution:

- Build systems are inherently change driven, so running multiple tools over the same code base without changing the code does not fit into the architecture of many build systems.
- Figuring out whether things have changed is often an IO bound process; this makes it hard to build low latency end user tools based on the build system.
- Build systems are inherently sequential in the build graph, for example due to generated source code. While tools that run independently of the build still need the generated source code to exist, running tools multiple times over unchanging source does not require serialization of the runs according to the build dependency graph.

Supported Systems

Currently [CMake](#) (since 2.8.5) supports generation of compilation databases for Unix Makefile builds (Ninja builds in the works) with the option `CMAKE_EXPORT_COMPILE_COMMANDS`.

For projects on Linux, there is an alternative to intercept compiler calls with a tool called [Bear](#).

Clang's tooling interface supports reading compilation databases; see the [LibTooling documentation](#). `libclang` and its python bindings also support this (since clang 3.2); see `CXCompilationDatabase.h`.

Format

A compilation database is a JSON file, which consist of an array of “command objects”, where each command object specifies one way a translation unit is compiled in the project.

Each command object contains the translation unit's main file, the working directory of the compile run and the actual compile command.

Example:

```
[
  {
    "directory": "/home/user/llvm/build",
    "command": "/usr/bin/clang++ -Irelative -DSOMEDEF=\"With spaces, quotes and \\-es.
    ↪\" -c -o file.o file.cc",
    "file": "file.cc" },
  ...
]
```

The contracts for each field in the command object are:

- **directory:** The working directory of the compilation. All paths specified in the **command** or **file** fields must be either absolute or relative to this directory.
- **file:** The main translation unit source processed by this compilation step. This is used by tools as the key into the compilation database. There can be multiple command objects for the same file, for example if the same source file is compiled with different configurations.
- **command:** The compile command executed. After JSON unescaping, this must be a valid command to rerun the exact compilation step for the translation unit in the environment the build system uses. Parameters use shell quoting and shell escaping of quotes, with ‘”’ and ‘\’ being the only special characters. Shell expansion is not supported.

Build System Integration

The convention is to name the file `compile_commands.json` and put it at the top of the build directory. Clang tools are pointed to the top of the build directory to detect the file and use the compilation database to parse C++ code in the source tree.

Using Clang Tools

Overview

Clang Tools are standalone command line (and potentially GUI) tools designed for use by C++ developers who are already using and enjoying Clang as their compiler. These tools provide developer-oriented functionality such as fast syntax checking, automatic formatting, refactoring, etc.

Only a couple of the most basic and fundamental tools are kept in the primary Clang Subversion project. The rest of the tools are kept in a side-project so that developers who don't want or need to build them don't. If you want to get access to the extra Clang Tools repository, simply check it out into the tools tree of your Clang checkout and follow the usual process for building and working with a combined LLVM/Clang checkout:

- With Subversion:
 - `cd llvm/tools/clang/tools`
 - `svn co http://llvm.org/svn/llvm-project/clang-tools-extra/trunk extra`
- Or with Git:
 - `cd llvm/tools/clang/tools`
 - `git clone http://llvm.org/git/clang-tools-extra.git extra`

This document describes a high-level overview of the organization of Clang Tools within the project as well as giving an introduction to some of the more important tools. However, it should be noted that this document is currently focused on Clang and Clang Tool developers, not on end users of these tools.

Clang Tools Organization

Clang Tools are CLI or GUI programs that are intended to be directly used by C++ developers. That is they are *not* primarily for use by Clang developers, although they are hopefully useful to C++ developers who happen to work on Clang, and we try to actively dogfood their functionality. They are developed in three components: the underlying infrastructure for building a standalone tool based on Clang, core shared logic used by many different tools in the form of refactoring and rewriting libraries, and the tools themselves.

The underlying infrastructure for Clang Tools is the [LibTooling](#) platform. See its documentation for much more detailed information about how this infrastructure works. The common refactoring and rewriting toolkit-style library is also part of LibTooling organizationally.

A few Clang Tools are developed along side the core Clang libraries as examples and test cases of fundamental functionality. However, most of the tools are developed in a side repository to provide easy separation from the core libraries. We intentionally do not support public libraries in the side repository, as we want to carefully review and find good APIs for libraries as they are lifted out of a few tools and into the core Clang library set.

Regardless of which repository Clang Tools' code resides in, the development process and practices for all Clang Tools are exactly those of Clang itself. They are entirely within the Clang *project*, regardless of the version control scheme.

Core Clang Tools

The core set of Clang tools that are within the main repository are tools that very specifically complement, and allow use and testing of *Clang* specific functionality.

`clang-check`

ClangCheck combines the LibTooling framework for running a Clang tool with the basic Clang diagnostics by syntax checking specific files in a fast, command line interface. It can also accept flags to re-display the diagnostics in different formats with different flags, suitable for use driving an IDE or editor. Furthermore, it can be used in fixit-mode to directly apply fixit-hints offered by clang. See [How To Setup Clang Tooling For LLVM](#) for instructions on how to setup and used *clang-check*.

`clang-format`

Clang-format is both a *library* and a *stand-alone tool* with the goal of automatically reformatting C++ sources files according to configurable style guides. To do so, clang-format uses Clang's `Lexer` to transform an input file into a token stream and then changes all the whitespace around those tokens. The goal is for clang-format to serve both as a user tool (ideally with powerful IDE integrations) and as part of other refactoring tools, e.g. to do a reformatting of all the lines changed during a renaming.

`clang-modernize`

`clang-modernize` migrates C++ code to use C++11 features where appropriate. Currently it can:

- convert loops to range-based for loops;
- convert null pointer constants (like `NULL` or `0`) to C++11 `nullptr`;
- replace the type specifier in variable declarations with the `auto` type specifier;
- add the `override` specifier to applicable member functions.

Extra Clang Tools

As various categories of Clang Tools are added to the extra repository, they'll be tracked here. The focus of this documentation is on the scope and features of the tools for other tool developers; each tool should provide its own user-focused documentation.

Ideas for new Tools

- C++ cast conversion tool. Will convert C-style casts ((type) value) to appropriate C++ cast (static_cast, const_cast or reinterpret_cast).
- Non-member begin() and end() conversion tool. Will convert foo.begin() into begin(foo) and similarly for end(), where foo is a standard container. We could also detect similar patterns for arrays.
- make_shared / make_unique conversion. Part of this transformation can be incorporated into the auto transformation. Will convert

```
std::shared_ptr<Foo> sp(new Foo);
std::unique_ptr<Foo> up(new Foo);

func(std::shared_ptr<Foo>(new Foo), bar());
```

into:

```
auto sp = std::make_shared<Foo>();
auto up = std::make_unique<Foo>(); // In C++14 mode.

// This also affects correctness. For the cases where bar() throws,
// make_shared() is safe and the original code may leak.
func(std::make_shared<Foo>(), bar());
```

- tr1 removal tool. Will migrate source code from using TR1 library features to C++11 library. For example:

```
#include <tr1/unordered_map>
int main()
{
    std::tr1::unordered_map <int, int> ma;
    std::cout << ma.size () << std::endl;
    return 0;
}
```

should be rewritten to:

```
#include <unordered_map>
int main()
{
    std::unordered_map <int, int> ma;
    std::cout << ma.size () << std::endl;
    return 0;
}
```

- A tool to remove auto. Will convert auto to an explicit type or add comments with deduced types. The motivation is that there are developers that don't want to use auto because they are afraid that they might lose control over their code.
- C++14: less verbose operator function objects (N3421). For example:

```
sort(v.begin(), v.end(), greater<ValueType>());
```

should be rewritten to:

```
sort(v.begin(), v.end(), greater<>());
```

ClangCheck

ClangCheck is a small wrapper around *LibTooling* which can be used to do basic error checking and AST dumping.

```
$ cat <<EOF > snippet.cc
> void f() {
>   int a = 0
> }
> EOF
$ ~/clang/build/bin/clang-check snippet.cc -ast-dump --
Processing: /Users/danieljasper/clang/llvm/tools/clang/docs/snippet.cc.
/Users/danieljasper/clang/llvm/tools/clang/docs/snippet.cc:2:12: error: expected ';'
↪at end of
    declaration
    int a = 0
        ^
    ;
(TranslationUnitDecl 0x7ff3a3029ed0 <<invalid sloc>>
 (TypedefDecl 0x7ff3a302a410 <<invalid sloc>> __int128_t '__int128')
 (TypedefDecl 0x7ff3a302a470 <<invalid sloc>> __uint128_t 'unsigned __int128')
 (TypedefDecl 0x7ff3a302a830 <<invalid sloc>> __builtin_va_list '__va_list_tag [1]')
 (FunctionDecl 0x7ff3a302a8d0 </Users/danieljasper/clang/llvm/tools/clang/docs/
 ↪snippet.cc:1:1, line:3:1> f 'void (void)')
 (CompoundStmt 0x7ff3a302aa10 <line:1:10, line:3:1>
 (DeclStmt 0x7ff3a302a9f8 <line:2:3, line:3:1>
 (VarDecl 0x7ff3a302a980 <line:2:3, col:11> a 'int'
 (IntegerLiteral 0x7ff3a302a9d8 <col:11> 'int' 0))))))
1 error generated.
Error while processing snippet.cc.
```

The ‘-’ at the end is important as it prevents *clang-check* from search for a compilation database. For more information on how to setup and use *clang-check* in a project, see *How To Setup Clang Tooling For LLVM*.

ClangFormat

ClangFormat describes a set of tools that are built on top of *LibFormat*. It can support your workflow in a variety of ways including a standalone tool and editor integrations.

Standalone Tool

clang-format is located in *clang/tools/clang-format* and can be used to format C/C++/Obj-C code.

```
$ clang-format -help
OVERVIEW: A tool to format C/C++/Obj-C code.

If no arguments are specified, it formats the code from standard input
and writes the result to the standard output.
If <file>s are given, it reformats the files. If -i is specified
together with <file>s, the files are edited in-place. Otherwise, the
result is written to the standard output.

USAGE: clang-format [options] [<file> ...]

OPTIONS:
```

Clang-format options:

- cursor=<uint> - The position of the cursor when invoking clang-format from an editor integration
- dump-config - Dump configuration options to stdout and exit. Can be used with -style option.
- i - Inplace edit <file>s, if specified.
- length=<uint> - Format a range of this length (in bytes). Multiple ranges can be formatted by specifying several -offset and -length pairs. When only a single -offset is specified without -length, clang-format will format up to the end of the file. Can only be used with one input file.
- lines=<string> - <start line>:<end line> - format a range of lines (both 1-based). Multiple ranges can be formatted by specifying several -lines arguments. Can't be used with -offset and -length. Can only be used with one input file.
- offset=<uint> - Format a range starting at this byte offset. Multiple ranges can be formatted by specifying several -offset and -length pairs. Can only be used with one input file.
- output-replacements-xml - Output replacements as XML.
- style=<string> - Coding style, currently supports:
 LLVM, Google, Chromium, Mozilla, WebKit.
 Use -style=file to load style configuration from .clang-format file located in one of the parent directories of the source file (or current directory for stdin).
 Use -style="{key: value, ...}" to set specific parameters, e.g.:
 -style="{BasedOnStyle: llvm, IndentWidth: 8}"

General options:

- help - Display available options (-help-hidden for more)
- help-list - Display list of available options (-help-list-hidden for more)
- version - Display the version of this program

When the desired code formatting style is different from the available options, the style can be customized using the `-style="{key: value, ...}"` option or by putting your style configuration in the `.clang-format` or `_clang-format` file in your project's directory and using `clang-format -style=file`.

An easy way to create the `.clang-format` file is:

```
clang-format -style=llvm -dump-config > .clang-format
```

Available style options are described in [Clang-Format Style Options](#).

Vim Integration

There is an integration for **vim** which lets you run the **clang-format** standalone tool on your current buffer, optionally selecting regions to reformat. The integration has the form of a *python*-file which can be found under

clang/tools/clang-format/clang-format.py.

This can be integrated by adding the following to your *.vimrc*:

```
map <C-K> :pyf <path-to-this-file>/clang-format.py<cr>
imap <C-K> <c-o>:pyf <path-to-this-file>/clang-format.py<cr>
```

The first line enables **clang-format** for NORMAL and VISUAL mode, the second line adds support for INSERT mode. Change “C-K” to another binding if you need **clang-format** on a different key (C-K stands for Ctrl+k).

With this integration you can press the bound key and clang-format will format the current line in NORMAL and INSERT mode or the selected region in VISUAL mode. The line or region is extended to the next bigger syntactic entity.

It operates on the current, potentially unsaved buffer and does not create or save any files. To revert a formatting, just undo.

Emacs Integration

Similar to the integration for **vim**, there is an integration for **emacs**. It can be found at *clang/tools/clang-format/clang-format.el* and used by adding this to your *.emacs*:

```
(load "<path-to-clang>/tools/clang-format/clang-format.el")
(global-set-key [C-M-tab] 'clang-format-region)
```

This binds the function *clang-format-region* to C-M-tab, which then formats the current line or selected region.

BBEdit Integration

clang-format cannot be used as a text filter with BBEdit, but works well via a script. The AppleScript to do this integration can be found at *clang/tools/clang-format/clang-format-bbedit.applescript*; place a copy in *~/Library/Application Support/BBEdit/Scripts*, and edit the path within it to point to your local copy of **clang-format**.

With this integration you can select the script from the Script menu and **clang-format** will format the selection. Note that you can rename the menu item by renaming the script, and can assign the menu item a keyboard shortcut in the BBEdit preferences, under Menus & Shortcuts.

Visual Studio Integration

Download the latest Visual Studio extension from the [alpha build site](#). The default key-binding is Ctrl-R, Ctrl-F.

Script for patch reformatting

The python script *clang/tools/clang-format-diff.py* parses the output of a unified diff and reformats all contained lines with **clang-format**.

```
usage: clang-format-diff.py [-h] [-i] [-p NUM] [-regex PATTERN] [-style STYLE]

Reformat changed lines in diff. Without -i option just output the diff that
would be introduced.

optional arguments:
  -h, --help            show this help message and exit
  -i                    apply edits to files instead of displaying a diff
```



```
-p NUM          strip the smallest prefix containing P slashes
-regex PATTERN  custom pattern selecting file paths to reformat
-style STYLE     formatting style to apply (LLVM, Google, Chromium, Mozilla,
                  WebKit)
```

So to reformat all the lines in the latest **git** commit, just do:

```
git diff -U0 HEAD^ | clang-format-diff.py -i -p1
```

In an SVN client, you can do:

```
svn diff --diff-cmd=diff -x-U0 | clang-format-diff.py -i
```

The `-U0` will create a diff without context lines (the script would format those as well).

Clang-Format Style Options

Clang-Format Style Options describes configurable formatting style options supported by *LibFormat* and *ClangFormat*.

When using **clang-format** command line utility or `clang::format::reformat(...)` functions from code, one can either use one of the predefined styles (LLVM, Google, Chromium, Mozilla, WebKit) or create a custom style by configuring specific style options.

Configuring Style with clang-format

clang-format supports two ways to provide custom style options: directly specify style configuration in the `-style=` command line option or use `-style=file` and put style configuration in the `.clang-format` or `_clang-format` file in the project directory.

When using `-style=file`, **clang-format** for each input file will try to find the `.clang-format` file located in the closest parent directory of the input file. When the standard input is used, the search is started from the current directory.

The `.clang-format` file uses YAML format:

```
key1: value1
key2: value2
# A comment.
...
```

The configuration file can consist of several sections each having different `Language:` parameter denoting the programming language this section of the configuration is targeted at. See the description of the **Language** option below for the list of supported languages. The first section may have no language set, it will set the default style options for all languages. Configuration sections for specific language will override options set in the default section.

When **clang-format** formats a file, it auto-detects the language using the file name. When formatting standard input or a file that doesn't have the extension corresponding to its language, `-assume-filename=` option can be used to override the file name **clang-format** uses to detect the language.

An example of a configuration file for multiple languages:

```
---
# We'll use defaults from the LLVM style, but with 4 columns indentation.
BasedOnStyle: LLVM
```

```
IndentWidth: 4
---
Language: Cpp
# Force pointers to the type for C++.
DerivePointerAlignment: false
PointerAlignment: Left
---
Language: JavaScript
# Use 100 columns for JS.
ColumnLimit: 100
---
Language: Proto
# Don't format .proto files.
DisableFormat: true
...
```

An easy way to get a valid `.clang-format` file containing all configuration options of a certain predefined style is:

```
clang-format -style=llvm -dump-config > .clang-format
```

When specifying configuration in the `-style=` option, the same configuration is applied for all input files. The format of the configuration is:

```
-style='{key1: value1, key2: value2, ...}'
```

Disabling Formatting on a Piece of Code

Clang-format understands also special comments that switch formatting in a delimited range. The code between a comment `// clang-format off` or `/* clang-format off */` up to a comment `// clang-format on` or `/* clang-format on */` will not be formatted. The comments themselves will be formatted (aligned) normally.

```
int formatted_code;
// clang-format off
    void    unformatted_code ;
// clang-format on
void formatted_code_again;
```

Configuring Style in Code

When using `clang::format::reformat(...)` functions, the format is specified by supplying the `clang::format::FormatStyle` structure.

Configurable Format Style Options

This section lists the supported style options. Value type is specified for each option. For enumeration types possible values are specified both as a C++ enumeration member (with a prefix, e.g. `LS_Auto`), and as a value usable in the configuration (without a prefix: `Auto`).

BasedOnStyle (**string**) The style used for all options not specifically set in the configuration.

This option is supported only in the **clang-format** configuration (both within `-style='{...}'` and the `.clang-format` file).

Possible values:

- LLVM A style complying with the [LLVM coding standards](#)
- Google A style complying with [Google’s C++ style guide](#)
- Chromium A style complying with [Chromium’s style guide](#)
- Mozilla A style complying with [Mozilla’s style guide](#)
- WebKit A style complying with [WebKit’s style guide](#)

AccessModifierOffset (int) The extra indent or outdent of access modifiers, e.g. `public:`.

AlignAfterOpenBracket (bool) If `true`, horizontally aligns arguments after an open bracket.

This applies to round brackets (parentheses), angle brackets and square brackets. This will result in formatings like code `someLongFunction(argument1, argument2);endcode`

AlignConsecutiveAssignments (bool) If `true`, aligns consecutive assignments.

This will align the assignment operators of consecutive lines. This will result in formatings like code `int aaaa = 12; int b = 23; int ccc = 23;endcode`

AlignEscapedNewlinesLeft (bool) If `true`, aligns escaped newlines as far left as possible. Otherwise puts them into the right-most column.

AlignOperands (bool) If `true`, horizontally align operands of binary and ternary expressions.

AlignTrailingComments (bool) If `true`, aligns trailing comments.

AllowAllParametersOfDeclarationOnNextLine (bool) Allow putting all parameters of a function declaration onto the next line even if `BinPackParameters` is `false`.

AllowShortBlocksOnASingleLine (bool) Allows contracting simple braced statements to a single line.

E.g., this allows `if (a) { return; }` to be put on a single line.

AllowShortCaseLabelsOnASingleLine (bool) If `true`, short case labels will be contracted to a single line.

AllowShortFunctionsOnASingleLine (ShortFunctionStyle) Dependent on the value, `int f() { return 0; }` can be put on a single line.

Possible values:

- `SFS_None` (in configuration: `None`) Never merge functions into a single line.
- `SFS_Empty` (in configuration: `Empty`) Only merge empty functions.
- `SFS_Inline` (in configuration: `Inline`) Only merge functions defined inside a class. Implies “empty”.
- `SFS_All` (in configuration: `All`) Merge all functions fitting on a single line.

AllowShortIfStatementsOnASingleLine (bool) If `true`, `if (a) return;` can be put on a single line.

AllowShortLoopsOnASingleLine (bool) If `true`, `while (true) continue;` can be put on a single line.

AlwaysBreakAfterDefinitionReturnType (DefinitionReturnTypeBreakingStyle) The function definition return type breaking style to use.

Possible values:

- `DRTBS_None` (in configuration: `None`) Break after return type automatically. `PenaltyReturnTypeOnItsOwnLine` is taken into account.
- `DRTBS_All` (in configuration: `All`) Always break after the return type.
- `DRTBS_TopLevel` (in configuration: `TopLevel`) Always break after the return types of top level functions.

AlwaysBreakBeforeMultilineStrings (bool) If `true`, always break before multiline string literals.

This flag is mean to make cases where there are multiple multiline strings in a file look more consistent. Thus, it will only take effect if wrapping the string at that point leads to it being indented `ContinuationIndentWidth` spaces from the start of the line.

AlwaysBreakTemplateDeclarations (bool) If `true`, always break after the `template<...>` of a template declaration.

BinPackArguments (bool) If `false`, a function call's arguments will either be all on the same line or will have one line each.

BinPackParameters (bool) If `false`, a function declaration's or function definition's parameters will either all be on the same line or will have one line each.

BreakBeforeBinaryOperators (BinaryOperatorStyle) The way to wrap binary operators.

Possible values:

- `BOS_None` (in configuration: `None`) Break after operators.
- `BOS_NonAssignment` (in configuration: `NonAssignment`) Break before operators that aren't assignments.
- `BOS_All` (in configuration: `All`) Break before operators.

BreakBeforeBraces (BraceBreakingStyle) The brace breaking style to use.

Possible values:

- `BS_Attach` (in configuration: `Attach`) Always attach braces to surrounding context.
- `BS_Linux` (in configuration: `Linux`) Like `Attach`, but break before braces on function, namespace and class definitions.
- `BS_Mozilla` (in configuration: `Mozilla`) Like `Attach`, but break before braces on enum, function, and record definitions.
- `BS_Stroustrup` (in configuration: `Stroustrup`) Like `Attach`, but break before function definitions, and `'else'`.
- `BS_Allman` (in configuration: `Allman`) Always break before braces.
- `BS_GNU` (in configuration: `GNU`) Always break before braces and add an extra level of indentation to braces of control statements, not to those of class, function or other definitions.

BreakBeforeTernaryOperators (bool) If `true`, ternary operators will be placed after line breaks.

BreakConstructorInitializersBeforeComma (bool) Always break constructor initializers before commas and align the commas with the colon.

ColumnLimit (unsigned) The column limit.

A column limit of 0 means that there is no column limit. In this case, clang-format will respect the input's line breaking decisions within statements unless they contradict other rules.

CommentPragmas (std::string) A regular expression that describes comments with special meaning, which should not be split into lines or otherwise changed.

ConstructorInitializerAllOnOneLineOrOnePerLine (bool) If the constructor initializers don't fit on a line, put each initializer on its own line.

ConstructorInitializerIndentWidth (unsigned) The number of characters to use for indentation of constructor initializer lists.

ContinuationIndentWidth (unsigned) Indent width for line continuations.

Cpp11BracedListStyle (bool) If `true`, format braced lists as best suited for C++11 braced lists.

Important differences: - No spaces inside the braced list. - No line break before the closing brace. - Indentation with the continuation indent, not with the block indent.

Fundamentally, C++11 braced lists are formatted exactly like function calls would be formatted in their place. If the braced list follows a name (e.g. a type or variable name), clang-format formats as if the `{ }` were the parentheses of a function call with that name. If there is no name, a zero-length name is assumed.

DerivePointerAlignment (bool) If `true`, analyze the formatted file for the most common alignment of `&` and `*`. `PointerAlignment` is then used only as fallback.

DisableFormat (bool) Disables formatting completely.

ExperimentalAutoDetectBinPacking (bool) If `true`, clang-format detects whether function calls and definitions are formatted with one parameter per line.

Each call can be bin-packed, one-per-line or inconclusive. If it is inconclusive, e.g. completely on one line, but a decision needs to be made, clang-format analyzes whether there are other bin-packed cases in the input file and act accordingly.

NOTE: This is an experimental flag, that might go away or be renamed. Do not use this in config files, etc. Use at your own risk.

ForEachMacros (std::vector<std::string>) A vector of macros that should be interpreted as foreach loops instead of as function calls.

These are expected to be macros of the form: `code FOREACH(<variable-declaration>, ...) <loop-body> end-code`

For example: `BOOST_FOREACH`.

IndentCaseLabels (bool) Indent case labels one level from the switch statement.

When `false`, use the same indentation level as for the switch statement. Switch statement body is always indented one level more than case labels.

IndentWidth (unsigned) The number of columns to use for indentation.

IndentWrappedFunctionNames (bool) Indent if a function definition or declaration is wrapped after the type.

KeepEmptyLinesAtTheStartOfBlocks (bool) If true, empty lines at the start of blocks are kept.

Language (LanguageKind) Language, this format style is targeted at.

Possible values:

- `LK_None` (in configuration: `None`) Do not use.
- `LK_Cpp` (in configuration: `Cpp`) Should be used for C, C++, ObjectiveC, ObjectiveC++.
- `LK_Java` (in configuration: `Java`) Should be used for Java.
- `LK_JavaScript` (in configuration: `JavaScript`) Should be used for JavaScript.
- `LK_Proto` (in configuration: `Proto`) Should be used for Protocol Buffers (<https://developers.google.com/protocol-buffers/>).

MacroBlockBegin (std::string) A regular expression matching macros that start a block.

MacroBlockEnd (std::string) A regular expression matching macros that end a block.

MaxEmptyLinesToKeep (unsigned) The maximum number of consecutive empty lines to keep.

NamespaceIndentation (NamespaceIndentationKind) The indentation used for namespaces.

Possible values:

- `NI_None` (in configuration: `None`) Don't indent in namespaces.
- `NI_Inner` (in configuration: `Inner`) Indent only in inner namespaces (nested in other namespaces).
- `NI_All` (in configuration: `All`) Indent in all namespaces.

ObjCBlockIndentWidth (unsigned) The number of characters to use for indentation of ObjC blocks.

ObjCSpaceAfterProperty (bool) Add a space after `@property` in Objective-C, i.e. use `\@property` (`readonly`) instead of `\@property(readonly)`.

ObjCSpaceBeforeProtocolList (bool) Add a space in front of an Objective-C protocol list, i.e. use `Foo <Protocol>` instead of `Foo<Protocol>`.

PenaltyBreakBeforeFirstCallParameter (unsigned) The penalty for breaking a function call after “call”.

PenaltyBreakComment (unsigned) The penalty for each line break introduced inside a comment.

PenaltyBreakFirstLessLess (unsigned) The penalty for breaking before the first `<<`.

PenaltyBreakString (unsigned) The penalty for each line break introduced inside a string literal.

PenaltyExcessCharacter (unsigned) The penalty for each character outside of the column limit.

PenaltyReturnTypeOnItsOwnLine (unsigned) Penalty for putting the return type of a function onto its own line.

PointerAlignment (PointerAlignmentStyle) Pointer and reference alignment style.

Possible values:

- `PAS_Left` (in configuration: `Left`) Align pointer to the left.
- `PAS_Right` (in configuration: `Right`) Align pointer to the right.
- `PAS_Middle` (in configuration: `Middle`) Align pointer in the middle.

SpaceAfterCStyleCast (bool) If `true`, a space may be inserted after C style casts.

SpaceBeforeAssignmentOperators (bool) If `false`, spaces will be removed before assignment operators.

SpaceBeforeParens (SpaceBeforeParensOptions) Defines in which cases to put a space before opening parentheses.

Possible values:

- `SBPO_Never` (in configuration: `Never`) Never put a space before opening parentheses.
- `SBPO_ControlStatements` (in configuration: `ControlStatements`) Put a space before opening parentheses only after control statement keywords (`for/if/while...`).
- `SBPO_Always` (in configuration: `Always`) Always put a space before opening parentheses, except when it's prohibited by the syntax rules (in function-like macro definitions) or when determined by other style rules (after unary operators, opening parentheses, etc.)

SpaceInEmptyParentheses (bool) If `true`, spaces may be inserted into ‘()’.

SpacesBeforeTrailingComments (unsigned) The number of spaces before trailing line comments (`//` - comments).

This does not affect trailing block comments (`/**/` - comments) as those commonly have different usage patterns and a number of special cases.

SpacesInAngles (bool) If `true`, spaces will be inserted after ‘<’ and before ‘>’ in template argument lists

SpacesInCStyleCastParentheses (bool) If `true`, spaces may be inserted into C style casts.

SpacesInContainerLiterals (bool) If `true`, spaces are inserted inside container literals (e.g. ObjC and Javascript array and dict literals).

SpacesInParentheses (bool) If `true`, spaces will be inserted after ‘(’ and before ‘)’.

SpacesInSquareBrackets (bool) If `true`, spaces will be inserted after ‘[’ and before ‘]’.

Standard (LanguageStandard) Format compatible with this standard, e.g. use `A<A<int>> >` instead of `A<A<int>>>` for `LS_Cpp03`.

Possible values:

- `LS_Cpp03` (in configuration: `Cpp03`) Use C++03-compatible syntax.
- `LS_Cpp11` (in configuration: `Cpp11`) Use features of C++11 (e.g. `A<A<int>>>` instead of `A<A<int>>>`).
- `LS_Auto` (in configuration: `Auto`) Automatic detection based on the input.

TabWidth (unsigned) The number of columns used for tab stops.

UseTab (UseTabStyle) The way to use tab characters in the resulting file.

Possible values:

- `UT_Never` (in configuration: `Never`) Never use tab.
- `UT_ForIndentation` (in configuration: `ForIndentation`) Use tabs only for indentation.
- `UT_Always` (in configuration: `Always`) Use tabs whenever we need to fill whitespace that spans at least from one tab stop to the next one.

Examples

A style similar to the [Linux Kernel style](#):

```
BasedOnStyle: LLVM
IndentWidth: 8
UseTab: Always
BreakBeforeBraces: Linux
AllowShortIfStatementsOnASingleLine: false
IndentCaseLabels: false
```

The result is (imagine that tabs are used for indentation here):

```
void test()
{
    switch (x) {
    case 0:
    case 1:
        do_something();
        break;
    case 2:
        do_something_else();
        break;
    default:
        break;
    }
    if (condition)
        do_something_completely_different();

    if (x == y) {
        q();
    } else if (x > y) {
```

```
        w();
    } else {
        r();
    }
}
```

A style similar to the default Visual Studio formatting style:

```
UseTab: Never
IndentWidth: 4
BreakBeforeBraces: Allman
AllowShortIfStatementsOnASingleLine: false
IndentCaseLabels: false
ColumnLimit: 0
```

The result is:

```
void test()
{
    switch (suffix)
    {
        case 0:
        case 1:
            do_something();
            break;
        case 2:
            do_something_else();
            break;
        default:
            break;
    }
    if (condition)
        do_somthing_completely_different();

    if (x == y)
    {
        q();
    }
    else if (x > y)
    {
        w();
    }
    else
    {
        r();
    }
}
```


“Clang” CFE Internals Manual

- *Introduction*
- *LLVM Support Library*
- *The Clang “Basic” Library*
 - *The Diagnostics Subsystem*
 - * *The Diagnostic*Kinds.td files*
 - * *The Format String*
 - * *Formatting a Diagnostic Argument*
 - * *Producing the Diagnostic*
 - * *Fix-It Hints*
 - * *The DiagnosticClient Interface*
 - * *Adding Translations to Clang*
 - *The SourceLocation and SourceManager classes*
 - *SourceRange and CharSourceRange*
- *The Driver Library*
- *Precompiled Headers*
- *The Frontend Library*
- *The Lexer and Preprocessor Library*
 - *The Token class*

- *Annotation Tokens*
- *The `Lexer` class*
- *The `TokenLexer` class*
- *The `MultipleIncludeOpt` class*
- *The Parser Library*
- *The AST Library*
 - *The `Type` class and its subclasses*
 - * *Canonical Types*
 - *The `QualType` class*
 - *Declaration names*
 - *Declaration contexts*
 - * *Redeclarations and Overloads*
 - * *Lexical and Semantic Contexts*
 - * *Transparent Declaration Contexts*
 - * *Multiply-Defined Declaration Contexts*
 - *The `CFG` class*
 - * *Basic Blocks*
 - * *Entry and Exit Blocks*
 - * *Conditional Control-Flow*
 - *Constant Folding in the Clang AST*
 - * *Implementation Approach*
 - * *Extensions*
- *The Sema Library*
- *The CodeGen Library*
- *How to change Clang*
 - *How to add an attribute*
 - * *Attribute Basics*
 - * *`include/clang/Basic/Attr.td`*
 - *Spellings*
 - *Subjects*
 - *Documentation*
 - *Arguments*
 - *Other Properties*
 - * *Boilerplate*
 - * *Semantic handling*

Introduction

This document describes some of the more important APIs and internal design decisions made in the Clang C front-end. The purpose of this document is to both capture some of this high level information and also describe some of the design decisions behind it. This is meant for people interested in hacking on Clang, not for end-users. The description below is categorized by libraries, and does not describe any of the clients of the libraries.

LLVM Support Library

The LLVM `libSupport` library provides many underlying libraries and [data-structures](#), including command line option processing, various containers and a system abstraction layer, which is used for file system access.

The Clang “Basic” Library

This library certainly needs a better name. The “basic” library contains a number of low-level utilities for tracking and manipulating source buffers, locations within the source buffers, diagnostics, tokens, target abstraction, and information about the subset of the language being compiled for.

Part of this infrastructure is specific to C (such as the `TargetInfo` class), other parts could be reused for other non-C-based languages (`SourceLocation`, `SourceManager`, `Diagnostics`, `FileManager`). When and if there is future demand we can figure out if it makes sense to introduce a new library, move the general classes somewhere else, or introduce some other solution.

We describe the roles of these classes in order of their dependencies.

The Diagnostics Subsystem

The Clang Diagnostics subsystem is an important part of how the compiler communicates with the human. Diagnostics are the warnings and errors produced when the code is incorrect or dubious. In Clang, each diagnostic produced has (at the minimum) a unique ID, an English translation associated with it, a [SourceLocation](#) to “put the caret”, and a severity (e.g., `WARNING` or `ERROR`). They can also optionally include a number of arguments to the diagnostic (which fill in “%0”s in the string) as well as a number of source ranges that related to the diagnostic.

In this section, we’ll be giving examples produced by the Clang command line driver, but diagnostics can be [rendered in many different ways](#) depending on how the `DiagnosticClient` interface is implemented. A representative example of a diagnostic is:

```
t.c:38:15: error: invalid operands to binary expression ('int *' and '_Complex float')
P = (P-42) + Gamma*4;
      ~~~~~ ^ ~~~~~
```

In this example, you can see the English translation, the severity (error), you can see the source location (the caret (“^”) and file/line/column info), the source ranges “~~~~”, arguments to the diagnostic (“int*” and “_Complex float”). You’ll have to believe me that there is a unique ID backing the diagnostic :).

Getting all of this to happen has several steps and involves many moving pieces, this section describes them and talks about best practices when adding a new diagnostic.

The Diagnostic*Kinds.td files

Diagnostics are created by adding an entry to one of the `clang/Basic/Diagnostic*Kinds.td` files, depending on what library will be using it. From this file, **tblgen** generates the unique ID of the diagnostic, the severity of the diagnostic and the English translation + format string.

There is little sanity with the naming of the unique ID's right now. Some start with `err_`, `warn_`, `ext_` to encode the severity into the name. Since the enum is referenced in the C++ code that produces the diagnostic, it is somewhat useful for it to be reasonably short.

The severity of the diagnostic comes from the set `{NOTE, REMARK, WARNING, EXTENSION, EXTWARN, ERROR}`. The `ERROR` severity is used for diagnostics indicating the program is never acceptable under any circumstances. When an error is emitted, the AST for the input code may not be fully built. The `EXTENSION` and `EXTWARN` severities are used for extensions to the language that Clang accepts. This means that Clang fully understands and can represent them in the AST, but we produce diagnostics to tell the user their code is non-portable. The difference is that the former are ignored by default, and the later warn by default. The `WARNING` severity is used for constructs that are valid in the currently selected source language but that are dubious in some way. The `REMARK` severity provides generic information about the compilation that is not necessarily related to any dubious code. The `NOTE` level is used to staple more information onto previous diagnostics.

These *severities* are mapped into a smaller set (the `Diagnostic::Level` enum, `{Ignored, Note, Remark, Warning, Error, Fatal}`) of output *levels* by the diagnostics subsystem based on various configuration options. Clang internally supports a fully fine grained mapping mechanism that allows you to map almost any diagnostic to the output level that you want. The only diagnostics that cannot be mapped are `NOTES`, which always follow the severity of the previously emitted diagnostic and `ERRORS`, which can only be mapped to `Fatal` (it is not possible to turn an error into a warning, for example).

Diagnostic mappings are used in many ways. For example, if the user specifies `-pedantic`, `EXTENSION` maps to `Warning`, if they specify `-pedantic-errors`, it turns into `Error`. This is used to implement options like `-Wunused_macros`, `-Wundef` etc.

Mapping to `Fatal` should only be used for diagnostics that are considered so severe that error recovery won't be able to recover sensibly from them (thus spewing a ton of bogus errors). One example of this class of error are failure to `#include` a file.

The Format String

The format string for the diagnostic is very simple, but it has some power. It takes the form of a string in English with markers that indicate where and how arguments to the diagnostic are inserted and formatted. For example, here are some simple format strings:

```
"binary integer literals are an extension"
"format string contains '\\0' within the string body"
"more '%" conversions than data arguments"
"invalid operands to binary expression (%0 and %1)"
"overloaded '%0' must be a %select{unary|binary|unary or binary}2 operator"
    " (has %1 parameter%s1)"
```

These examples show some important points of format strings. You can use any plain ASCII character in the diagnostic string except “%” without a problem, but these are C strings, so you have to use and be aware of all the C escape sequences (as in the second example). If you want to produce a “%” in the output, use the “%%” escape sequence, like the third diagnostic. Finally, Clang uses the “%...[digit]” sequences to specify where and how arguments to the diagnostic are formatted.

Arguments to the diagnostic are numbered according to how they are specified by the C++ code that *produces them*, and are referenced by `%0 .. %9`. If you have more than 10 arguments to your diagnostic, you are doing something

wrong :). Unlike `printf`, there is no requirement that arguments to the diagnostic end up in the output in the same order as they are specified, you could have a format string with “%1 %0” that swaps them, for example. The text in between the percent and digit are formatting instructions. If there are no instructions, the argument is just turned into a string and substituted in.

Here are some “best practices” for writing the English format string:

- Keep the string short. It should ideally fit in the 80 column limit of the `DiagnosticKinds.td` file. This avoids the diagnostic wrapping when printed, and forces you to think about the important point you are conveying with the diagnostic.
- Take advantage of location information. The user will be able to see the line and location of the caret, so you don’t need to tell them that the problem is with the 4th argument to the function: just point to it.
- Do not capitalize the diagnostic string, and do not end it with a period.
- If you need to quote something in the diagnostic string, use single quotes.

Diagnostics should never take random English strings as arguments: you shouldn’t use “you have a problem with %0” and pass in things like “your argument” or “your return value” as arguments. Doing this prevents *translating* the Clang diagnostics to other languages (because they’ll get random English words in their otherwise localized diagnostic). The exceptions to this are C/C++ language keywords (e.g., `auto`, `const`, `mutable`, etc) and C/C++ operators (`/=`). Note that things like “pointer” and “reference” are not keywords. On the other hand, you *can* include anything that comes from the user’s source code, including variable names, types, labels, etc. The “select” format can be used to achieve this sort of thing in a localizable way, see below.

Formatting a Diagnostic Argument

Arguments to diagnostics are fully typed internally, and come from a couple different classes: integers, types, names, and random strings. Depending on the class of the argument, it can be optionally formatted in different ways. This gives the `DiagnosticClient` information about what the argument means without requiring it to use a specific presentation (consider this MVC for Clang :).

Here are the different diagnostic argument formats currently supported by Clang:

“s” format

Example: `"requires %1 parameter%s1"`

Class: Integers

Description: This is a simple formatter for integers that is useful when producing English diagnostics. When the integer is 1, it prints as nothing. When the integer is not 1, it prints as “s”. This allows some simple grammatical forms to be to be handled correctly, and eliminates the need to use gross things like `"requires %1 parameter(s) "`.

“select” format

Example: `"must be a %select{unary|binary|unary or binary}2 operator"`

Class: Integers

Description: This format specifier is used to merge multiple related diagnostics together into one common one, without requiring the difference to be specified as an English string argument. Instead of specifying the string, the diagnostic gets an integer argument and the format string selects the numbered option. In this case, the “%2” value must be an integer in the range [0..2]. If it is 0, it prints “unary”, if it is 1 it prints “binary” if it is 2, it prints “unary or binary”. This allows other language translations to substitute reasonable words (or entire phrases) based on the semantics of the diagnostic instead of having to do things textually. The selected string does undergo formatting.

“plural” format

Example: `"you have %1 %plural{1:mouse|mice}1 connected to your computer"`

Class: `Integers`

Description: This is a formatter for complex plural forms. It is designed to handle even the requirements of languages with very complex plural forms, as many Baltic languages have. The argument consists of a series of expression/form pairs, separated by “:”, where the first form whose expression evaluates to true is the result of the modifier.

An expression can be empty, in which case it is always true. See the example at the top. Otherwise, it is a series of one or more numeric conditions, separated by “,”. If any condition matches, the expression matches. Each numeric condition can take one of three forms.

- **number:** A simple decimal number matches if the argument is the same as the number. Example: `"%plural{1:mouse|mice}4"`
- **range:** A range in square brackets matches if the argument is within the range. Then range is inclusive on both ends. Example: `"%plural{0:none|1:one|[2,5]:some|:many}2"`
- **modulo:** A modulo operator is followed by a number, and equals sign and either a number or a range. The tests are the same as for plain numbers and ranges, but the argument is taken modulo the number first. Example: `"%plural{%100=0:even hundred|%100=[1,50]:lower half|:everything else}1"`

The parser is very unforgiving. A syntax error, even whitespace, will abort, as will a failure to match the argument against any expression.

“ordinal” format

Example: `"ambiguity in %ordinal0 argument"`

Class: `Integers`

Description: This is a formatter which represents the argument number as an ordinal: the value 1 becomes 1st, 3 becomes 3rd, and so on. Values less than 1 are not supported. This formatter is currently hard-coded to use English ordinals.

“objcclass” format

Example: `"method %objcclass0 not found"`

Class: `DeclarationName`

Description: This is a simple formatter that indicates the `DeclarationName` corresponds to an Objective-C class method selector. As such, it prints the selector with a leading “+”.

“objcinstance” format

Example: `"method %objcinstance0 not found"`

Class: `DeclarationName`

Description: This is a simple formatter that indicates the `DeclarationName` corresponds to an Objective-C instance method selector. As such, it prints the selector with a leading “-”.

“q” format

Example: `"candidate found by name lookup is %q0"`

Class: `NamedDecl *`

Description: This formatter indicates that the fully-qualified name of the declaration should be printed, e.g., “std::vector” rather than “vector”.

“diff” format

Example: "no known conversion %diff{from \$ to \$|from argument type to parameter type}%1,2"

Class: `QualType`

Description: This formatter takes two `QualTypes` and attempts to print a template difference between the two. If tree printing is off, the text inside the braces before the pipe is printed, with the formatted text replacing the \$. If tree printing is on, the text after the pipe is printed and a type tree is printed after the diagnostic message.

It is really easy to add format specifiers to the Clang diagnostics system, but they should be discussed before they are added. If you are creating a lot of repetitive diagnostics and/or have an idea for a useful formatter, please bring it up on the cfe-dev mailing list.

Producing the Diagnostic

Now that you've created the diagnostic in the `Diagnostic*Kinds.td` file, you need to write the code that detects the condition in question and emits the new diagnostic. Various components of Clang (e.g., the preprocessor, Sema, etc.) provide a helper function named "Diag". It creates a diagnostic and accepts the arguments, ranges, and other information that goes along with it.

For example, the binary expression error comes from code like this:

```
if (various things that are bad)
    Diag(Loc, diag::err_typecheck_invalid_operands)
        << lex->getType() << rex->getType()
        << lex->getSourceRange() << rex->getSourceRange();
```

This shows that use of the `Diag` method: it takes a location (a [SourceLocation](#) object) and a diagnostic enum value (which matches the name from `Diagnostic*Kinds.td`). If the diagnostic takes arguments, they are specified with the `<<` operator: the first argument becomes `%0`, the second becomes `%1`, etc. The diagnostic interface allows you to specify arguments of many different types, including `int` and `unsigned` for integer arguments, `const char*` and `std::string` for string arguments, `DeclarationName` and `const IdentifierInfo *` for names, `QualType` for types, etc. `SourceRanges` are also specified with the `<<` operator, but do not have a specific ordering requirement.

As you can see, adding and producing a diagnostic is pretty straightforward. The hard part is deciding exactly what you need to say to help the user, picking a suitable wording, and providing the information needed to format it correctly. The good news is that the call site that issues a diagnostic should be completely independent of how the diagnostic is formatted and in what language it is rendered.

Fix-It Hints

In some cases, the front end emits diagnostics when it is clear that some small change to the source code would fix the problem. For example, a missing semicolon at the end of a statement or a use of deprecated syntax that is easily rewritten into a more modern form. Clang tries very hard to emit the diagnostic and recover gracefully in these and other cases.

However, for these cases where the fix is obvious, the diagnostic can be annotated with a hint (referred to as a "fix-it hint") that describes how to change the code referenced by the diagnostic to fix the problem. For example, it might add the missing semicolon at the end of the statement or rewrite the use of a deprecated construct into something more palatable. Here is one such example from the C++ front end, where we warn about the right-shift operator changing meaning from C++98 to C++11:

```
test.cpp:3:7: warning: use of right-shift operator ('>>') in template argument
                    will require parentheses in C++11
A<100 >> 2> *a;
```

```
    ^  
(      )
```

Here, the fix-it hint is suggesting that parentheses be added, and showing exactly where those parentheses would be inserted into the source code. The fix-it hints themselves describe what changes to make to the source code in an abstract manner, which the text diagnostic printer renders as a line of “insertions” below the caret line. *Other diagnostic clients* might choose to render the code differently (e.g., as markup inline) or even give the user the ability to automatically fix the problem.

Fix-it hints on errors and warnings need to obey these rules:

- Since they are automatically applied if `-Xclang -fixit` is passed to the driver, they should only be used when it’s very likely they match the user’s intent.
- Clang must recover from errors as if the fix-it had been applied.

If a fix-it can’t obey these rules, put the fix-it on a note. Fix-its on notes are not applied automatically.

All fix-it hints are described by the `FixItHint` class, instances of which should be attached to the diagnostic using the `<<` operator in the same way that highlighted source ranges and arguments are passed to the diagnostic. Fix-it hints can be created with one of three constructors:

- `FixItHint::CreateInsertion(Loc, Code)`
Specifies that the given `Code` (a string) should be inserted before the source location `Loc`.
- `FixItHint::CreateRemoval(Range)`
Specifies that the code in the given source `Range` should be removed.
- `FixItHint::CreateReplacement(Range, Code)`
Specifies that the code in the given source `Range` should be removed, and replaced with the given `Code` string.

The DiagnosticClient Interface

Once code generates a diagnostic with all of the arguments and the rest of the relevant information, Clang needs to know what to do with it. As previously mentioned, the diagnostic machinery goes through some filtering to map a severity onto a diagnostic level, then (assuming the diagnostic is not mapped to “Ignore”) it invokes an object that implements the `DiagnosticClient` interface with the information.

It is possible to implement this interface in many different ways. For example, the normal Clang `DiagnosticClient` (named `TextDiagnosticPrinter`) turns the arguments into strings (according to the various formatting rules), prints out the file/line/column information and the string, then prints out the line of code, the source ranges, and the caret. However, this behavior isn’t required.

Another implementation of the `DiagnosticClient` interface is the `TextDiagnosticBuffer` class, which is used when Clang is in `-verify` mode. Instead of formatting and printing out the diagnostics, this implementation just captures and remembers the diagnostics as they fly by. Then `-verify` compares the list of produced diagnostics to the list of expected ones. If they disagree, it prints out its own output. Full documentation for the `-verify` mode can be found in the Clang API documentation for `VerifyDiagnosticConsumer`.

There are many other possible implementations of this interface, and this is why we prefer diagnostics to pass down rich structured information in arguments. For example, an HTML output might want declaration names be linkified to where they come from in the source. Another example is that a GUI might let you click on typedefs to expand them. This application would want to pass significantly more information about types through to the GUI than a simple flat string. The interface allows this to happen.

Adding Translations to Clang

Not possible yet! Diagnostic strings should be written in UTF-8, the client can translate to the relevant code page if needed. Each translation completely replaces the format string for the diagnostic.

The `SourceLocation` and `SourceManager` classes

Strangely enough, the `SourceLocation` class represents a location within the source code of the program. Important design points include:

1. `sizeof(SourceLocation)` must be extremely small, as these are embedded into many AST nodes and are passed around often. Currently it is 32 bits.
2. `SourceLocation` must be a simple value object that can be efficiently copied.
3. We should be able to represent a source location for any byte of any input file. This includes in the middle of tokens, in whitespace, in trigraphs, etc.
4. A `SourceLocation` must encode the current `#include` stack that was active when the location was processed. For example, if the location corresponds to a token, it should contain the set of `#includes` active when the token was lexed. This allows us to print the `#include` stack for a diagnostic.
5. `SourceLocation` must be able to describe macro expansions, capturing both the ultimate instantiation point and the source of the original character data.

In practice, the `SourceLocation` works together with the `SourceManager` class to encode two pieces of information about a location: its spelling location and its instantiation location. For most tokens, these will be the same. However, for a macro expansion (or tokens that came from a `_Pragma` directive) these will describe the location of the characters corresponding to the token and the location where the token was used (i.e., the macro instantiation point or the location of the `_Pragma` itself).

The Clang front-end inherently depends on the location of a token being tracked correctly. If it is ever incorrect, the front-end may get confused and die. The reason for this is that the notion of the “spelling” of a `Token` in Clang depends on being able to find the original input characters for the token. This concept maps directly to the “spelling location” for the token.

`SourceRange` and `CharSourceRange`

Clang represents most source ranges by `[first, last]`, where “first” and “last” each point to the beginning of their respective tokens. For example consider the `SourceRange` of the following statement:

```
x = foo + bar;
^first    ^last
```

To map from this representation to a character-based representation, the “last” location needs to be adjusted to point to (or past) the end of that token with either `Lexer::MeasureTokenLength()` or `Lexer::getLocForEndOfToken()`. For the rare cases where character-level source ranges information is needed we use the `CharSourceRange` class.

The Driver Library

The clang Driver and library are documented [here](#).

Precompiled Headers

Clang supports two implementations of precompiled headers. The default implementation, precompiled headers (*PCH*) uses a serialized representation of Clang’s internal data structures, encoded with the [LLVM bitstream format](#). Pretokenized headers (*PTH*), on the other hand, contain a serialized representation of the tokens encountered when preprocessing a header (and anything that header includes).

The Frontend Library

The Frontend library contains functionality useful for building tools on top of the Clang libraries, for example several methods for outputting diagnostics.

The Lexer and Preprocessor Library

The Lexer library contains several tightly-connected classes that are involved with the nasty process of lexing and preprocessing C source code. The main interface to this library for outside clients is the large `Preprocessor` class. It contains the various pieces of state that are required to coherently read tokens out of a translation unit.

The core interface to the `Preprocessor` object (once it is set up) is the `Preprocessor::Lex` method, which returns the next *Token* from the preprocessor stream. There are two types of token providers that the preprocessor is capable of reading from: a buffer lexer (provided by the *Lexer* class) and a buffered token stream (provided by the *TokenLexer* class).

The Token class

The `Token` class is used to represent a single lexed token. Tokens are intended to be used by the lexer/preprocess and parser libraries, but are not intended to live beyond them (for example, they should not live in the ASTs).

Tokens most often live on the stack (or some other location that is efficient to access) as the parser is running, but occasionally do get buffered up. For example, macro definitions are stored as a series of tokens, and the C++ front-end periodically needs to buffer tokens up for tentative parsing and various pieces of look-ahead. As such, the size of a `Token` matters. On a 32-bit system, `sizeof(Token)` is currently 16 bytes.

Tokens occur in two forms: *annotation tokens* and normal tokens. Normal tokens are those returned by the lexer, annotation tokens represent semantic information and are produced by the parser, replacing normal tokens in the token stream. Normal tokens contain the following information:

- **A `SourceLocation`** — This indicates the location of the start of the token.
- **A `length`** — This stores the length of the token as stored in the `SourceBuffer`. For tokens that include them, this length includes trigraphs and escaped newlines which are ignored by later phases of the compiler. By pointing into the original source buffer, it is always possible to get the original spelling of a token completely accurately.
- **`IdentifierInfo`** — If a token takes the form of an identifier, and if identifier lookup was enabled when the token was lexed (e.g., the lexer was not reading in “raw” mode) this contains a pointer to the unique hash value for the identifier. Because the lookup happens before keyword identification, this field is set even for language keywords like `for`.
- **`TokenKind`** — This indicates the kind of token as classified by the lexer. This includes things like `tok::starequal` (for the “`*=`” operator), `tok::ampamp` for the “`&&`” token, and keyword values (e.g., `tok::kw_for`) for identifiers that correspond to keywords. Note that some tokens can be spelled multiple ways. For example, C++ supports “operator keywords”, where things like “`and`” are treated exactly like the “`&&`” operator. In these cases, the kind value is set to `tok::ampamp`, which is good for the parser, which

doesn't have to consider both forms. For something that cares about which form is used (e.g., the preprocessor "stringize" operator) the spelling indicates the original form.

- **Flags** — There are currently four flags tracked by the lexer/preprocessor system on a per-token basis:
 1. **StartOfLine** — This was the first token that occurred on its input source line.
 2. **LeadingSpace** — There was a space character either immediately before the token or transitively before the token as it was expanded through a macro. The definition of this flag is very closely defined by the stringizing requirements of the preprocessor.
 3. **DisableExpand** — This flag is used internally to the preprocessor to represent identifier tokens which have macro expansion disabled. This prevents them from being considered as candidates for macro expansion ever in the future.
 4. **NeedsCleaning** — This flag is set if the original spelling for the token includes a trigraph or escaped newline. Since this is uncommon, many pieces of code can fast-path on tokens that did not need cleaning.

One interesting (and somewhat unusual) aspect of normal tokens is that they don't contain any semantic information about the lexed value. For example, if the token was a pp-number token, we do not represent the value of the number that was lexed (this is left for later pieces of code to decide). Additionally, the lexer library has no notion of typedef names vs variable names: both are returned as identifiers, and the parser is left to decide whether a specific identifier is a typedef or a variable (tracking this requires scope information among other things). The parser can do this translation by replacing tokens returned by the preprocessor with "Annotation Tokens".

Annotation Tokens

Annotation tokens are tokens that are synthesized by the parser and injected into the preprocessor's token stream (replacing existing tokens) to record semantic information found by the parser. For example, if "foo" is found to be a typedef, the "foo" tok::identifier token is replaced with an tok::annot_typename. This is useful for a couple of reasons: 1) this makes it easy to handle qualified type names (e.g., "foo::bar::baz<42>::t") in C++ as a single "token" in the parser. 2) if the parser backtracks, the reparse does not need to redo semantic analysis to determine whether a token sequence is a variable, type, template, etc.

Annotation tokens are created by the parser and reinjected into the parser's token stream (when backtracking is enabled). Because they can only exist in tokens that the preprocessor-proper is done with, it doesn't need to keep around flags like "start of line" that the preprocessor uses to do its job. Additionally, an annotation token may "cover" a sequence of preprocessor tokens (e.g., "a::b::c" is five preprocessor tokens). As such, the valid fields of an annotation token are different than the fields for a normal token (but they are multiplexed into the normal Token fields):

- **SourceLocation "Location"** — The SourceLocation for the annotation token indicates the first token replaced by the annotation token. In the example above, it would be the location of the "a" identifier.
- **SourceLocation "AnnotationEndLoc"** — This holds the location of the last token replaced with the annotation token. In the example above, it would be the location of the "c" identifier.
- **void* "AnnotationValue"** — This contains an opaque object that the parser gets from Sema. The parser merely preserves the information for Sema to later interpret based on the annotation token kind.
- **TokenKind "Kind"** — This indicates the kind of Annotation token this is. See below for the different valid kinds.

Annotation tokens currently come in three kinds:

1. **tok::annot_typename**: This annotation token represents a resolved typename token that is potentially qualified. The AnnotationValue field contains the QualType returned by Sema::getTypeName(), possibly with source location information attached.
2. **tok::annot_cxxscope**: This annotation token represents a C++ scope specifier, such as "A::B::". This corresponds to the grammar productions "::" and "::: [opt] nested-name-specifier". The AnnotationValue

pointer is a `NestedNameSpecifier *` returned by the `Sema::ActOnCXXGlobalScopeSpecifier` and `Sema::ActOnCXXNestedNameSpecifier` callbacks.

3. **`tok::annot_template_id`**: This annotation token represents a C++ template-id such as “`foo<int, 4>`”, where “`foo`” is the name of a template. The `AnnotationValue` pointer is a pointer to a `malloc'd` `TemplateIdAnnotation` object. Depending on the context, a parsed template-id that names a type might become a typename annotation token (if all we care about is the named type, e.g., because it occurs in a type specifier) or might remain a template-id token (if we want to retain more source location information or produce a new type, e.g., in a declaration of a class template specialization). template-id annotation tokens that refer to a type can be “upgraded” to typename annotation tokens by the parser.

As mentioned above, annotation tokens are not returned by the preprocessor, they are formed on demand by the parser. This means that the parser has to be aware of cases where an annotation could occur and form it where appropriate. This is somewhat similar to how the parser handles Translation Phase 6 of C99: String Concatenation (see C99 5.1.1.2). In the case of string concatenation, the preprocessor just returns distinct `tok::string_literal` and `tok::wide_string_literal` tokens and the parser eats a sequence of them wherever the grammar indicates that a string literal can occur.

In order to do this, whenever the parser expects a `tok::identifier` or `tok::coloncolon`, it should call the `TryAnnotateTypeOrScopeToken` or `TryAnnotateCXXScopeToken` methods to form the annotation token. These methods will maximally form the specified annotation tokens and replace the current token with them, if applicable. If the current tokens is not valid for an annotation token, it will remain an identifier or “`:`” token.

The `Lexer` class

The `Lexer` class provides the mechanics of lexing tokens out of a source buffer and deciding what they mean. The `Lexer` is complicated by the fact that it operates on raw buffers that have not had spelling eliminated (this is a necessity to get decent performance), but this is countered with careful coding as well as standard performance techniques (for example, the comment handling code is vectorized on X86 and PowerPC hosts).

The lexer has a couple of interesting modal features:

- The lexer can operate in “raw” mode. This mode has several features that make it possible to quickly lex the file (e.g., it stops identifier lookup, doesn’t specially handle preprocessor tokens, handles EOF differently, etc). This mode is used for lexing within an “`#if 0`” block, for example.
- The lexer can capture and return comments as tokens. This is required to support the `-C` preprocessor mode, which passes comments through, and is used by the diagnostic checker to identifier expect-error annotations.
- The lexer can be in `ParsingFilename` mode, which happens when preprocessing after reading a `#include` directive. This mode changes the parsing of “`<`” to return an “angled string” instead of a bunch of tokens for each thing within the filename.
- When parsing a preprocessor directive (after “`#`”) the `ParsingPreprocessorDirective` mode is entered. This changes the parser to return EOD at a newline.
- The `Lexer` uses a `LangOptions` object to know whether trigraphs are enabled, whether C++ or ObjC keywords are recognized, etc.

In addition to these modes, the lexer keeps track of a couple of other features that are local to a lexed buffer, which change as the buffer is lexed:

- The `Lexer` uses `BufferPtr` to keep track of the current character being lexed.
- The `Lexer` uses `IsAtStartOfLine` to keep track of whether the next lexed token will start with its “start of line” bit set.
- The `Lexer` keeps track of the current “`#if`” directives that are active (which can be nested).

- The `Lexer` keeps track of an *MultipleIncludeOpt* object, which is used to detect whether the buffer uses the standard “`#ifndef XX / #define XX`” idiom to prevent multiple inclusion. If a buffer does, subsequent includes can be ignored if the “XX” macro is defined.

The `TokenLexer` class

The `TokenLexer` class is a token provider that returns tokens from a list of tokens that came from somewhere else. It typically used for two things: 1) returning tokens from a macro definition as it is being expanded 2) returning tokens from an arbitrary buffer of tokens. The later use is used by `_Pragma` and will most likely be used to handle unbounded look-ahead for the C++ parser.

The `MultipleIncludeOpt` class

The `MultipleIncludeOpt` class implements a really simple little state machine that is used to detect the standard “`#ifndef XX / #define XX`” idiom that people typically use to prevent multiple inclusion of headers. If a buffer uses this idiom and is subsequently `#include`d, the preprocessor can simply check to see whether the guarding condition is defined or not. If so, the preprocessor can completely ignore the include of the header.

The Parser Library

This library contains a recursive-descent parser that polls tokens from the preprocessor and notifies a client of the parsing progress.

Historically, the parser used to talk to an abstract `Action` interface that had virtual methods for parse events, for example `ActOnBinOp()`. When Clang grew C++ support, the parser stopped supporting general `Action` clients – it now always talks to the *Sema library*. However, the Parser still accesses AST objects only through opaque types like `ExprResult` and `StmtResult`. Only *Sema* looks at the AST node contents of these wrappers.

The AST Library

The `Type` class and its subclasses

The `Type` class (and its subclasses) are an important part of the AST. Types are accessed through the `ASTContext` class, which implicitly creates and uniques them as they are needed. Types have a couple of non-obvious features: 1) they do not capture type qualifiers like `const` or `volatile` (see *QualType*), and 2) they implicitly capture typedef information. Once created, types are immutable (unlike decls).

Typedefs in C make semantic analysis a bit more complex than it would be without them. The issue is that we want to capture typedef information and represent it in the AST perfectly, but the semantics of operations need to “see through” typedefs. For example, consider this code:

```
void func() {
    typedef int foo;
    foo X, *Y;
    typedef foo *bar;
    bar Z;
    *X; // error
    **Y; // error
    **Z; // error
}
```

The code above is illegal, and thus we expect there to be diagnostics emitted on the annotated lines. In this example, we expect to get:

```
test.c:6:1: error: indirection requires pointer operand ('foo' invalid)
  *X; // error
  ^~
test.c:7:1: error: indirection requires pointer operand ('foo' invalid)
  **Y; // error
  ^~~
test.c:8:1: error: indirection requires pointer operand ('foo' invalid)
  **Z; // error
  ^~~
```

While this example is somewhat silly, it illustrates the point: we want to retain typedef information where possible, so that we can emit errors about “std::string” instead of “std::basic_string<char, std::...”. Doing this requires properly keeping typedef information (for example, the type of `X` is “foo”, not “int”), and requires properly propagating it through the various operators (for example, the type of `*Y` is “foo”, not “int”). In order to retain this information, the type of these expressions is an instance of the `TypedefType` class, which indicates that the type of these expressions is a typedef for “foo”.

Representing types like this is great for diagnostics, because the user-specified type is always immediately available. There are two problems with this: first, various semantic checks need to make judgements about the *actual structure* of a type, ignoring typedefs. Second, we need an efficient way to query whether two types are structurally identical to each other, ignoring typedefs. The solution to both of these problems is the idea of canonical types.

Canonical Types

Every instance of the `Type` class contains a canonical type pointer. For simple types with no typedefs involved (e.g., “int”, “int*”, “int**”), the type just points to itself. For types that have a typedef somewhere in their structure (e.g., “foo”, “foo*”, “foo**”, “bar”), the canonical type pointer points to their structurally equivalent type without any typedefs (e.g., “int”, “int*”, “int**”, and “int*” respectively).

This design provides a constant time operation (dereferencing the canonical type pointer) that gives us access to the structure of types. For example, we can trivially tell that “bar” and “foo*” are the same type by dereferencing their canonical type pointers and doing a pointer comparison (they both point to the single “int*” type).

Canonical types and typedef types bring up some complexities that must be carefully managed. Specifically, the `isa/cast/dyn_cast` operators generally shouldn’t be used in code that is inspecting the AST. For example, when type checking the indirection operator (unary “*” on a pointer), the type checker must verify that the operand has a pointer type. It would not be correct to check that with “`isa<PointerType>(SubExpr->getType())`”, because this predicate would fail if the subexpression had a typedef type.

The solution to this problem are a set of helper methods on `Type`, used to check their properties. In this case, it would be correct to use “`SubExpr->getType()->isPointerType()`” to do the check. This predicate will return true if the *canonical type is a pointer*, which is true any time the type is structurally a pointer type. The only hard part here is remembering not to use the `isa/cast/dyn_cast` operations.

The second problem we face is how to get access to the pointer type once we know it exists. To continue the example, the result type of the indirection operator is the pointee type of the subexpression. In order to determine the type, we need to get the instance of `PointerType` that best captures the typedef information in the program. If the type of the expression is literally a `PointerType`, we can return that, otherwise we have to dig through the typedefs to find the pointer type. For example, if the subexpression had type “foo*”, we could return that type as the result. If the subexpression had type “bar”, we want to return “foo*” (note that we do *not* want “int*”). In order to provide all of this, `Type` has a `getAsPointerType()` method that checks whether the type is structurally a `PointerType` and, if so, returns the best one. If not, it returns a null pointer.

This structure is somewhat mystical, but after meditating on it, it will make sense to you :).

The QualType class

The `QualType` class is designed as a trivial value class that is small, passed by-value and is efficient to query. The idea of `QualType` is that it stores the type qualifiers (`const`, `volatile`, `restrict`, plus some extended qualifiers required by language extensions) separately from the types themselves. `QualType` is conceptually a pair of “`Type*`” and the bits for these type qualifiers.

By storing the type qualifiers as bits in the conceptual pair, it is extremely efficient to get the set of qualifiers on a `QualType` (just return the field of the pair), add a type qualifier (which is a trivial constant-time operation that sets a bit), and remove one or more type qualifiers (just return a `QualType` with the bitfield set to empty).

Further, because the bits are stored outside of the type itself, we do not need to create duplicates of types with different sets of qualifiers (i.e. there is only a single heap allocated “`int`” type: “`const int`” and “`volatile const int`” both point to the same heap allocated “`int`” type). This reduces the heap size used to represent bits and also means we do not have to consider qualifiers when uniquing types (*Type* does not even contain qualifiers).

In practice, the two most common type qualifiers (`const` and `restrict`) are stored in the low bits of the pointer to the `Type` object, together with a flag indicating whether extended qualifiers are present (which must be heap-allocated). This means that `QualType` is exactly the same size as a pointer.

Declaration names

The `DeclarationName` class represents the name of a declaration in Clang. Declarations in the C family of languages can take several different forms. Most declarations are named by simple identifiers, e.g., “`f`” and “`x`” in the function declaration `f(int x)`. In C++, declaration names can also name class constructors (“`Class`” in `struct Class { Class(); }`), class destructors (“`~Class`”), overloaded operator names (“`operator+`”), and conversion functions (“`operator void const *`”). In Objective-C, declaration names can refer to the names of Objective-C methods, which involve the method name and the parameters, collectively called a *selector*, e.g., “`setWidth:height:`”. Since all of these kinds of entities — variables, functions, Objective-C methods, C++ constructors, destructors, and operators — are represented as subclasses of Clang’s common `NamedDecl` class, `DeclarationName` is designed to efficiently represent any kind of name.

Given a `DeclarationName` `N`, `N.getNameKind()` will produce a value that describes what kind of name `N` stores. There are 10 options (all of the names are inside the `DeclarationName` class).

`Identifier`

The name is a simple identifier. Use `N.getAsIdentifierInfo()` to retrieve the corresponding `IdentifierInfo*` pointing to the actual identifier.

`ObjCZeroArgSelector`, `ObjCOneArgSelector`, `ObjCMultiArgSelector`

The name is an Objective-C selector, which can be retrieved as a `Selector` instance via `N.getObjCSelector()`. The three possible name kinds for Objective-C reflect an optimization within the `DeclarationName` class: both zero- and one-argument selectors are stored as a masked `IdentifierInfo` pointer, and therefore require very little space, since zero- and one-argument selectors are far more common than multi-argument selectors (which use a different structure).

`CXXConstructorName`

The name is a C++ constructor name. Use `N.getCXXNameType()` to retrieve the *type* that this constructor is meant to construct. The type is always the canonical type, since all constructors for a given type have the same name.

`CXXDestructorName`

The name is a C++ destructor name. Use `N.getCXXNameType()` to retrieve the *type* whose destructor is being named. This type is always a canonical type.

`CXXConversionFunctionName`

The name is a C++ conversion function. Conversion functions are named according to the type they convert to, e.g., “operator void const *”. Use `N.getCXXNameType()` to retrieve the type that this conversion function converts to. This type is always a canonical type.

CXXOperatorName

The name is a C++ overloaded operator name. Overloaded operators are named according to their spelling, e.g., “operator+” or “operator new []”. Use `N.getCXXOverloadedOperator()` to retrieve the overloaded operator (a value of type `OverloadedOperatorKind`).

CXXLiteralOperatorName

The name is a C++11 user defined literal operator. User defined Literal operators are named according to the suffix they define, e.g., “_foo” for “operator "" _foo”. Use `N.getCXXLiteralIdentifier()` to retrieve the corresponding `IdentifierInfo*` pointing to the identifier.

CXXUsingDirective

The name is a C++ using directive. Using directives are not really `NamedDecls`, in that they all have the same name, but they are implemented as such in order to store them in `DeclContext` effectively.

`DeclarationNames` are cheap to create, copy, and compare. They require only a single pointer’s worth of storage in the common cases (identifiers, zero- and one-argument Objective-C selectors) and use dense, uniqued storage for the other kinds of names. Two `DeclarationNames` can be compared for equality (`==`, `!=`) using a simple bitwise comparison, can be ordered with `<`, `>`, `<=`, and `>=` (which provide a lexicographical ordering for normal identifiers but an unspecified ordering for other kinds of names), and can be placed into LLVM `DenseMaps` and `DenseSets`.

`DeclarationName` instances can be created in different ways depending on what kind of name the instance will store. Normal identifiers (`IdentifierInfo` pointers) and Objective-C selectors (`Selector`) can be implicitly converted to `DeclarationNames`. Names for C++ constructors, destructors, conversion functions, and overloaded operators can be retrieved from the `DeclarationNameTable`, an instance of which is available as `ASTContext::DeclarationNames`. The member functions `getCXXConstructorName`, `getCXXDestructorName`, `getCXXConversionFunctionName`, and `getCXXOperatorName`, respectively, return `DeclarationName` instances for the four kinds of C++ special function names.

Declaration contexts

Every declaration in a program exists within some *declaration context*, such as a translation unit, namespace, class, or function. Declaration contexts in Clang are represented by the `DeclContext` class, from which the various declaration-context AST nodes (`TranslationUnitDecl`, `NamespaceDecl`, `RecordDecl`, `FunctionDecl`, etc.) will derive. The `DeclContext` class provides several facilities common to each declaration context:

Source-centric vs. Semantics-centric View of Declarations

`DeclContext` provides two views of the declarations stored within a declaration context. The source-centric view accurately represents the program source code as written, including multiple declarations of entities where present (see the section [Redeclarations and Overloads](#)), while the semantics-centric view represents the program semantics. The two views are kept synchronized by semantic analysis while the ASTs are being constructed.

Storage of declarations within that context

Every declaration context can contain some number of declarations. For example, a C++ class (represented by `RecordDecl`) contains various member functions, fields, nested types, and so on. All of these declarations will be stored within the `DeclContext`, and one can iterate over the declarations via `[DeclContext::decls_begin(), DeclContext::decls_end())`. This mechanism provides the source-centric view of declarations in the context.

Lookup of declarations within that context

The `DeclContext` structure provides efficient name lookup for names within that declaration context. For example, if `N` is a namespace we can look for the name `N::f` using `DeclContext::lookup`. The lookup itself is based on a lazily-constructed array (for declaration contexts with a small number of declarations) or hash table (for declaration contexts with more declarations). The lookup operation provides the semantics-centric view of the declarations in the context.

Ownership of declarations

The `DeclContext` owns all of the declarations that were declared within its declaration context, and is responsible for the management of their memory as well as their (de-)serialization.

All declarations are stored within a declaration context, and one can query information about the context in which each declaration lives. One can retrieve the `DeclContext` that contains a particular `Decl` using `Decl::getDeclContext`. However, see the section [Lexical and Semantic Contexts](#) for more information about how to interpret this context information.

Redeclarations and Overloads

Within a translation unit, it is common for an entity to be declared several times. For example, we might declare a function “`f`” and then later re-declare it as part of an inlined definition:

```
void f(int x, int y, int z = 1);

inline void f(int x, int y, int z) { /* ... */ }
```

The representation of “`f`” differs in the source-centric and semantics-centric views of a declaration context. In the source-centric view, all redeclarations will be present, in the order they occurred in the source code, making this view suitable for clients that wish to see the structure of the source code. In the semantics-centric view, only the most recent “`f`” will be found by the lookup, since it effectively replaces the first declaration of “`f`”.

In the semantics-centric view, overloading of functions is represented explicitly. For example, given two declarations of a function “`g`” that are overloaded, e.g.,

```
void g();
void g(int);
```

the `DeclContext::lookup` operation will return a `DeclContext::lookup_result` that contains a range of iterators over declarations of “`g`”. Clients that perform semantic analysis on a program that is not concerned with the actual source code will primarily use this semantics-centric view.

Lexical and Semantic Contexts

Each declaration has two potentially different declaration contexts: a *lexical* context, which corresponds to the source-centric view of the declaration context, and a *semantic* context, which corresponds to the semantics-centric view. The lexical context is accessible via `Decl::getLexicalDeclContext` while the semantic context is accessible via `Decl::getDeclContext`, both of which return `DeclContext` pointers. For most declarations, the two contexts are identical. For example:

```
class X {
public:
    void f(int x);
};
```

Here, the semantic and lexical contexts of `X::f` are the `DeclContext` associated with the class `X` (itself stored as a `RecordDecl` AST node). However, we can now define `X::f` out-of-line:

```
void X::f(int x = 17) { /* ... */ }
```

This definition of “f” has different lexical and semantic contexts. The lexical context corresponds to the declaration context in which the actual declaration occurred in the source code, e.g., the translation unit containing X. Thus, this declaration of `X::f` can be found by traversing the declarations provided by `[decls_begin(), decls_end())` in the translation unit.

The semantic context of `X::f` corresponds to the class X, since this member function is (semantically) a member of X. Lookup of the name `f` into the `DeclContext` associated with X will then return the definition of `X::f` (including information about the default argument).

Transparent Declaration Contexts

In C and C++, there are several contexts in which names that are logically declared inside another declaration will actually “leak” out into the enclosing scope from the perspective of name lookup. The most obvious instance of this behavior is in enumeration types, e.g.,

```
enum Color {  
    Red,  
    Green,  
    Blue  
};
```

Here, `Color` is an enumeration, which is a declaration context that contains the enumerators `Red`, `Green`, and `Blue`. Thus, traversing the list of declarations contained in the enumeration `Color` will yield `Red`, `Green`, and `Blue`. However, outside of the scope of `Color` one can name the enumerator `Red` without qualifying the name, e.g.,

```
Color c = Red;
```

There are other entities in C++ that provide similar behavior. For example, linkage specifications that use curly braces:

```
extern "C" {  
    void f(int);  
    void g(int);  
}  
// f and g are visible here
```

For source-level accuracy, we treat the linkage specification and enumeration type as a declaration context in which its enclosed declarations (“Red”, “Green”, and “Blue”; “f” and “g”) are declared. However, these declarations are visible outside of the scope of the declaration context.

These language features (and several others, described below) have roughly the same set of requirements: declarations are declared within a particular lexical context, but the declarations are also found via name lookup in scopes enclosing the declaration itself. This feature is implemented via *transparent* declaration contexts (see `DeclContext::isTransparentContext()`), whose declarations are visible in the nearest enclosing non-transparent declaration context. This means that the lexical context of the declaration (e.g., an enumerator) will be the transparent `DeclContext` itself, as will the semantic context, but the declaration will be visible in every outer context up to and including the first non-transparent declaration context (since transparent declaration contexts can be nested).

The transparent `DeclContext`s are:

- Enumerations (but not C++11 “scoped enumerations”):

```
enum Color {  
    Red,
```

```

    Green,
    Blue
};
// Red, Green, and Blue are in scope

```

- C++ linkage specifications:

```

extern "C" {
    void f(int);
    void g(int);
}
// f and g are in scope

```

- Anonymous unions and structs:

```

struct LookupTable {
    bool IsVector;
    union {
        std::vector<Item> *Vector;
        std::set<Item> *Set;
    };
};

LookupTable LT;
LT.Vector = 0; // Okay: finds Vector inside the unnamed union

```

- C++11 inline namespaces:

```

namespace mylib {
    inline namespace debug {
        class X;
    }
}
mylib::X *xp; // okay: mylib::X refers to mylib::debug::X

```

Multiply-Defined Declaration Contexts

C++ namespaces have the interesting — and, so far, unique — property that the namespace can be defined multiple times, and the declarations provided by each namespace definition are effectively merged (from the semantic point of view). For example, the following two code snippets are semantically indistinguishable:

```

// Snippet #1:
namespace N {
    void f();
}
namespace N {
    void f(int);
}

// Snippet #2:
namespace N {
    void f();
    void f(int);
}

```

In Clang’s representation, the source-centric view of declaration contexts will actually have two separate `NamespaceDecl` nodes in Snippet #1, each of which is a declaration context that contains a single declaration of “f”. However, the semantics-centric view provided by name lookup into the namespace N for “f” will return a `DeclContext::lookup_result` that contains a range of iterators over declarations of “f”.

`DeclContext` manages multiply-defined declaration contexts internally. The function `DeclContext::getPrimaryContext` retrieves the “primary” context for a given `DeclContext` instance, which is the `DeclContext` responsible for maintaining the lookup table used for the semantics-centric view. Given a `DeclContext`, one can obtain the set of declaration contexts that are semantically connected to this declaration context, in source order, including this context (which will be the only result, for non-namespace contexts) via `DeclContext::collectAllContexts`. Note that these functions are used internally within the lookup and insertion methods of the `DeclContext`, so the vast majority of clients can ignore them.

The CFG class

The CFG class is designed to represent a source-level control-flow graph for a single statement (`Stmt*`). Typically instances of CFG are constructed for function bodies (usually an instance of `CompoundStmt`), but can also be instantiated to represent the control-flow of any class that subclasses `Stmt`, which includes simple expressions. Control-flow graphs are especially useful for performing [flow- or path-sensitive](#) program analyses on a given function.

Basic Blocks

Concretely, an instance of CFG is a collection of basic blocks. Each basic block is an instance of `CFGBlock`, which simply contains an ordered sequence of `Stmt*` (each referring to statements in the AST). The ordering of statements within a block indicates unconditional flow of control from one statement to the next. [Conditional control-flow](#) is represented using edges between basic blocks. The statements within a given `CFGBlock` can be traversed using the `CFGBlock::iterator` interface.

A CFG object owns the instances of `CFGBlock` within the control-flow graph it represents. Each `CFGBlock` within a CFG is also uniquely numbered (accessible via `CFGBlock::getBlockID()`). Currently the number is based on the ordering the blocks were created, but no assumptions should be made on how `CFGBlocks` are numbered other than their numbers are unique and that they are numbered from 0..N-1 (where N is the number of basic blocks in the CFG).

Entry and Exit Blocks

Each instance of CFG contains two special blocks: an *entry* block (accessible via `CFG::getEntry()`), which has no incoming edges, and an *exit* block (accessible via `CFG::getExit()`), which has no outgoing edges. Neither block contains any statements, and they serve the role of providing a clear entrance and exit for a body of code such as a function body. The presence of these empty blocks greatly simplifies the implementation of many analyses built on top of CFGs.

Conditional Control-Flow

Conditional control-flow (such as those induced by if-statements and loops) is represented as edges between `CFGBlocks`. Because different C language constructs can induce control-flow, each `CFGBlock` also records an extra `Stmt*` that represents the *terminator* of the block. A terminator is simply the statement that caused the control-flow, and is used to identify the nature of the conditional control-flow between blocks. For example, in the case of an if-statement, the terminator refers to the `IfStmt` object in the AST that represented the given branch.

To illustrate, consider the following code example:

```
int foo(int x) {
    x = x + 1;
    if (x > 2)
        x++;
    else {
        x += 2;
        x *= 2;
    }

    return x;
}
```

After invoking the parser+semantic analyzer on this code fragment, the AST of the body of `foo` is referenced by a single `Stmt*`. We can then construct an instance of `CFG` representing the control-flow graph of this function body by single call to a static class method:

```
Stmt *FooBody = ...
std::unique_ptr<CFG> FooCFG = CFG::buildCFG(FooBody);
```

Along with providing an interface to iterate over its `CFGBlocks`, the `CFG` class also provides methods that are useful for debugging and visualizing CFGs. For example, the method `CFG::dump()` dumps a pretty-printed version of the CFG to standard error. This is especially useful when one is using a debugger such as `gdb`. For example, here is the output of `FooCFG->dump()`:

```
[ B5 (ENTRY) ]
  Predecessors (0):
  Successors (1): B4

[ B4 ]
  1: x = x + 1
  2: (x > 2)
  T: if [B4.2]
  Predecessors (1): B5
  Successors (2): B3 B2

[ B3 ]
  1: x++
  Predecessors (1): B4
  Successors (1): B1

[ B2 ]
  1: x += 2
  2: x *= 2
  Predecessors (1): B4
  Successors (1): B1

[ B1 ]
  1: return x;
  Predecessors (2): B2 B3
  Successors (1): B0

[ B0 (EXIT) ]
  Predecessors (1): B1
  Successors (0):
```

For each block, the pretty-printed output displays for each block the number of *predecessor* blocks (blocks that have outgoing control-flow to the given block) and *successor* blocks (blocks that have control-flow that have incoming

control-flow from the given block). We can also clearly see the special entry and exit blocks at the beginning and end of the pretty-printed output. For the entry block (block B5), the number of predecessor blocks is 0, while for the exit block (block B0) the number of successor blocks is 0.

The most interesting block here is B4, whose outgoing control-flow represents the branching caused by the sole if-statement in `foo`. Of particular interest is the second statement in the block, `(x > 2)`, and the terminator, printed as `if [B4.2]`. The second statement represents the evaluation of the condition of the if-statement, which occurs before the actual branching of control-flow. Within the `CFGBlock` for B4, the `Stmt*` for the second statement refers to the actual expression in the AST for `(x > 2)`. Thus pointers to subclasses of `Expr` can appear in the list of statements in a block, and not just subclasses of `Stmt` that refer to proper C statements.

The terminator of block B4 is a pointer to the `IfStmt` object in the AST. The pretty-printer outputs `if [B4.2]` because the condition expression of the if-statement has an actual place in the basic block, and thus the terminator is essentially *referring* to the expression that is the second statement of block B4 (i.e., B4.2). In this manner, conditions for control-flow (which also includes conditions for loops and switch statements) are hoisted into the actual basic block.

Constant Folding in the Clang AST

There are several places where constants and constant folding matter a lot to the Clang front-end. First, in general, we prefer the AST to retain the source code as close to how the user wrote it as possible. This means that if they wrote “5+4”, we want to keep the addition and two constants in the AST, we don’t want to fold to “9”. This means that constant folding in various ways turns into a tree walk that needs to handle the various cases.

However, there are places in both C and C++ that require constants to be folded. For example, the C standard defines what an “integer constant expression” (i-c-e) is with very precise and specific requirements. The language then requires i-c-e’s in a lot of places (for example, the size of a bitfield, the value for a case statement, etc). For these, we have to be able to constant fold the constants, to do semantic checks (e.g., verify bitfield size is non-negative and that case statements aren’t duplicated). We aim for Clang to be very pedantic about this, diagnosing cases when the code does not use an i-c-e where one is required, but accepting the code unless running with `-pedantic-errors`.

Things get a little bit more tricky when it comes to compatibility with real-world source code. Specifically, GCC has historically accepted a huge superset of expressions as i-c-e’s, and a lot of real world code depends on this unfortunate accident of history (including, e.g., the glibc system headers). GCC accepts anything its “fold” optimizer is capable of reducing to an integer constant, which means that the definition of what it accepts changes as its optimizer does. One example is that GCC accepts things like “`case X-X:`” even when `X` is a variable, because it can fold this to 0.

Another issue are how constants interact with the extensions we support, such as `__builtin_constant_p`, `__builtin_inf`, `__extension__` and many others. C99 obviously does not specify the semantics of any of these extensions, and the definition of i-c-e does not include them. However, these extensions are often used in real code, and we have to have a way to reason about them.

Finally, this is not just a problem for semantic analysis. The code generator and other clients have to be able to fold constants (e.g., to initialize global variables) and has to handle a superset of what C99 allows. Further, these clients can benefit from extended information. For example, we know that “`foo() || 1`” always evaluates to `true`, but we can’t replace the expression with `true` because it has side effects.

Implementation Approach

After trying several different approaches, we’ve finally converged on a design (Note, at the time of this writing, not all of this has been implemented, consider this a design goal!). Our basic approach is to define a single recursive method evaluation method (`Expr::Evaluate`), which is implemented in `AST/ExprConstant.cpp`. Given an expression with “scalar” type (integer, fp, complex, or pointer) this method returns the following information:

- Whether the expression is an integer constant expression, a general constant that was folded but has no side effects, a general constant that was folded but that does have side effects, or an uncomputable/unfoldable value.

- If the expression was computable in any way, this method returns the `APValue` for the result of the expression.
- If the expression is not evaluable at all, this method returns information on one of the problems with the expression. This includes a `SourceLocation` for where the problem is, and a diagnostic ID that explains the problem. The diagnostic should have `ERROR` type.
- If the expression is not an integer constant expression, this method returns information on one of the problems with the expression. This includes a `SourceLocation` for where the problem is, and a diagnostic ID that explains the problem. The diagnostic should have `EXTENSION` type.

This information gives various clients the flexibility that they want, and we will eventually have some helper methods for various extensions. For example, `Sema` should have a `Sema::VerifyIntegerConstantExpression` method, which calls `Evaluate` on the expression. If the expression is not foldable, the error is emitted, and it would return `true`. If the expression is not an i-c-e, the `EXTENSION` diagnostic is emitted. Finally it would return `false` to indicate that the AST is OK.

Other clients can use the information in other ways, for example, `codegen` can just use expressions that are foldable in any way.

Extensions

This section describes how some of the various extensions Clang supports interacts with constant evaluation:

- `__extension__`: The expression form of this extension causes any evaluable subexpression to be accepted as an integer constant expression.
- `__builtin_constant_p`: This returns `true` (as an integer constant expression) if the operand evaluates to either a numeric value (that is, not a pointer cast to integral type) of integral, enumeration, floating or complex type, or if it evaluates to the address of the first character of a string literal (possibly cast to some other type). As a special case, if `__builtin_constant_p` is the (potentially parenthesized) condition of a conditional operator expression (“?:”), only the true side of the conditional operator is considered, and it is evaluated with full constant folding.
- `__builtin_choose_expr`: The condition is required to be an integer constant expression, but we accept any constant as an “extension of an extension”. This only evaluates one operand depending on which way the condition evaluates.
- `__builtin_classify_type`: This always returns an integer constant expression.
- `__builtin_inf`, `nan`, ...: These are treated just like a floating-point literal.
- `__builtin_abs`, `copysign`, ...: These are constant folded as general constant expressions.
- `__builtin_strlen` and `strlen`: These are constant folded as integer constant expressions if the argument is a string literal.

The Sema Library

This library is called by the *Parser library* during parsing to do semantic analysis of the input. For valid programs, `Sema` builds an AST for parsed constructs.

The CodeGen Library

`CodeGen` takes an *AST* as input and produces LLVM IR code from it.

How to change Clang

How to add an attribute

Attributes are a form of metadata that can be attached to a program construct, allowing the programmer to pass semantic information along to the compiler for various uses. For example, attributes may be used to alter the code generation for a program construct, or to provide extra semantic information for static analysis. This document explains how to add a custom attribute to Clang. Documentation on existing attributes can be found [here](#).

Attribute Basics

Attributes in Clang are handled in three stages: parsing into a parsed attribute representation, conversion from a parsed attribute into a semantic attribute, and then the semantic handling of the attribute.

Parsing of the attribute is determined by the various syntactic forms attributes can take, such as GNU, C++11, and Microsoft style attributes, as well as other information provided by the table definition of the attribute. Ultimately, the parsed representation of an attribute object is an `AttributeList` object. These parsed attributes chain together as a list of parsed attributes attached to a declarator or declaration specifier. The parsing of attributes is handled automatically by Clang, except for attributes spelled as keywords. When implementing a keyword attribute, the parsing of the keyword and creation of the `AttributeList` object must be done manually.

Eventually, `Sema::ProcessDeclAttributeList()` is called with a `Decl` and an `AttributeList`, at which point the parsed attribute can be transformed into a semantic attribute. The process by which a parsed attribute is converted into a semantic attribute depends on the attribute definition and semantic requirements of the attribute. The end result, however, is that the semantic attribute object is attached to the `Decl` object, and can be obtained by a call to `Decl::getAttr<T>()`.

The structure of the semantic attribute is also governed by the attribute definition given in `Attr.td`. This definition is used to automatically generate functionality used for the implementation of the attribute, such as a class derived from `clang::Attr`, information for the parser to use, automated semantic checking for some attributes, etc.

`include/clang/Basic/Attr.td`

The first step to adding a new attribute to Clang is to add its definition to `include/clang/Basic/Attr.td`. This tablegen definition must derive from the `Attr` (tablegen, not semantic) type, or one of its derivatives. Most attributes will derive from the `InheritableAttr` type, which specifies that the attribute can be inherited by later redeclarations of the `Decl` it is associated with. `InheritableParamAttr` is similar to `InheritableAttr`, except that the attribute is written on a parameter instead of a declaration. If the attribute is intended to apply to a type instead of a declaration, such an attribute should derive from `TypeAttr`, and will generally not be given an AST representation. (Note that this document does not cover the creation of type attributes.) An attribute that inherits from `IgnoredAttr` is parsed, but will generate an ignored attribute diagnostic when used, which may be useful when an attribute is supported by another vendor but not supported by clang.

The definition will specify several key pieces of information, such as the semantic name of the attribute, the spellings the attribute supports, the arguments the attribute expects, and more. Most members of the `Attr` tablegen type do not require definitions in the derived definition as the default suffice. However, every attribute must specify at least a spelling list, a subject list, and a documentation list.

Spellings

All attributes are required to specify a spelling list that denotes the ways in which the attribute can be spelled. For instance, a single semantic attribute may have a keyword spelling, as well as a C++11 spelling and a GNU spelling.

An empty spelling list is also permissible and may be useful for attributes which are created implicitly. The following spellings are accepted:

Spelling	Description
GNU	Spelled with a GNU-style <code>__attribute__((attr))</code> syntax and placement.
CXX11	Spelled with a C++-style <code>[[attr]]</code> syntax. If the attribute is meant to be used by Clang, it should set the namespace to "clang".
Declspec	Spelled with a Microsoft-style <code>__declspec(attr)</code> syntax.
Keyword	The attribute is spelled as a keyword, and required custom parsing.
GCC	Specifies two spellings: the first is a GNU-style spelling, and the second is a C++-style spelling with the <code>gnu</code> namespace. Attributes should only specify this spelling for attributes supported by GCC.
Pragma	The attribute is spelled as a <code>#pragma</code> , and requires custom processing within the preprocessor. If the attribute is meant to be used by Clang, it should set the namespace to "clang". Note that this spelling is not used for declaration attributes.

Subjects

Attributes appertain to one or more `Decl` subjects. If the attribute attempts to attach to a subject that is not in the subject list, a diagnostic is issued automatically. Whether the diagnostic is a warning or an error depends on how the attribute's `SubjectList` is defined, but the default behavior is to warn. The diagnostics displayed to the user are automatically determined based on the subjects in the list, but a custom diagnostic parameter can also be specified in the `SubjectList`. The diagnostics generated for subject list violations are either `diag::warn_attribute_wrong_decl_type` or `diag::err_attribute_wrong_decl_type`, and the parameter enumeration is found in `include/clang/Sema/AttributeList.h`. If a previously unused `Decl` node is added to the `SubjectList`, the logic used to automatically determine the diagnostic parameter in `utils/TableGen/ClangAttrEmitter.cpp` may need to be updated.

By default, all subjects in the `SubjectList` must either be a `Decl` node defined in `DeclNodes.td`, or a statement node defined in `StmtNodes.td`. However, more complex subjects can be created by creating a `SubsetSubject` object. Each such object has a base subject which it appertains to (which must be a `Decl` or `Stmt` node, and not a `SubsetSubject` node), and some custom code which is called when determining whether an attribute appertains to the subject. For instance, a `NonBitField` `SubsetSubject` appertains to a `FieldDecl`, and tests whether the given `FieldDecl` is a bit field. When a `SubsetSubject` is specified in a `SubjectList`, a custom diagnostic parameter must also be provided.

Diagnostic checking for attribute subject lists is automated except when `HasCustomParsing` is set to 1.

Documentation

All attributes must have some form of documentation associated with them. Documentation is table generated on the public web server by a server-side process that runs daily. Generally, the documentation for an attribute is a stand-alone definition in `include/clang/Basic/AttrDocs.td` that is named after the attribute being documented.

If the attribute is not for public consumption, or is an implicitly-created attribute that has no visible spelling, the documentation list can specify the `Undocumented` object. Otherwise, the attribute should have its documentation added to `AttrDocs.td`.

Documentation derives from the `Documentation` tablegen type. All derived types must specify a documentation category and the actual documentation itself. Additionally, it can specify a custom heading for the attribute, though a default heading will be chosen when possible.

There are four predefined documentation categories: `DocCatFunction` for attributes that appertain to function-like subjects, `DocCatVariable` for attributes that appertain to variable-like subjects, `DocCatType` for type attributes, and `DocCatStmt` for statement attributes. A custom documentation category should be used for groups of attributes

with similar functionality. Custom categories are good for providing overview information for the attributes grouped under it. For instance, the consumed annotation attributes define a custom category, `DocCatConsumed`, that explains what consumed annotations are at a high level.

Documentation content (whether it is for an attribute or a category) is written using reStructuredText (RST) syntax.

After writing the documentation for the attribute, it should be locally tested to ensure that there are no issues generating the documentation on the server. Local testing requires a fresh build of `clang-tblgen`. To generate the attribute documentation, execute the following command:

```
clang-tblgen -gen-attr-docs -I /path/to/clang/include /path/to/clang/include/clang/  
↳Basic/Attr.td -o /path/to/clang/docs/AttributeReference.rst
```

When testing locally, *do not* commit changes to `AttributeReference.rst`. This file is generated by the server automatically, and any changes made to this file will be overwritten.

Arguments

Attributes may optionally specify a list of arguments that can be passed to the attribute. Attribute arguments specify both the parsed form and the semantic form of the attribute. For example, if `Args` is `[StringArgument<"Arg1">, IntArgument<"Arg2">]` then `__attribute__((myattribute("Hello", 3)))` will be a valid use; it requires two arguments while parsing, and the `Attr` subclass' constructor for the semantic attribute will require a string and integer argument.

All arguments have a name and a flag that specifies whether the argument is optional. The associated C++ type of the argument is determined by the argument definition type. If the existing argument types are insufficient, new types can be created, but it requires modifying `utils/TableGen/ClangAttrEmitter.cpp` to properly support the type.

Other Properties

The `Attr` definition has other members which control the behavior of the attribute. Many of them are special-purpose and beyond the scope of this document, however a few deserve mention.

If the parsed form of the attribute is more complex, or differs from the semantic form, the `HasCustomParsing` bit can be set to 1 for the class, and the parsing code in `Parser::ParseGNUAttributeArgs()` can be updated for the special case. Note that this only applies to arguments with a GNU spelling – attributes with a `__declspec` spelling currently ignore this flag and are handled by `Parser::ParseMicrosoftDeclSpec`.

Note that setting this member to 1 will opt out of common attribute semantic handling, requiring extra implementation efforts to ensure the attribute appertains to the appropriate subject, etc.

If the attribute should not be propagated from a template declaration to an instantiation of the template, set the `Clone` member to 0. By default, all attributes will be cloned to template instantiations.

Attributes that do not require an AST node should set the `ASTNode` field to 0 to avoid polluting the AST. Note that anything inheriting from `TypeAttr` or `IgnoredAttr` automatically do not generate an AST node. All other attributes generate an AST node by default. The AST node is the semantic representation of the attribute.

The `LangOpts` field specifies a list of language options required by the attribute. For instance, all of the CUDA-specific attributes specify `[CUDA]` for the `LangOpts` field, and when the CUDA language option is not enabled, an “attribute ignored” warning diagnostic is emitted. Since language options are not table generated nodes, new language options must be created manually and should specify the spelling used by `LangOptions` class.

Custom accessors can be generated for an attribute based on the spelling list for that attribute. For instance, if an attribute has two different spellings: ‘Foo’ and ‘Bar’, accessors can be created: `[Accessor<"isFoo", [GNU<"Foo">]>, Accessor<"isBar", [GNU<"Bar">]>]` These accessors will be generated on the semantic form of the attribute, accepting no arguments and returning a `bool`.

Attributes that do not require custom semantic handling should set the `SemaHandler` field to 0. Note that anything inheriting from `IgnoredAttr` automatically do not get a semantic handler. All other attributes are assumed to use a semantic handler by default. Attributes without a semantic handler are not given a parsed attribute `Kind` enumerator.

Target-specific attributes may share a spelling with other attributes in different targets. For instance, the ARM and MSP430 targets both have an attribute spelled `GNU<"interrupt">`, but with different parsing and semantic requirements. To support this feature, an attribute inheriting from `TargetSpecificAttribute` may specify a `ParseKind` field. This field should be the same value between all arguments sharing a spelling, and corresponds to the parsed attribute's `Kind` enumerator. This allows attributes to share a parsed attribute kind, but have distinct semantic attribute classes. For instance, `AttributeList::AT_Interrupt` is the shared parsed attribute kind, but `ARMInterruptAttr` and `MSP430InterruptAttr` are the semantic attributes generated.

By default, when declarations are merging attributes, an attribute will not be duplicated. However, if an attribute can be duplicated during this merging stage, set `DuplicatesAllowedWhileMerging` to 1, and the attribute will be merged.

By default, attribute arguments are parsed in an evaluated context. If the arguments for an attribute should be parsed in an unevaluated context (akin to the way the argument to a `sizeof` expression is parsed), set `ParseArgumentsAsUnevaluated` to 1.

If additional functionality is desired for the semantic form of the attribute, the `AdditionalMembers` field specifies code to be copied verbatim into the semantic attribute class object, with `public` access.

Boilerplate

All semantic processing of declaration attributes happens in `lib/Sema/SemaDeclAttr.cpp`, and generally starts in the `ProcessDeclAttribute()` function. If the attribute is a “simple” attribute – meaning that it requires no custom semantic processing aside from what is automatically provided, add a call to `handleSimpleAttribute<YourAttr>(S, D, Attr);` to the switch statement. Otherwise, write a new `handleYourAttr()` function, and add that to the switch statement. Please do not implement handling logic directly in the `case` for the attribute.

Unless otherwise specified by the attribute definition, common semantic checking of the parsed attribute is handled automatically. This includes diagnosing parsed attributes that do not appertain to the given `Decl`, ensuring the correct minimum number of arguments are passed, etc.

If the attribute adds additional warnings, define a `DiagGroup` in `include/clang/Basic/DiagnosticGroups.td` named after the attribute's `Spelling` with “_”s replaced by “-”s. If there is only a single diagnostic, it is permissible to use `InGroup<DiagGroup<"your-attribute">>` directly in `DiagnosticSemaKinds.td`

All semantic diagnostics generated for your attribute, including automatically-generated ones (such as subjects and argument counts), should have a corresponding test case.

Semantic handling

Most attributes are implemented to have some effect on the compiler. For instance, to modify the way code is generated, or to add extra semantic checks for an analysis pass, etc. Having added the attribute definition and conversion to the semantic representation for the attribute, what remains is to implement the custom logic requiring use of the attribute.

The `clang::Decl` object can be queried for the presence or absence of an attribute using `hasAttr<T>()`. To obtain a pointer to the semantic representation of the attribute, `getAttr<T>` may be used.

How to add an expression or statement

Expressions and statements are one of the most fundamental constructs within a compiler, because they interact with many different parts of the AST, semantic analysis, and IR generation. Therefore, adding a new expression or statement kind into Clang requires some care. The following list details the various places in Clang where an expression or statement needs to be introduced, along with patterns to follow to ensure that the new expression or statement works well across all of the C languages. We focus on expressions, but statements are similar.

1. Introduce parsing actions into the parser. Recursive-descent parsing is mostly self-explanatory, but there are a few things that are worth keeping in mind:
 - Keep as much source location information as possible! You'll want it later to produce great diagnostics and support Clang's various features that map between source code and the AST.
 - Write tests for all of the "bad" parsing cases, to make sure your recovery is good. If you have matched delimiters (e.g., parentheses, square brackets, etc.), use `Parser::BalancedDelimiterTracker` to give nice diagnostics when things go wrong.
2. Introduce semantic analysis actions into `Sema`. Semantic analysis should always involve two functions: an `ActOnXXX` function that will be called directly from the parser, and a `BuildXXX` function that performs the actual semantic analysis and will (eventually!) build the AST node. It's fairly common for the `ActOnCXX` function to do very little (often just some minor translation from the parser's representation to `Sema`'s representation of the same thing), but the separation is still important: C++ template instantiation, for example, should always call the `BuildXXX` variant. Several notes on semantic analysis before we get into construction of the AST:
 - Your expression probably involves some types and some subexpressions. Make sure to fully check that those types, and the types of those subexpressions, meet your expectations. Add implicit conversions where necessary to make sure that all of the types line up exactly the way you want them. Write extensive tests to check that you're getting good diagnostics for mistakes and that you can use various forms of subexpressions with your expression.
 - When type-checking a type or subexpression, make sure to first check whether the type is "dependent" (`Type::isDependentType()`) or whether a subexpression is type-dependent (`Expr::isTypeDependent()`). If any of these return `true`, then you're inside a template and you can't do much type-checking now. That's normal, and your AST node (when you get there) will have to deal with this case. At this point, you can write tests that use your expression within templates, but don't try to instantiate the templates.
 - For each subexpression, be sure to call `Sema::CheckPlaceholderExpr()` to deal with "weird" expressions that don't behave well as subexpressions. Then, determine whether you need to perform lvalue-to-rvalue conversions (`Sema::DefaultLvalueConversions`) or the usual unary conversions (`Sema::UsualUnaryConversions`), for places where the subexpression is producing a value you intend to use.
 - Your `BuildXXX` function will probably just return `ExprError()` at this point, since you don't have an AST. That's perfectly fine, and shouldn't impact your testing.
3. Introduce an AST node for your new expression. This starts with declaring the node in `include/Basic/StmtNodes.td` and creating a new class for your expression in the appropriate `include/AST/Expr*.h` header. It's best to look at the class for a similar expression to get ideas, and there are some specific things to watch for:
 - If you need to allocate memory, use the `ASTContext` allocator to allocate memory. Never use raw `malloc` or `new`, and never hold any resources in an AST node, because the destructor of an AST node is never called.
 - Make sure that `getSourceRange()` covers the exact source range of your expression. This is needed for diagnostics and for IDE support.

- Make sure that `children()` visits all of the subexpressions. This is important for a number of features (e.g., IDE support, C++ variadic templates). If you have sub-types, you'll also need to visit those sub-types in `RecursiveASTVisitor` and `DataRecursiveASTVisitor`.
 - Add printing support (`StmtPrinter.cpp`) for your expression.
 - Add profiling support (`StmtProfile.cpp`) for your AST node, noting the distinguishing (non-source location) characteristics of an instance of your expression. Omitting this step will lead to hard-to-diagnose failures regarding matching of template declarations.
 - Add serialization support (`ASTReaderStmt.cpp`, `ASTWriterStmt.cpp`) for your AST node.
4. Teach semantic analysis to build your AST node. At this point, you can wire up your `Sema::BuildXXX` function to actually create your AST. A few things to check at this point:
 - If your expression can construct a new C++ class or return a new Objective-C object, be sure to update and then call `Sema::MaybeBindToTemporary` for your just-created AST node to be sure that the object gets properly destructed. An easy way to test this is to return a C++ class with a private destructor: semantic analysis should flag an error here with the attempt to call the destructor.
 - Inspect the generated AST by printing it using `clang -cc1 -ast-print`, to make sure you're capturing all of the important information about how the AST was written.
 - Inspect the generated AST under `clang -cc1 -ast-dump` to verify that all of the types in the generated AST line up the way you want them. Remember that clients of the AST should never have to "think" to understand what's going on. For example, all implicit conversions should show up explicitly in the AST.
 - Write tests that use your expression as a subexpression of other, well-known expressions. Can you call a function using your expression as an argument? Can you use the ternary operator?
 5. Teach code generation to create IR to your AST node. This step is the first (and only) that requires knowledge of LLVM IR. There are several things to keep in mind:
 - Code generation is separated into scalar/aggregate/complex and lvalue/rvalue paths, depending on what kind of result your expression produces. On occasion, this requires some careful factoring of code to avoid duplication.
 - `CodeGenFunction` contains functions `ConvertType` and `ConvertTypeForMem` that convert Clang's types (`clang::Type*` or `clang::QualType`) to LLVM types. Use the former for values, and the later for memory locations: test with the C++ "bool" type to check this. If you find that you are having to use LLVM bitcasts to make the subexpressions of your expression have the type that your expression expects, STOP! Go fix semantic analysis and the AST so that you don't need these bitcasts.
 - The `CodeGenFunction` class has a number of helper functions to make certain operations easy, such as generating code to produce an lvalue or an rvalue, or to initialize a memory location with a given value. Prefer to use these functions rather than directly writing loads and stores, because these functions take care of some of the tricky details for you (e.g., for exceptions).
 - If your expression requires some special behavior in the event of an exception, look at the `push*Cleanup` functions in `CodeGenFunction` to introduce a cleanup. You shouldn't have to deal with exception-handling directly.
 - Testing is extremely important in IR generation. Use `clang -cc1 -emit-llvm` and `FileCheck` to verify that you're generating the right IR.
 6. Teach template instantiation how to cope with your AST node, which requires some fairly simple code:
 - Make sure that your expression's constructor properly computes the flags for type dependence (i.e., the type your expression produces can change from one instantiation to the next), value dependence (i.e., the constant value your expression produces can change from one instantiation to the next), instantiation dependence (i.e., a template parameter occurs anywhere in your expression), and whether your expression

contains a parameter pack (for variadic templates). Often, computing these flags just means combining the results from the various types and subexpressions.

- Add `TransformXXX` and `RebuildXXX` functions to the `TreeTransform` class template in `Sema`. `TransformXXX` should (recursively) transform all of the subexpressions and types within your expression, using `getDerived().TransformYYY`. If all of the subexpressions and types transform without error, it will then call the `RebuildXXX` function, which will in turn call `getSema().BuildXXX` to perform semantic analysis and build your expression.
 - To test template instantiation, take those tests you wrote to make sure that you were type checking with type-dependent expressions and dependent types (from step #2) and instantiate those templates with various types, some of which type-check and some that don't, and test the error messages in each case.
7. There are some “extras” that make other features work better. It's worth handling these extras to give your expression complete integration into Clang:
- Add code completion support for your expression in `SemaCodeComplete.cpp`.
 - If your expression has types in it, or has any “interesting” features other than subexpressions, extend `libclang's CursorVisitor` to provide proper visitation for your expression, enabling various IDE features such as syntax highlighting, cross-referencing, and so on. The `c-index-test` helper program can be used to test these features.

Driver Design & Internals

- *Introduction*
- *Features and Goals*
 - *GCC Compatibility*
 - *Flexible*
 - *Low Overhead*
 - *Simple*
- *Internal Design and Implementation*
 - *Internals Introduction*
 - *Design Overview*
 - *Driver Stages*
 - *Additional Notes*
 - * *The Compilation Object*
 - * *Unified Parsing & Pipelining*
 - * *ToolChain Argument Translation*
 - * *Unused Argument Warnings*
 - *Relation to GCC Driver Concepts*

Introduction

This document describes the Clang driver. The purpose of this document is to describe both the motivation and design goals for the driver, as well as details of the internal implementation.

Features and Goals

The Clang driver is intended to be a production quality compiler driver providing access to the Clang compiler and tools, with a command line interface which is compatible with the gcc driver.

Although the driver is part of and driven by the Clang project, it is logically a separate tool which shares many of the same goals as Clang:

Features

- *GCC Compatibility*
- *Flexible*
- *Low Overhead*
- *Simple*

GCC Compatibility

The number one goal of the driver is to ease the adoption of Clang by allowing users to drop Clang into a build system which was designed to call GCC. Although this makes the driver much more complicated than might otherwise be necessary, we decided that being very compatible with the gcc command line interface was worth it in order to allow users to quickly test clang on their projects.

Flexible

The driver was designed to be flexible and easily accommodate new uses as we grow the clang and LLVM infrastructure. As one example, the driver can easily support the introduction of tools which have an integrated assembler; something we hope to add to LLVM in the future.

Similarly, most of the driver functionality is kept in a library which can be used to build other tools which want to implement or accept a gcc like interface.

Low Overhead

The driver should have as little overhead as possible. In practice, we found that the gcc driver by itself incurred a small but meaningful overhead when compiling many small files. The driver doesn't do much work compared to a compilation, but we have tried to keep it as efficient as possible by following a few simple principles:

- Avoid memory allocation and string copying when possible.
- Don't parse arguments more than once.
- Provide a few simple interfaces for efficiently searching arguments.

Simple

Finally, the driver was designed to be “as simple as possible”, given the other goals. Notably, trying to be completely compatible with the gcc driver adds a significant amount of complexity. However, the design of the driver attempts to mitigate this complexity by dividing the process into a number of independent stages instead of a single monolithic task.

Internal Design and Implementation

- *Internals Introduction*
- *Design Overview*
- *Driver Stages*
- *Additional Notes*
- *Relation to GCC Driver Concepts*

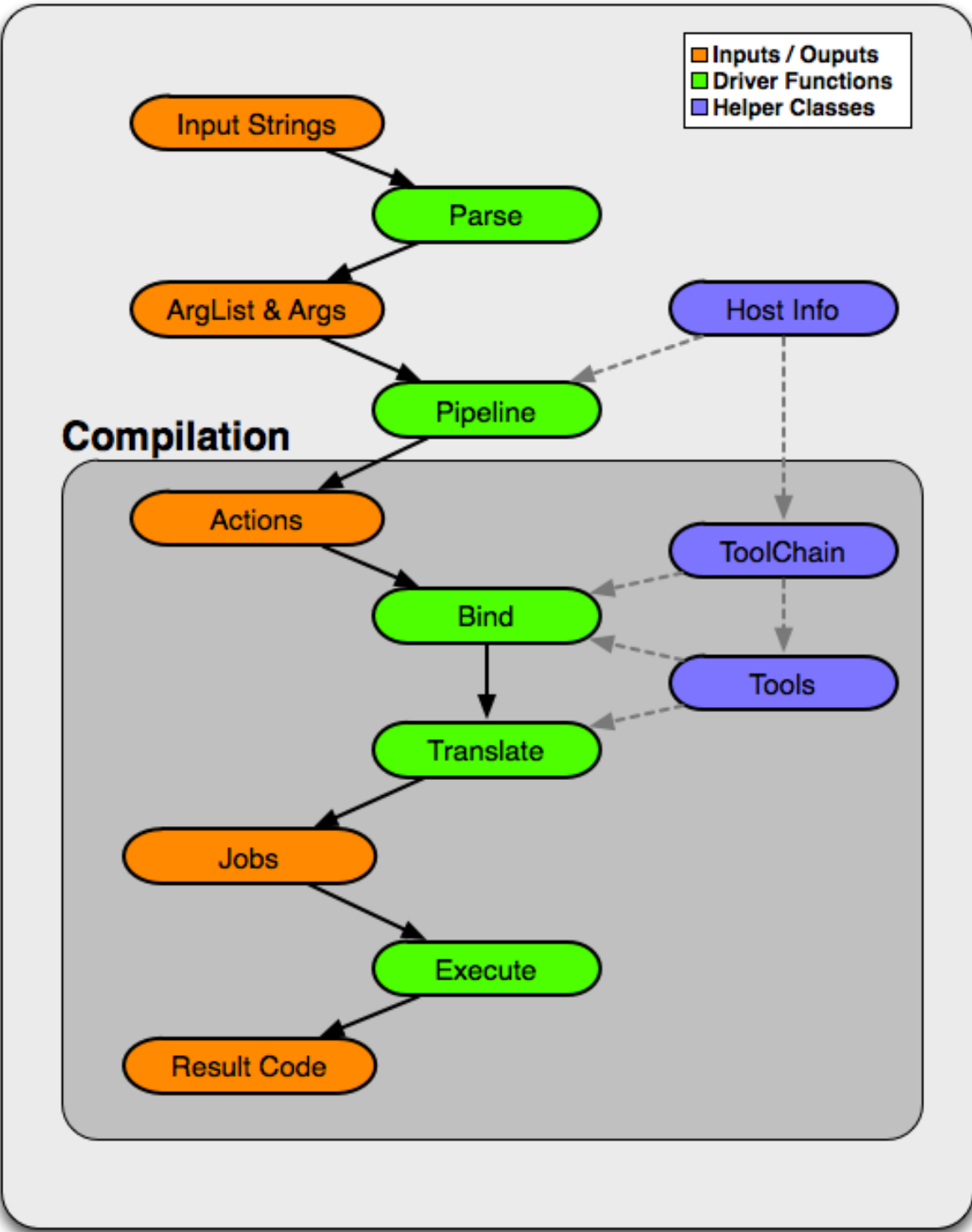
Internals Introduction

In order to satisfy the stated goals, the driver was designed to completely subsume the functionality of the gcc executable; that is, the driver should not need to delegate to gcc to perform subtasks. On Darwin, this implies that the Clang driver also subsumes the gcc driver-driver, which is used to implement support for building universal images (binaries and object files). This also implies that the driver should be able to call the language specific compilers (e.g. cc1) directly, which means that it must have enough information to forward command line arguments to child processes correctly.

Design Overview

The diagram below shows the significant components of the driver architecture and how they relate to one another. The orange components represent concrete data structures built by the driver, the green components indicate conceptually distinct stages which manipulate these data structures, and the blue components are important helper classes.

Driver



Driver Stages

The driver functionality is conceptually divided into five stages:

1. Parse: Option Parsing

The command line argument strings are decomposed into arguments (`Arg` instances). The driver expects to understand all available options, although there is some facility for just passing certain classes of options through (like `-Wl,`).

Each argument corresponds to exactly one abstract `Option` definition, which describes how the option is parsed along with some additional metadata. The `Arg` instances themselves are lightweight and merely contain enough information for clients to determine which option they correspond to and their values (if they have additional parameters).

For example, a command line like “-Ifoo -I foo” would parse to two `Arg` instances (a `JoinedArg` and a `SeparateArg` instance), but each would refer to the same `Option`.

Options are lazily created in order to avoid populating all `Option` classes when the driver is loaded. Most of the driver code only needs to deal with options by their unique ID (e.g., `options::OPT_I`),

`Arg` instances themselves do not generally store the values of parameters. In many cases, this would simply result in creating unnecessary string copies. Instead, `Arg` instances are always embedded inside an `ArgList` structure, which contains the original vector of argument strings. Each `Arg` itself only needs to contain an index into this vector instead of storing its values directly.

The clang driver can dump the results of this stage using the `-###` flag (which must precede any actual command line arguments). For example:

```
$ clang -### -Xarch_i386 -fomit-frame-pointer -Wa,-fast -Ifoo -I foo t.c
Option 0 - Name: "-Xarch_", Values: {"i386", "-fomit-frame-pointer"}
Option 1 - Name: "-Wa,", Values: {"-fast"}
Option 2 - Name: "-I", Values: {"foo"}
Option 3 - Name: "-I", Values: {"foo"}
Option 4 - Name: "<input>", Values: {"t.c"}
```

After this stage is complete the command line should be broken down into well defined option objects with their appropriate parameters. Subsequent stages should rarely, if ever, need to do any string processing.

2. Pipeline: Compilation Action Construction

Once the arguments are parsed, the tree of subprocess jobs needed for the desired compilation sequence are constructed. This involves determining the input files and their types, what work is to be done on them (preprocess, compile, assemble, link, etc.), and constructing a list of `Action` instances for each task. The result is a list of one or more top-level actions, each of which generally corresponds to a single output (for example, an object or linked executable).

The majority of `Actions` correspond to actual tasks, however there are two special `Actions`. The first is `InputAction`, which simply serves to adapt an input argument for use as an input to other `Actions`. The second is `BindArchAction`, which conceptually alters the architecture to be used for all of its input `Actions`.

The clang driver can dump the results of this stage using the `-ccc-print-phases` flag. For example:

```
$ clang -ccc-print-phases -x c t.c -x assembler t.s
0: input, "t.c", c
1: preprocessor, {0}, cpp-output
2: compiler, {1}, assembler
3: assembler, {2}, object
4: input, "t.s", assembler
5: assembler, {4}, object
6: linker, {3, 5}, image
```

Here the driver is constructing seven distinct actions, four to compile the “t.c” input into an object file, two to assemble the “t.s” input, and one to link them together.

A rather different compilation pipeline is shown here; in this example there are two top level actions to compile the input files into two separate object files, where each object file is built using `lipo` to merge results built for two separate architectures.

```
$ clang -ccc-print-phases -c -arch i386 -arch x86_64 t0.c t1.c
0: input, "t0.c", c
1: preprocessor, {0}, cpp-output
2: compiler, {1}, assembler
3: assembler, {2}, object
4: bind-arch, "i386", {3}, object
5: bind-arch, "x86_64", {3}, object
6: lipo, {4, 5}, object
7: input, "t1.c", c
8: preprocessor, {7}, cpp-output
9: compiler, {8}, assembler
10: assembler, {9}, object
11: bind-arch, "i386", {10}, object
12: bind-arch, "x86_64", {10}, object
13: lipo, {11, 12}, object
```

After this stage is complete the compilation process is divided into a simple set of actions which need to be performed to produce intermediate or final outputs (in some cases, like `-fsyntax-only`, there is no “real” final output). Phases are well known compilation steps, such as “preprocess”, “compile”, “assemble”, “link”, etc.

3. Bind: Tool & Filename Selection

This stage (in conjunction with the Translate stage) turns the tree of Actions into a list of actual subprocess to run. Conceptually, the driver performs a top down matching to assign Action(s) to Tools. The ToolChain is responsible for selecting the tool to perform a particular action; once selected the driver interacts with the tool to see if it can match additional actions (for example, by having an integrated preprocessor).

Once Tools have been selected for all actions, the driver determines how the tools should be connected (for example, using an inprocess module, pipes, temporary files, or user provided filenames). If an output file is required, the driver also computes the appropriate file name (the suffix and file location depend on the input types and options such as `-save-temps`).

The driver interacts with a ToolChain to perform the Tool bindings. Each ToolChain contains information about all the tools needed for compilation for a particular architecture, platform, and operating system. A single driver invocation may query multiple ToolChains during one compilation in order to interact with tools for separate architectures.

The results of this stage are not computed directly, but the driver can print the results via the `-ccc-print-bindings` option. For example:

```
$ clang -ccc-print-bindings -arch i386 -arch ppc t0.c
# "i386-apple-darwin9" - "clang", inputs: ["t0.c"], output: "/tmp/cc-Sn4RKF.s"
# "i386-apple-darwin9" - "darwin::Assemble", inputs: ["/tmp/cc-Sn4RKF.s"],
  ↳ output: "/tmp/cc-gvSnbS.o"
# "i386-apple-darwin9" - "darwin::Link", inputs: ["/tmp/cc-gvSnbS.o"], output: "/
  ↳ tmp/cc-jgHQxi.out"
# "ppc-apple-darwin9" - "gcc::Compile", inputs: ["t0.c"], output: "/tmp/cc-Q0bTox.
  ↳ s"
# "ppc-apple-darwin9" - "gcc::Assemble", inputs: ["/tmp/cc-Q0bTox.s"], output: "/
  ↳ tmp/cc-WCdicw.o"
```

```
# "ppc-apple-darwin9" - "gcc::Link", inputs: ["/tmp/cc-WCdicw.o"], output: "/tmp/cc-HHBEBh.out"
# "i386-apple-darwin9" - "darwin::Lipo", inputs: ["/tmp/cc-jgHQxi.out", "/tmp/cc-HHBEBh.out"], output: "a.out"
```

This shows the tool chain, tool, inputs and outputs which have been bound for this compilation sequence. Here clang is being used to compile t0.c on the i386 architecture and darwin specific versions of the tools are being used to assemble and link the result, but generic gcc versions of the tools are being used on PowerPC.

4. Translate: Tool Specific Argument Translation

Once a Tool has been selected to perform a particular Action, the Tool must construct concrete Commands which will be executed during compilation. The main work is in translating from the gcc style command line options to whatever options the subprocess expects.

Some tools, such as the assembler, only interact with a handful of arguments and just determine the path of the executable to call and pass on their input and output arguments. Others, like the compiler or the linker, may translate a large number of arguments in addition.

The ArgList class provides a number of simple helper methods to assist with translating arguments; for example, to pass on only the last of arguments corresponding to some option, or all arguments for an option.

The result of this stage is a list of Commands (executable paths and argument strings) to execute.

5. Execute

Finally, the compilation pipeline is executed. This is mostly straightforward, although there is some interaction with options like `-pipe`, `-pass-exit-codes` and `-time`.

Additional Notes

The Compilation Object

The driver constructs a Compilation object for each set of command line arguments. The Driver itself is intended to be invariant during construction of a Compilation; an IDE should be able to construct a single long lived driver instance to use for an entire build, for example.

The Compilation object holds information that is particular to each compilation sequence. For example, the list of used temporary files (which must be removed once compilation is finished) and result files (which should be removed if compilation fails).

Unified Parsing & Pipelining

Parsing and pipelining both occur without reference to a Compilation instance. This is by design; the driver expects that both of these phases are platform neutral, with a few very well defined exceptions such as whether the platform uses a driver driver.

ToolChain Argument Translation

In order to match gcc very closely, the clang driver currently allows tool chains to perform their own translation of the argument list (into a new ArgList data structure). Although this allows the clang driver to match gcc easily, it also makes the driver operation much harder to understand (since the Tools stop seeing some arguments the user provided, and see new ones instead).

For example, on Darwin `-gfull` gets translated into two separate arguments, `-g` and `-fno-eliminate-unused-debug-symbols`. Trying to write Tool logic to do something with `-gfull` will not work, because Tool argument translation is done after the arguments have been translated.

A long term goal is to remove this tool chain specific translation, and instead force each tool to change its own logic to do the right thing on the untranslated original arguments.

Unused Argument Warnings

The driver operates by parsing all arguments but giving Tools the opportunity to choose which arguments to pass on. One downside of this infrastructure is that if the user misspells some option, or is confused about which options to use, some command line arguments the user really cared about may go unused. This problem is particularly important when using clang as a compiler, since the clang compiler does not support anywhere near all the options that gcc does, and we want to make sure users know which ones are being used.

To support this, the driver maintains a bit associated with each argument of whether it has been used (at all) during the compilation. This bit usually doesn't need to be set by hand, as the key `ArgList` accessors will set it automatically.

When a compilation is successful (there are no errors), the driver checks the bit and emits an “unused argument” warning for any arguments which were never accessed. This is conservative (the argument may not have been used to do what the user wanted) but still catches the most obvious cases.

Relation to GCC Driver Concepts

For those familiar with the gcc driver, this section provides a brief overview of how things from the gcc driver map to the clang driver.

- **Driver Driver**

The driver driver is fully integrated into the clang driver. The driver simply constructs additional Actions to bind the architecture during the *Pipeline* phase. The tool chain specific argument translation is responsible for handling `-Xarch_`.

The one caveat is that this approach requires `-Xarch_` not be used to alter the compilation itself (for example, one cannot provide `-S` as an `-Xarch_` argument). The driver attempts to reject such invocations, and overall there isn't a good reason to abuse `-Xarch_` to that end in practice.

The upside is that the clang driver is more efficient and does little extra work to support universal builds. It also provides better error reporting and UI consistency.

- **Specs**

The clang driver has no direct correspondent for “specs”. The majority of the functionality that is embedded in specs is in the Tool specific argument translation routines. The parts of specs which control the compilation pipeline are generally part of the *Pipeline* stage.

- **Toolchains**

The gcc driver has no direct understanding of tool chains. Each gcc binary roughly corresponds to the information which is embedded inside a single ToolChain.

The clang driver is intended to be portable and support complex compilation environments. All platform and tool chain specific code should be protected behind either abstract or well defined interfaces (such as whether the platform supports use as a driver driver).

Pretokenized Headers (PTH)

This document first describes the low-level interface for using PTH and then briefly elaborates on its design and implementation. If you are interested in the end-user view, please see the *User's Manual*.

Using Pretokenized Headers with clang (Low-level Interface)

The Clang compiler frontend, `clang -cc1`, supports three command line options for generating and using PTH files.

To generate PTH files using `clang -cc1`, use the option `-emit-pth`:

```
$ clang -cc1 test.h -emit-pth -o test.h.pth
```

This option is transparently used by `clang` when generating PTH files. Similarly, PTH files can be used as prefix headers using the `-include-pth` option:

```
$ clang -cc1 -include-pth test.h.pth test.c -o test.s
```

Alternatively, Clang's PTH files can be used as a raw “token-cache” (or “content” cache) of the source included by the original header file. This means that the contents of the PTH file are searched as substitutes for *any* source files that are used by `clang -cc1` to process a source file. This is done by specifying the `-token-cache` option:

```
$ cat test.h
#include <stdio.h>
$ clang -cc1 -emit-pth test.h -o test.h.pth
$ cat test.c
#include "test.h"
$ clang -cc1 test.c -o test -token-cache test.h.pth
```

In this example the contents of `stdio.h` (and the files it includes) will be retrieved from `test.h.pth`, as the PTH file is being used in this case as a raw cache of the contents of `test.h`. This is a low-level interface used to both implement the high-level PTH interface as well as to provide alternative means to use PTH-style caching.

PTH Design and Implementation

Unlike GCC's precompiled headers, which cache the full ASTs and preprocessor state of a header file, Clang's pretokenized header files mainly cache the raw lexer *tokens* that are needed to segment the stream of characters in a source file into keywords, identifiers, and operators. Consequently, PTH serves to mainly directly speed up the lexing and preprocessing of a source file, while parsing and type-checking must be completely redone every time a PTH file is used.

Basic Design Tradeoffs

In the long term there are plans to provide an alternate PCH implementation for Clang that also caches the work for parsing and type checking the contents of header files. The current implementation of PCH in Clang as pretokenized header files was motivated by the following factors:

Language independence PTH files work with any language that Clang's lexer can handle, including C, Objective-C, and (in the early stages) C++. This means development on language features at the parsing level or above (which is basically almost all interesting pieces) does not require PTH to be modified.

Simple design Relatively speaking, PTH has a simple design and implementation, making it easy to test. Further, because the machinery for PTH resides at the lower-levels of the Clang library stack it is fairly straightforward to profile and optimize.

Further, compared to GCC's PCH implementation (which is the dominate precompiled header file implementation that Clang can be directly compared against) the PTH design in Clang yields several attractive features:

Architecture independence In contrast to GCC's PCH files (and those of several other compilers), Clang's PTH files are architecture independent, requiring only a single PTH file when building a program for multiple architectures.

For example, on Mac OS X one may wish to compile a "universal binary" that runs on PowerPC, 32-bit Intel (i386), and 64-bit Intel architectures. In contrast, GCC requires a PCH file for each architecture, as the definitions of types in the AST are architecture-specific. Since a Clang PTH file essentially represents a lexical cache of header files, a single PTH file can be safely used when compiling for multiple architectures. This can also reduce compile times because only a single PTH file needs to be generated during a build instead of several.

Reduced memory pressure Similar to GCC, Clang reads PTH files via the use of memory mapping (i.e., `mmap`). Clang, however, memory maps PTH files as read-only, meaning that multiple invocations of `clang -cc1` can share the same pages in memory from a memory-mapped PTH file. In comparison, GCC also memory maps its PCH files but also modifies those pages in memory, incurring the copy-on-write costs. The read-only nature of PTH can greatly reduce memory pressure for builds involving multiple cores, thus improving overall scalability.

Fast generation PTH files can be generated in a small fraction of the time needed to generate GCC's PCH files. Since PTH/PCH generation is a serial operation that typically blocks progress during a build, faster generation time leads to improved processor utilization with parallel builds on multicore machines.

Despite these strengths, PTH's simple design suffers some algorithmic handicaps compared to other PCH strategies such as those used by GCC. While PTH can greatly speed up the processing time of a header file, the amount of work required to process a header file is still roughly linear in the size of the header file. In contrast, the amount of work done by GCC to process a precompiled header is (theoretically) constant (the ASTs for the header are literally memory mapped into the compiler). This means that only the pieces of the header file that are referenced by the source file including the header are the only ones the compiler needs to process during actual compilation. While GCC's particular implementation of PCH mitigates some of these algorithmic strengths via the use of copy-on-write pages, the approach itself can fundamentally dominate at an algorithmic level, especially when one considers header files of arbitrary size.

There is also a PCH implementation for Clang based on the lazy deserialization of ASTs. This approach theoretically has the same constant-time algorithmic advantages just mentioned but also retains some of the strengths of PTH such as reduced memory pressure (ideal for multi-core builds).

Internal PTH Optimizations

While the main optimization employed by PTH is to reduce lexing time of header files by caching pre-lexed tokens, PTH also employs several other optimizations to speed up the processing of header files:

- **stat caching:** PTH files cache information obtained via calls to `stat` that `clang -cc1` uses to resolve which files are included by `#include` directives. This greatly reduces the overhead involved in context-switching to the kernel to resolve included files.
- **Fast skipping of `#ifdef ... #endif` chains:** PTH files record the basic structure of nested preprocessor blocks. When the condition of the preprocessor block is false, all of its tokens are immediately skipped instead of requiring them to be handled by Clang's preprocessor.

Precompiled Header and Modules Internals

- *Using Precompiled Headers with clang*
- *Design Philosophy*
- *AST File Contents*
 - *Metadata Block*
 - *Source Manager Block*
 - *Preprocessor Block*
 - *Types Block*
 - *Declarations Block*
 - *Statements and Expressions*
 - *Identifier Table Block*
 - *Method Pool Block*
- *AST Reader Integration Points*
- *Chained precompiled headers*
- *Modules*

This document describes the design and implementation of Clang’s precompiled headers (PCH) and modules. If you are interested in the end-user view, please see the *User’s Manual*.

Using Precompiled Headers with clang

The Clang compiler frontend, `clang -cc1`, supports two command line options for generating and using PCH files.

To generate PCH files using `clang -cc1`, use the option `-emit-pch`:

```
$ clang -cc1 test.h -emit-pch -o test.h.pch
```

This option is transparently used by `clang` when generating PCH files. The resulting PCH file contains the serialized form of the compiler’s internal representation after it has completed parsing and semantic analysis. The PCH file can then be used as a prefix header with the `-include-pch` option:

```
$ clang -cc1 -include-pch test.h.pch test.c -o test.s
```

Design Philosophy

Precompiled headers are meant to improve overall compile times for projects, so the design of precompiled headers is entirely driven by performance concerns. The use case for precompiled headers is relatively simple: when there is a common set of headers that is included in nearly every source file in the project, we *precompile* that bundle of headers into a single precompiled header (PCH file). Then, when compiling the source files in the project, we load the PCH file first (as a prefix header), which acts as a stand-in for that bundle of headers.

A precompiled header implementation improves performance when:

- Loading the PCH file is significantly faster than re-parsing the bundle of headers stored within the PCH file. Thus, a precompiled header design attempts to minimize the cost of reading the PCH file. Ideally, this cost should not vary with the size of the precompiled header file.
- The cost of generating the PCH file initially is not so large that it counters the per-source-file performance improvement due to eliminating the need to parse the bundled headers in the first place. This is particularly important on multi-core systems, because PCH file generation serializes the build when all compilations require the PCH file to be up-to-date.

Modules, as implemented in Clang, use the same mechanisms as precompiled headers to save a serialized AST file (one per module) and use those AST modules. From an implementation standpoint, modules are a generalization of precompiled headers, lifting a number of restrictions placed on precompiled headers. In particular, there can only be one precompiled header and it must be included at the beginning of the translation unit. The extensions to the AST file format required for modules are discussed in the section on [modules](#).

Clang’s AST files are designed with a compact on-disk representation, which minimizes both creation time and the time required to initially load the AST file. The AST file itself contains a serialized representation of Clang’s abstract syntax trees and supporting data structures, stored using the same compressed bitstream as [LLVM’s bytecode file format](#).

Clang’s AST files are loaded “lazily” from disk. When an AST file is initially loaded, Clang reads only a small amount of data from the AST file to establish where certain important data structures are stored. The amount of data read in this initial load is independent of the size of the AST file, such that a larger AST file does not lead to longer AST load times. The actual header data in the AST file — macros, functions, variables, types, etc. — is loaded only when it is referenced from the user’s code, at which point only that entity (and those entities it depends on) are deserialized from the AST file. With this approach, the cost of using an AST file for a translation unit is proportional to the amount of code actually used from the AST file, rather than being proportional to the size of the AST file itself.

When given the `-print-stats` option, Clang produces statistics describing how much of the AST file was actually loaded from disk. For a simple “Hello, World!” program that includes the Apple `Cocoa.h` header (which is built as a precompiled header), this option illustrates how little of the actual precompiled header is required:

```
*** AST File Statistics:
 895/39981 source location entries read (2.238563%)
 19/15315 types read (0.124061%)
20/82685 declarations read (0.024188%)
154/58070 identifiers read (0.265197%)
 0/7260 selectors read (0.000000%)
 0/30842 statements read (0.000000%)
 4/8400 macros read (0.047619%)
 1/4995 lexical declcontexts read (0.020020%)
 0/4413 visible declcontexts read (0.000000%)
 0/7230 method pool entries read (0.000000%)
 0 method pool misses
```

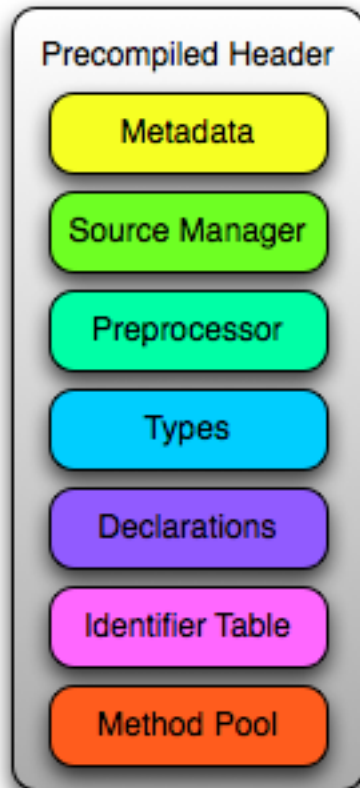
For this small program, only a tiny fraction of the source locations, types, declarations, identifiers, and macros were actually deserialized from the precompiled header. These statistics can be useful to determine whether the AST file implementation can be improved by making more of the implementation lazy.

Precompiled headers can be chained. When you create a PCH while including an existing PCH, Clang can create the new PCH by referencing the original file and only writing the new data to the new file. For example, you could create a PCH out of all the headers that are very commonly used throughout your project, and then create a PCH for every single source file in the project that includes the code that is specific to that file, so that recompiling the file itself is very fast, without duplicating the data from the common headers for every file. The mechanisms behind chained precompiled headers are discussed in a [later section](#).

AST File Contents

An AST file produced by clang is an object file container with a `clangast` (COFF) or `__clangast` (ELF and Mach-O) section containing the serialized AST. Other target-specific sections in the object file container are used to hold debug information for the data types defined in the AST. Tools built on top of libclang that do not need debug information may also produce raw AST files that only contain the serialized AST.

The `clangast` section is organized into several different blocks, each of which contains the serialized representation of a part of Clang's internal representation. Each of the blocks corresponds to either a block or a record within LLVM's [bitstream format](#). The contents of each of these logical blocks are described below.



The `llvm-objdump` utility provides a `-raw-clang-ast` option to extract the binary contents of the AST section from an object file container.

The `llvm-bcanalyzer` utility can be used to examine the actual structure of the bitstream for the AST section. This information can be used both to help understand the structure of the AST section and to isolate areas where the AST representation can still be optimized, e.g., through the introduction of abbreviations.

Metadata Block

The metadata block contains several records that provide information about how the AST file was built. This metadata is primarily used to validate the use of an AST file. For example, a precompiled header built for a 32-bit x86 target cannot be used when compiling for a 64-bit x86 target. The metadata block contains information about:

Language options Describes the particular language dialect used to compile the AST file, including major options (e.g., Objective-C support) and more minor options (e.g., support for “//” comments). The contents of this record correspond to the `LangOptions` class.

Target architecture The target triple that describes the architecture, platform, and ABI for which the AST file was generated, e.g., `i386-apple-darwin9`.

AST version The major and minor version numbers of the AST file format. Changes in the minor version number should not affect backward compatibility, while changes in the major version number imply that a newer compiler cannot read an older precompiled header (and vice-versa).

Original file name The full path of the header that was used to generate the AST file.

Predefines buffer Although not explicitly stored as part of the metadata, the predefines buffer is used in the validation of the AST file. The predefines buffer itself contains code generated by the compiler to initialize the preprocessor state according to the current target, platform, and command-line options. For example, the predefines buffer will contain `#define __STDC__ 1` when we are compiling C without Microsoft extensions. The predefines buffer itself is stored within the *Source Manager Block*, but its contents are verified along with the rest of the metadata.

A chained PCH file (that is, one that references another PCH) and a module (which may import other modules) have additional metadata containing the list of all AST files that this AST file depends on. Each of those files will be loaded along with this AST file.

For chained precompiled headers, the language options, target architecture and predefines buffer data is taken from the end of the chain, since they have to match anyway.

Source Manager Block

The source manager block contains the serialized representation of Clang’s *SourceManager* class, which handles the mapping from source locations (as represented in Clang’s abstract syntax tree) into actual column/line positions within a source file or macro instantiation. The AST file’s representation of the source manager also includes information about all of the headers that were (transitively) included when building the AST file.

The bulk of the source manager block is dedicated to information about the various files, buffers, and macro instantiations into which a source location can refer. Each of these is referenced by a numeric “file ID”, which is a unique number (allocated starting at 1) stored in the source location. Clang serializes the information for each kind of file ID, along with an index that maps file IDs to the position within the AST file where the information about that file ID is stored. The data associated with a file ID is loaded only when required by the front end, e.g., to emit a diagnostic that includes a macro instantiation history inside the header itself.

The source manager block also contains information about all of the headers that were included when building the AST file. This includes information about the controlling macro for the header (e.g., when the preprocessor identified that the contents of the header dependent on a macro like `LLVM_CLANG_SOURCEMANAGER_H`).

Preprocessor Block

The preprocessor block contains the serialized representation of the preprocessor. Specifically, it contains all of the macros that have been defined by the end of the header used to build the AST file, along with the token sequences that comprise each macro. The macro definitions are only read from the AST file when the name of the macro first occurs in the program. This lazy loading of macro definitions is triggered by lookups into the *identifier table*.

Types Block

The types block contains the serialized representation of all of the types referenced in the translation unit. Each Clang type node (*PointerType*, *FunctionProtoType*, etc.) has a corresponding record type in the AST file. When types are deserialized from the AST file, the data within the record is used to reconstruct the appropriate type node using the AST context.

Each type has a unique type ID, which is an integer that uniquely identifies that type. Type ID 0 represents the NULL type, type IDs less than `NUM_PREDEF_TYPE_IDS` represent predefined types (`void`, `float`, etc.), while other “user-defined” type IDs are assigned consecutively from `NUM_PREDEF_TYPE_IDS` upward as the types are encountered. The AST file has an associated mapping from the user-defined types block to the location within the types block where the serialized representation of that type resides, enabling lazy deserialization of types. When a type is referenced from within the AST file, that reference is encoded using the type ID shifted left by 3 bits. The lower three bits are used to represent the `const`, `volatile`, and `restrict` qualifiers, as in Clang’s *QualType* class.

Declarations Block

The declarations block contains the serialized representation of all of the declarations referenced in the translation unit. Each Clang declaration node (`VarDecl`, `FunctionDecl`, etc.) has a corresponding record type in the AST file. When declarations are deserialized from the AST file, the data within the record is used to build and populate a new instance of the corresponding `Decl` node. As with types, each declaration node has a numeric ID that is used to refer to that declaration within the AST file. In addition, a lookup table provides a mapping from that numeric ID to the offset within the precompiled header where that declaration is described.

Declarations in Clang’s abstract syntax trees are stored hierarchically. At the top of the hierarchy is the translation unit (`TranslationUnitDecl`), which contains all of the declarations in the translation unit but is not actually written as a specific declaration node. Its child declarations (such as functions or struct types) may also contain other declarations inside them, and so on. Within Clang, each declaration is stored within a *declaration context*, as represented by the `DeclContext` class. Declaration contexts provide the mechanism to perform name lookup within a given declaration (e.g., find the member named `x` in a structure) and iterate over the declarations stored within a context (e.g., iterate over all of the fields of a structure for structure layout).

In Clang’s AST file format, deserializing a declaration that is a `DeclContext` is a separate operation from deserializing all of the declarations stored within that declaration context. Therefore, Clang will deserialize the translation unit declaration without deserializing the declarations within that translation unit. When required, the declarations stored within a declaration context will be deserialized. There are two representations of the declarations within a declaration context, which correspond to the name-lookup and iteration behavior described above:

- When the front end performs name lookup to find a name `x` within a given declaration context (for example, during semantic analysis of the expression `p->x`, where `p`’s type is defined in the precompiled header), Clang refers to an on-disk hash table that maps from the names within that declaration context to the declaration IDs that represent each visible declaration with that name. The actual declarations will then be deserialized to provide the results of name lookup.
- When the front end performs iteration over all of the declarations within a declaration context, all of those declarations are immediately de-serialized. For large declaration contexts (e.g., the translation unit), this operation is expensive; however, large declaration contexts are not traversed in normal compilation, since such a traversal is unnecessary. However, it is common for the code generator and semantic analysis to traverse declaration contexts for structs, classes, unions, and enumerations, although those contexts contain relatively few declarations in the common case.

Statements and Expressions

Statements and expressions are stored in the AST file in both the *types* and the *declarations* blocks, because every statement or expression will be associated with either a type or declaration. The actual statement and expression records are stored immediately following the declaration or type that owns the statement or expression. For example, the statement representing the body of a function will be stored directly following the declaration of the function.

As with types and declarations, each statement and expression kind in Clang’s abstract syntax tree (`ForStmt`, `CallExpr`, etc.) has a corresponding record type in the AST file, which contains the serialized representation of that statement or expression. Each substatement or subexpression within an expression is stored as a separate record

(which keeps most records to a fixed size). Within the AST file, the subexpressions of an expression are stored, in reverse order, prior to the expression that owns those expression, using a form of [Reverse Polish Notation](#). For example, an expression `3 - 4 + 5` would be represented as follows:

<code>IntegerLiteral(5)</code>
<code>IntegerLiteral(4)</code>
<code>IntegerLiteral(3)</code>
<code>IntegerLiteral(-)</code>
<code>IntegerLiteral(+)</code>
<code>STOP</code>

When reading this representation, Clang evaluates each expression record it encounters, builds the appropriate abstract syntax tree node, and then pushes that expression on to a stack. When a record contains N subexpressions — `BinaryOperator` has two of them — those expressions are popped from the top of the stack. The special `STOP` code indicates that we have reached the end of a serialized expression or statement; other expression or statement records may follow, but they are part of a different expression.

Identifier Table Block

The identifier table block contains an on-disk hash table that maps each identifier mentioned within the AST file to the serialized representation of the identifier's information (e.g, the `IdentifierInfo` structure). The serialized representation contains:

- The actual identifier string.
- Flags that describe whether this identifier is the name of a built-in, a poisoned identifier, an extension token, or a macro.
- If the identifier names a macro, the offset of the macro definition within the [Preprocessor Block](#).
- If the identifier names one or more declarations visible from translation unit scope, the [declaration IDs](#) of these declarations.

When an AST file is loaded, the AST file reader mechanism introduces itself into the identifier table as an external lookup source. Thus, when the user program refers to an identifier that has not yet been seen, Clang will perform a lookup into the identifier table. If an identifier is found, its contents (macro definitions, flags, top-level declarations, etc.) will be deserialized, at which point the corresponding `IdentifierInfo` structure will have the same contents it would have after parsing the headers in the AST file.

Within the AST file, the identifiers used to name declarations are represented with an integral value. A separate table provides a mapping from this integral value (the identifier ID) to the location within the on-disk hash table where that identifier is stored. This mapping is used when deserializing the name of a declaration, the identifier of a token, or any other construct in the AST file that refers to a name.

Method Pool Block

The method pool block is represented as an on-disk hash table that serves two purposes: it provides a mapping from the names of Objective-C selectors to the set of Objective-C instance and class methods that have that particular selector (which is required for semantic analysis in Objective-C) and also stores all of the selectors used by entities within the AST file. The design of the method pool is similar to that of the [identifier table](#): the first time a particular selector is formed during the compilation of the program, Clang will search in the on-disk hash table of selectors; if found, Clang will read the Objective-C methods associated with that selector into the appropriate front-end data structure (`Sema::InstanceMethodPool` and `Sema::FactoryMethodPool` for instance and class methods, respectively).

As with identifiers, selectors are represented by numeric values within the AST file. A separate index maps these numeric selector values to the offset of the selector within the on-disk hash table, and will be used when de-serializing an Objective-C method declaration (or other Objective-C construct) that refers to the selector.

AST Reader Integration Points

The “lazy” deserialization behavior of AST files requires their integration into several completely different submodules of Clang. For example, lazily deserializing the declarations during name lookup requires that the name-lookup routines be able to query the AST file to find entities stored there.

For each Clang data structure that requires direct interaction with the AST reader logic, there is an abstract class that provides the interface between the two modules. The `ASTReader` class, which handles the loading of an AST file, inherits from all of these abstract classes to provide lazy deserialization of Clang’s data structures. `ASTReader` implements the following abstract classes:

ExternalLocEntrySource This abstract interface is associated with the `SourceManager` class, and is used whenever the *source manager* needs to load the details of a file, buffer, or macro instantiation.

IdentifierInfoLookup This abstract interface is associated with the `IdentifierTable` class, and is used whenever the program source refers to an identifier that has not yet been seen. In this case, the AST reader searches for this identifier within its *identifier table* to load any top-level declarations or macros associated with that identifier.

ExternalASTSource This abstract interface is associated with the `ASTContext` class, and is used whenever the abstract syntax tree nodes need to be loaded from the AST file. It provides the ability to de-serialize declarations and types identified by their numeric values, read the bodies of functions when required, and read the declarations stored within a declaration context (either for iteration or for name lookup).

ExternalSemaSource This abstract interface is associated with the `Sema` class, and is used whenever semantic analysis needs to read information from the *global method pool*.

Chained precompiled headers

Chained precompiled headers were initially intended to improve the performance of IDE-centric operations such as syntax highlighting and code completion while a particular source file is being edited by the user. To minimize the amount of reparsing required after a change to the file, a form of precompiled header — called a precompiled *preamble* — is automatically generated by parsing all of the headers in the source file, up to and including the last `#include`. When only the source file changes (and none of the headers it depends on), reparsing of that source file can use the precompiled preamble and start parsing after the `#includes`, so parsing time is proportional to the size of the source file (rather than all of its includes). However, the compilation of that translation unit may already use a precompiled header: in this case, Clang will create the precompiled preamble as a chained precompiled header that refers to the original precompiled header. This drastically reduces the time needed to serialize the precompiled preamble for use in reparsing.

Chained precompiled headers get their name because each precompiled header can depend on one other precompiled header, forming a chain of dependencies. A translation unit will then include the precompiled header that starts the chain (i.e., nothing depends on it). This linearity of dependencies is important for the semantic model of chained precompiled headers, because the most-recent precompiled header can provide information that overrides the information provided by the precompiled headers it depends on, just like a header file `B.h` that includes another header `A.h` can modify the state produced by parsing `A.h`, e.g., by `#undef`’ing a macro defined in `A.h`.

There are several ways in which chained precompiled headers generalize the AST file model:

Numbering of IDs Many different kinds of entities — identifiers, declarations, types, etc. — have ID numbers that start at 1 or some other predefined constant and grow upward. Each precompiled header records the maximum ID number it has assigned in each category. Then, when a new precompiled header is generated that depends

on (chains to) another precompiled header, it will start counting at the next available ID number. This way, one can determine, given an ID number, which AST file actually contains the entity.

Name lookup When writing a chained precompiled header, Clang attempts to write only information that has changed from the precompiled header on which it is based. This changes the lookup algorithm for the various tables, such as the *identifier table*: the search starts at the most-recent precompiled header. If no entry is found, lookup then proceeds to the identifier table in the precompiled header it depends on, and so on. Once a lookup succeeds, that result is considered definitive, overriding any results from earlier precompiled headers.

Update records There are various ways in which a later precompiled header can modify the entities described in an earlier precompiled header. For example, later precompiled headers can add entries into the various name-lookup tables for the translation unit or namespaces, or add new categories to an Objective-C class. Each of these updates is captured in an “update record” that is stored in the chained precompiled header file and will be loaded along with the original entity.

Modules

Modules generalize the chained precompiled header model yet further, from a linear chain of precompiled headers to an arbitrary directed acyclic graph (DAG) of AST files. All of the same techniques used to make chained precompiled headers work — ID number, name lookup, update records — are shared with modules. However, the DAG nature of modules introduce a number of additional complications to the model:

Numbering of IDs The simple, linear numbering scheme used in chained precompiled headers falls apart with the module DAG, because different modules may end up with different numbering schemes for entities they imported from common shared modules. To account for this, each module file provides information about which modules it depends on and which ID numbers it assigned to the entities in those modules, as well as which ID numbers it took for its own new entities. The AST reader then maps these “local” ID numbers into a “global” ID number space for the current translation unit, providing a 1-1 mapping between entities (in whatever AST file they inhabit) and global ID numbers. If that translation unit is then serialized into an AST file, this mapping will be stored for use when the AST file is imported.

Declaration merging It is possible for a given entity (from the language’s perspective) to be declared multiple times in different places. For example, two different headers can have the declaration of `printf` or could forward-declare `struct stat`. If each of those headers is included in a module, and some third party imports both of those modules, there is a potentially serious problem: name lookup for `printf` or `struct stat` will find both declarations, but the AST nodes are unrelated. This would result in a compilation error, due to an ambiguity in name lookup. Therefore, the AST reader performs declaration merging according to the appropriate language semantics, ensuring that the two disjoint declarations are merged into a single redeclaration chain (with a common canonical declaration), so that it is as if one of the headers had been included before the other.

Name Visibility Modules allow certain names that occur during module creation to be “hidden”, so that they are not part of the public interface of the module and are not visible to its clients. The AST reader maintains a “visible” bit on various AST nodes (declarations, macros, etc.) to indicate whether that particular AST node is currently visible; the various name lookup mechanisms in Clang inspect the visible bit to determine whether that entity, which is still in the AST (because other, visible AST nodes may depend on it), can actually be found by name lookup. When a new (sub)module is imported, it may make existing, non-visible, already-deserialized AST nodes visible; it is the responsibility of the AST reader to find and update these AST nodes when it is notified of the import.

CHAPTER 8

Indices and tables

- `genindex`
- `modindex`
- `search`

Symbols

- help
 - command line option, 232
- D<macroname>=<value>
 - command line option, 233
- E
 - command line option, 229
- F<directory>
 - command line option, 234
- I<directory>
 - command line option, 233
- MV
 - command line option, 20
- O0, -O1, -O2, -O3, -Ofast, -Os, -Oz, -O, -O4
 - command line option, 231
- ObjC, -ObjC++
 - command line option, 230
- Qunused-arguments
 - command line option, 232
- S
 - command line option, 229
- U<macroname>
 - command line option, 233
- Wa,<args>
 - command line option, 232
- Wambiguous-member-template
 - command line option, 18
- Wbind-to-temporary-copy
 - command line option, 18
- Wdocumentation
 - command line option, 32
- Werror
 - command line option, 14
- Weverything
 - command line option, 14
- Wextra-tokens
 - command line option, 18
- Wfoo
 - command line option, 14
- Wl,<args>
 - command line option, 232
- Wno-documentation-unknown-command
 - command line option, 32
- Wno-error=foo
 - command line option, 14
- Wno-foo
 - command line option, 14
- Wp,<args>
 - command line option, 232
- Wsystem-headers
 - command line option, 14
- Xanalyzer <arg>
 - command line option, 232
- Xassembler <arg>
 - command line option, 233
- Xlinker <arg>
 - command line option, 233
- Xpreprocessor <arg>
 - command line option, 233
- ###
 - command line option, 232
- ansi
 - command line option, 230
- arch <architecture>
 - command line option, 231
- c
 - command line option, 229
- fblocks
 - command line option, 230
- fborland-extensions
 - command line option, 230
- fbracket-depth=N
 - command line option, 35
- fcomment-block-commands=[commands]
 - command line option, 32
- fcommon
 - command line option, 232
- fconstexpr-depth=N
 - command line option, 35

- fdiagnostics-format=clang/msvc/vi
command line option, [15](#)
- fdiagnostics-parseable-fixits
command line option, [17](#)
- fdiagnostics-show-category=none/id/name
command line option, [16](#)
- fdiagnostics-show-template-tree
command line option, [17](#)
- ferror-limit=123
command line option, [14](#)
- fexceptions
command line option, [232](#)
- ffreestanding
command line option, [230](#)
- flax-vector-conversions
command line option, [230](#)
- flto, -emit-llvm
command line option, [232](#)
- fmath-errno
command line option, [230](#)
- fms-extensions
command line option, [230](#)
- fmsc-version=
command line option, [230](#)
- fno-assume-sane-operator-new
command line option, [27](#)
- fno-builtin
command line option, [230](#)
- fno-crash-diagnostics
command line option, [19](#)
- fno-elide-type
command line option, [17](#)
- fno-standalone-debug
command line option, [32](#)
- fobjc-abi-version=version
command line option, [230](#)
- fobjc-gc
command line option, [230](#)
- fobjc-gc-only
command line option, [230](#)
- fobjc-nonfragile-abi
command line option, [231](#)
- fobjc-nonfragile-abi-version=<version>
command line option, [231](#)
- foperator-arrow-depth=N
command line option, [35](#)
- fparse-all-comments
command line option, [32](#)
- fpascal-strings
command line option, [230](#)
- fprofile-generate[=<dirname>]
command line option, [31](#)
- fprofile-use[=<pathname>]
command line option, [31](#)
- fsanitize-undefined-trap-on-error
command line option, [27](#)
- fshow-column, -fshow-source-location, -fcaret-
diagnostics, -fdiagnostics-fixit-info, -
fdiagnostics-parseable-fixits, -fdiagnostics-
print-source-range-info, -fprint-source-range-
info, -fdiagnostics-show-option, -fmessage-
length
command line option, [233](#)
- fstandalone-debug
command line option, [32](#)
- fstandalone-debug -fno-standalone-debug
command line option, [232](#)
- fsyntax-only
command line option, [229](#)
- ftemplate-backtrace-limit=123
command line option, [14](#)
- ftemplate-depth=N
command line option, [35](#)
- ftime-report
command line option, [233](#)
- ftls-model=<model>
command line option, [232](#)
- ftls-model=[model]
command line option, [27](#)
- ftrap-function=[name]
command line option, [27](#)
- ftrapv
command line option, [232](#)
- fvisibility
command line option, [232](#)
- fwritable-strings
command line option, [230](#)
- g
command line option, [32](#), [231](#)
- g0
command line option, [32](#)
- gline-tables-only
command line option, [32](#)
- include <filename>
command line option, [233](#)
- integrated-as, -no-integrated-as
command line option, [233](#)
- march=<cpu>
command line option, [231](#)
- mgeneral-regs-only
command line option, [27](#)
- mhwdiv=[values]
command line option, [27](#)
- miphoneos-version-min
command line option, [231](#)
- mmacosx-version-min=<version>
command line option, [231](#)
- m[no-]crc

command line option, 27
 -nobuiltininc
 command line option, 234
 -nostdinc
 command line option, 234
 -nostdlibinc
 command line option, 234
 -o <file>
 command line option, 233
 -pedantic
 command line option, 14
 -pedantic-errors
 command line option, 14
 -print-file-name=<file>
 command line option, 233
 -print-libgcc-file-name
 command line option, 233
 -print-prog-name=<name>
 command line option, 233
 -print-search-dirs
 command line option, 233
 -save-temps
 command line option, 233
 -std=<language>
 command line option, 230
 -stdlib=<library>
 command line option, 230
 -time
 command line option, 233
 -trigraphs
 command line option, 230
 -v
 command line option, 233
 -w
 command line option, 14
 -x <language>
 command line option, 230

C

command line option
 -help, 232
 -D<macroname>=<value>, 233
 -E, 229
 -F<directory>, 234
 -I<directory>, 233
 -MV, 20
 -O0, -O1, -O2, -O3, -Ofast, -Os, -Oz, -O, -O4, 231
 -ObjC, -ObjC++, 230
 -Qunused-arguments, 232
 -S, 229
 -U<macroname>, 233
 -Wa,<args>, 232
 -Wambiguous-member-template, 18
 -Wbind-to-temporary-copy, 18

-Wdocumentation, 32
 -Werror, 14
 -Weverything, 14
 -Wextra-tokens, 18
 -Wfoo, 14
 -Wl,<args>, 232
 -Wno-documentation-unknown-command, 32
 -Wno-error=foo, 14
 -Wno-foo, 14
 -Wp,<args>, 232
 -Wsystem-headers, 14
 -Xanalyzer <arg>, 232
 -Xassembler <arg>, 233
 -Xlinker <arg>, 233
 -Xpreprocessor <arg>, 233
 -###, 232
 -ansi, 230
 -arch <architecture>, 231
 -c, 229
 -fblocks, 230
 -fborland-extensions, 230
 -fbracket-depth=N, 35
 -fcomment-block-commands=[commands], 32
 -fcommon, 232
 -fconstexpr-depth=N, 35
 -fdiagnostics-format=clang/msvc/vi, 15
 -fdiagnostics-parseable-fixits, 17
 -fdiagnostics-show-category=none/id/name, 16
 -fdiagnostics-show-template-tree, 17
 -ferror-limit=123, 14
 -fexceptions, 232
 -ffreestanding, 230
 -flax-vector-conversions, 230
 -flto, -emit-llvm, 232
 -fmath-errno, 230
 -fms-extensions, 230
 -fmsc-version=, 230
 -fno-assume-sane-operator-new, 27
 -fno-builtin, 230
 -fno-crash-diagnostics, 19
 -fno-elide-type, 17
 -fno-standalone-debug, 32
 -fobjc-abi-version=version, 230
 -fobjc-gc, 230
 -fobjc-gc-only, 230
 -fobjc-nonfragile-abi, 231
 -fobjc-nonfragile-abi-version=<version>, 231
 -foperator-arrow-depth=N, 35
 -fparse-all-comments, 32
 -fpascal-strings, 230
 -fprofile-generate[=<dirname>], 31
 -fprofile-use[=<pathname>], 31
 -fsanitize-undefined-trap-on-error, 27

-fshow-column, -fshow-source-location, -fcaret-diagnostics, -fdiagnostics-fixit-info, -fdiagnostics-parseable-fixits, -fdiagnostics-print-source-range-info, -fprint-source-range-info, -fdiagnostics-show-option, -fmessage-length, 233

-fstandalone-debug, 32

-fstandalone-debug -fno-standalone-debug, 232

-fsyntax-only, 229

-ftemplate-backtrace-limit=123, 14

-ftemplate-depth=N, 35

-ftime-report, 233

-ftls-model=<model>, 232

-ftls-model=[model], 27

-ftrap-function=[name], 27

-ftrapv, 232

-fvisibility, 232

-fwritable-strings, 230

-g, 32, 231

-g0, 32

-gline-tables-only, 32

-include <filename>, 233

-integrated-as, -no-integrated-as, 233

-march=<cpu>, 231

-mgeneral-regs-only, 27

-mhwdiv=[values], 27

-miphoneos-version-min, 231

-mmacosx-version-min=<version>, 231

-m[no-]crc, 27

-nobuiltininc, 234

-nostdinc, 234

-nostdlibinc, 234

-o <file>, 233

-pedantic, 14

-pedantic-errors, 14

-print-file-name=<file>, 233

-print-libgcc-file-name, 233

-print-prog-name=<name>, 233

-print-search-dirs, 233

-save-temps, 233

-std=<language>, 230

-stdlib=<library>, 230

-time, 233

-trigraphs, 230

-v, 233

-w, 14

-x <language>, 230

no stage selection option, 229

CPATH, 234

E

environment variable

C_INCLUDE_PATH, OBJC_INCLUDE_PATH, CPLUS_INCLUDE_PATH, OBJCPLUS_INCLUDE_PATH, 234

CPATH, 234

MACOSX_DEPLOYMENT_TARGET, 234

TMPDIR, TEMP, TMP, 234

N

no stage selection option

command line option, 229