Attributes in Clang

Introduction

This page lists the attributes currently supported by Clang.

Function Attributes

#pragma omp declare simd

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
					X	

The *declare simd* construct can be applied to a function to enable the creation of one or more versions that can process multiple arguments using SIMD instructions from a single invocation in a SIMD loop. The *declare simd* directive is a declarative directive. There may be multiple *declare simd* directives for a function. The use of a *declare simd* construct on a function enables the creation of SIMD versions of the associated function that can be used to process multiple arguments from a single invocation from a SIMD loop concurrently. The syntax of the *declare simd* construct is as follows:

```
#pragma omp declare simd [clause[[,] clause] ...] new-line [#pragma omp declare simd [clause[[,] clause] ...] new-line] [...] function definition or declaration
```

where clause is one of the following:

simdlen(length) linear(argument-list[:constant-linear-step]) aligned(argument-list[:alignment]) uniform(argument-list) inbranch notinbranch

#pragma omp declare target

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
					X	

The *declare target* directive specifies that variables and functions are mapped to a device for OpenMP offload mechanism.

The syntax of the declare target directive is as follows:

#pragma omp declare target new-line declarations-definition-seq #pragma omp end declare target new-line

_Noreturn

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
				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.

abi_tag (gnu::abi_tag)

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
X	X					X

The abi_tag attribute can be applied to a function, variable, class or inline namespace declaration to modify the mangled name of the entity. It gives the ability to distinguish between different versions of the same entity but with different ABI versions supported. For example, a newer version of a class could have a different set of data members and thus have a different size. Using the abi_tag attribute, it is possible to have different mangled names for a global variable of the class type. Therefor, the old code could keep using the old manged name and the new code will use the new mangled name with tags.

acquire_capability (acquire_shared_capability, clang::acquire_capability, clang::acquire_shared_capability)

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
X	X					

Marks a function as acquiring a capability.

alloc_align (gnu::alloc_align)

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
X	X					

Use __attribute__((alloc_align(<alignment>)) on a function declaration to specify that the return value of the function (which must be a pointer type) is at least as aligned as the value of the indicated parameter. The parameter is given by its index in the list of formal parameters; the first parameter has index 1 unless the function is a C++ non-static member function, in which case the first parameter has index 2 to account for the implicit this parameter.

```
// The returned pointer has the alignment specified by the first parameter.
void *a(size_t align) __attribute__((alloc_align(1)));
```

```
// The returned pointer has the alignment specified by the second parameter.
void *b(void *v, size_t align) __attribute__((alloc_align(2)));

// The returned pointer has the alignment specified by the second visible
// parameter, however it must be adjusted for the implicit 'this' parameter.
void *Foo::b(void *v, size_t align) __attribute__((alloc_align(3)));
```

Note that this attribute merely informs the compiler that a function always returns a sufficiently aligned pointer. It does not cause the compiler to emit code to enforce that alignment. The behavior is undefined if the returned poitner is not sufficiently aligned.

alloc_size (gnu::alloc_size)

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
X	X					X

The alloc_size attribute can be placed on functions that return pointers in order to hint to the compiler how many bytes of memory will be available at the returned poiner. alloc_size takes one or two arguments.

alloc_size(N) implies that argument number N equals the number of available bytes at the returned pointer.

alloc_size(N, M) implies that the product of argument number N and argument number M equals the number of available bytes at the returned pointer.

Argument numbers are 1-based.

An example of how to use alloc_size

```
void *my_malloc(int a) __attribute__((alloc_size(1)));
void *my_calloc(int a, int b) __attribute__((alloc_size(1, 2)));

int main() {
  void *const p = my_malloc(100);
  assert(__builtin_object_size(p, 0) == 100);
  void *const a = my_calloc(20, 5);
```

```
  assert(__builtin_object_size(a, 0) == 100);
}
```

Note

This attribute works differently in clang than it does in GCC. Specifically, clang will only trace `const` pointers (as above); we give up on pointers that are not marked as `const`. In the vast majority of cases, this is unimportant, because LLVM has support for the `alloc_size` attribute. However, this may cause mildly unintuitive behavior when used with other attributes, such as `enable_if`.

assert_capability (assert_shared_capability, clang::assert_capability, clang::assert_shared_capability)

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
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)

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
X	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

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
X						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(macos,introduced=10.4,deprecated=10.6,obsoleted=10.7)));
```

The availability attribute states that f was introduced in macOS 10.4, deprecated in macOS 10.6, and obsoleted in macOS 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 macOS 10.5, a call to f() succeeds. If Clang is instructed to compile code for macOS 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 macOS 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.

replacement=*string-literal*

Additional message text that Clang will use to provide Fix-It when emitting a warning about use of a deprecated declaration. The Fix-It will replace the deprecated declaration with the new declaration specified.

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.

macos

Apple's macOS operating system. The minimum deployment target is specified by the -mmacosx-version-min=*version* command-line argument. macosx is supported for backward-compatibility reasons, but it is deprecated.

tvos

Apple's tvOS operating system. The minimum deployment target is specified by the -mtvos-version-min=*version* command-line argument.

watchos

Apple's watchOS operating system. The minimum deployment target is specified by the -mwatchos-version-min=*version* command-line argument.

A declaration can typically 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.

The flag strict disallows using API when deploying back to a platform version prior to when the declaration was introduced. An attempt to use such API before its introduction causes a hard error. Weakly-linking is almost always a better API choice, since it allows users to query

availability at runtime.

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(macos,introduced=10.4)));
void g(void) __attribute__((availability(macos,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 macos and ios availability from above.
void g(void) __attribute__((availability(macos,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(macos,introduced=10.4)));
- (id)method2 __attribute__((availability(macos,introduced=10.4)));
@end

@interface B : A
- (id)method __attribute__((availability(macos,introduced=10.3))); // okay: method moved into base class later
- (id)method __attribute__((availability(macos,introduced=10.5))); // error: this method was available via the base class in 10.4
@end
```

Starting with the macOS 10.12 SDK, the API_AVAILABLE macro from <os/availability.h> can simplify the spelling:

```
@interface A
- (id)method API_AVAILABLE(macos(10.11));
- (id)otherMethod API_AVAILABLE(macos(10.11), ios(11.0));
@end
```

Also see the documentation for **@available**

carries_dependency

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
X	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.

convergent (clang::convergent)

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
X	X					X

The convergent attribute can be placed on a function declaration. It is translated into the LLVM convergent attribute, which indicates that the call instructions of a function with this attribute cannot be made control-dependent on any additional values.

In languages designed for SPMD/SIMT programming model, e.g. OpenCL or CUDA, the call instructions of a function with this attribute must be executed by all work items or threads in a work group or sub group.

This attribute is different from noduplicate because it allows duplicating function calls if it can be proved that the duplicated function calls are not made control-dependent on any additional values, e.g., unrolling a loop executed by all work items.

Sample usage: .. code-block:: c

```
void convfunc(void) __attribute__((convergent)); // Setting it as a C++11 attribute is also valid in a C++ program. // void convfunc(void) [[clang::convergent]];
```

deprecated (gnu::deprecated)

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
X	X	X	X			

The deprecated attribute can be applied to a function, a variable, or a type. This is useful when identifying functions, variables, or types that are expected to be removed in a future version of a program.

Consider the function declaration for a hypothetical function f:

```
void f(void) __attribute__((deprecated("message", "replacement")));
```

When spelled as *__attribute__((deprecated))*, the deprecated attribute can have two optional string arguments. The first one is the message to display when emitting the warning; the second one enables the compiler to provide a Fix-It to replace the deprecated name with a new name. Otherwise, when spelled as *[[gnu::deprecated]] or [[deprecated]]*, the attribute can have one optional string argument which is the message to display when emitting the warning.

diagnose_if

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
X						

The diagnose_if attribute can be placed on function declarations to emit warnings or errors at compile-time if calls to the attributed function meet certain user-defined criteria. For example:

```
void abs(int a)
  __attribute__((diagnose_if(a >= 0, "Redundant abs call", "warning")));
void must_abs(int a)
  __attribute__((diagnose_if(a >= 0, "Redundant abs call", "error")));

int val = abs(1); // warning: Redundant abs call
int val2 = must_abs(1); // error: Redundant abs call
```

```
int val3 = abs(val);
int val4 = must_abs(val); // Because run-time checks are not emitted for
                          // diagnose_if attributes, this executes without
                          // issue.
```

diagnose_if is closely related to enable_if, with a few key differences:

- Overload resolution is not aware of diagnose_if attributes: they're considered only after we select the best candidate from a given candidate set.
- Function declarations that differ only in their diagnose_if attributes are considered to be redeclarations of the same function (not overloads).
- If the condition provided to diagnose_if cannot be evaluated, no diagnostic will be emitted.

Otherwise, diagnose_if is essentially the logical negation of enable_if.

As a result of bullet number two, diagnose_if attributes will stack on the same function. For example:

```
int foo() __attribute__((diagnose_if(1, "diag1", "warning")));
int foo() __attribute__((diagnose_if(1, "diag2", "warning")));

int bar = foo(); // warning: diag1
                 // warning: diag2
int (*fooptr)(void) = foo; // warning: diag1
                           // warning: diag2

constexpr int supportsAPILevel(int N) { return N < 5; }
int baz(int a)
  __attribute__((diagnose_if(!supportsAPILevel(10),
                             "Upgrade to API level 10 to use baz", "error")));
int baz(int a)
  __attribute__((diagnose_if(!a, "0 is not recommended.", "warning")));

int (*bazptr)(int) = baz; // error: Upgrade to API level 10 to use baz
int v = baz(0); // error: Upgrade to API level 10 to use baz
```

Query for this feature with __has_attribute(diagnose_if).

disable_tail_calls (clang::disable_tail_calls)

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
X	X					X

The disable_tail_calls attribute instructs the backend to not perform tail call optimization inside the marked function.

For example:

```
int callee(int);

int foo(int a) __attribute__((disable_tail_calls)) {
  return callee(a); // This call is not tail-call optimized.
}
```

Marking virtual functions as disable_tail_calls is legal.

```
int callee(int);

class Base {
public:
  [[clang::disable_tail_calls]] virtual int foo1() {
    return callee(); // This call is not tail-call optimized.
  }
};

class Derived1 : public Base {
public:
  int foo1() override {
    return callee(); // This call is tail-call optimized.
  }
};
```

enable_if

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
X						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 strnlen(strbuf, maxlen) to resolve to strnlen_chk(strbuf, maxlen, size of strbuf) only if the size of strbuf can be determined:

```
__attribute__((always_inline))
static inline size_t strnlen(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 strnlen_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)`.

Note that functions with one or more `enable_if` attributes may not have their address taken, unless all of the conditions specified by said `enable_if` are constants that evaluate to `true`. For example:

```
const int TrueConstant = 1;
const int FalseConstant = 0;
int f(int a) __attribute__((enable_if(a > 0, "")));
int g(int a) __attribute__((enable_if(a == 0 || a != 0, "")));
int h(int a) __attribute__((enable_if(1, "")));
int i(int a) __attribute__((enable_if(TrueConstant, "")));
```

```
int j(int a) __attribute__((enable_if(FalseConstant, "")));

void fn() {
  int (*ptr)(int);
  ptr = &f; // error: 'a > 0' is not always true
  ptr = &g; // error: 'a == 0 || a != 0' is not a truthy constant
  ptr = &h; // OK: 1 is a truthy constant
  ptr = &i; // OK: 'TrueConstant' is a truthy constant
  ptr = &j; // error: 'FalseConstant' is a constant, but not truthy
}
```

Because enable_if evaluation happens during overload resolution, enable_if may give unintuitive results when used with templates, depending on when overloads are resolved. In the example below, clang will emit a diagnostic about no viable overloads for foo in bar, but not in baz:

```
double foo(int i) __attribute__((enable_if(i > 0, "")));
void *foo(int i) __attribute__((enable_if(i <= 0, "")));
template <int I>
auto bar() { return foo(I); }

template <typename T>
auto baz() { return foo(T::number); }

struct WithNumber { constexpr static int number = 1; };
void callThem() {
  bar<sizeof(WithNumber)>();
  baz<WithNumber>();
}
```

This is because, in bar, foo is resolved prior to template instantiation, so the value for I isn't known (thus, both enable_if conditions for foo fail). However, in baz, foo is resolved during template instantiation, so the value for T::number is known.

external_source_symbol (clang::external_source_symbol)

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute

X	X	X

The external_source_symbol attribute specifies that a declaration originates from an external source and describes the nature of that source.

The fact that Clang is capable of recognizing declarations that were defined externally can be used to provide better tooling support for mixed-language projects or projects that rely on auto-generated code. For instance, an IDE that uses Clang and that supports mixed-language projects can use this attribute to provide a correct 'jump-to-definition' feature. For a concrete example, consider a protocol that's defined in a Swift file:

```
@objc public protocol SwiftProtocol {
  func method()
}
```

This protocol can be used from Objective-C code by including a header file that was generated by the Swift compiler. The declarations in that header can use the external_source_symbol attribute to make Clang aware of the fact that SwiftProtocol actually originates from a Swift module:

```
__attribute__((external_source_symbol(language="Swift",defined_in="module")))
@protocol SwiftProtocol
@required
- (void) method;
@end
```

Consequently, when 'jump-to-definition' is performed at a location that references SwiftProtocol, the IDE can jump to the original definition in the Swift source file rather than jumping to the Objective-C declaration in the auto-generated header file.

The external_source_symbol attribute is a comma-separated list that includes clauses that describe the origin and the nature of the particular declaration. Those clauses can be:

language=*string-literal*
: The name of the source language in which this declaration was defined.

defined_in=*string-literal*
: The name of the source container in which the declaration was defined. The exact definition of source container is language-specific, e.g. Swift's source containers are modules, so defined_in should specify the Swift module name.

generated_declaration
: This declaration was automatically generated by some tool.

The clauses can be specified in any order. The clauses that are listed above are all optional, but the attribute has to have at least one clause.

flatten (gnu::flatten)

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
X	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.

force_align_arg_pointer (gnu::force_align_arg_pointer)

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
X	X					

Use this attribute to force stack alignment.

Legacy x86 code uses 4-byte stack alignment. Newer aligned SSE instructions (like ‘movaps’) that work with the stack require operands to be 16-byte aligned. This attribute realigns the stack in the function prologue to make sure the stack can be used with SSE instructions.

Note that the x86_64 ABI forces 16-byte stack alignment at the call site. Because of this, ‘force_align_arg_pointer’ is not needed on x86_64, except in rare cases where the caller does not align the stack properly (e.g. flow jumps from i386 arch code).

```
__attribute__ ((force_align_arg_pointer))
void f () {
  ...
}
```

format (gnu::format)

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
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.

ifunc (gnu::ifunc)

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
X	X					X

`__attribute__((ifunc("resolver")))` is used to mark that the address of a declaration should be resolved at runtime by calling a resolver function.

The symbol name of the resolver function is given in quotes. A function with this name (after mangling) must be defined in the current translation unit; it may be `static`. The resolver function should take no arguments and return a pointer.

The `ifunc` attribute may only be used on a function declaration. A function declaration with an `ifunc` attribute is considered to be a definition of the declared entity. The entity must not have weak linkage; for example, in C++, it cannot be applied to a declaration if a definition at that location would be considered inline.

Not all targets support this attribute. ELF targets support this attribute when using binutils v2.20.1 or higher and glibc v2.11.1 or higher. Non-ELF targets currently do not support this attribute.

internal_linkage (clang::internal_linkage)

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
X	X					X

The `internal_linkage` attribute changes the linkage type of the declaration to internal. This is similar to C-style `static`, but can be used on classes and class methods. When applied to a class definition, this attribute affects all methods and static data members of that class. This can be used to contain the ABI of a C++ library by excluding unwanted class methods from the export tables.

interrupt (ARM)

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
X	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.

interrupt (AVR)

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
X	X					X

Clang supports the GNU style __attribute__((interrupt)) attribute on AVR 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.

On the AVR, the hardware globally disables interrupts when an interrupt is executed. The first instruction of an interrupt handler declared with this attribute is a SEI instruction to re-enable interrupts. See also the signal attribute that does not insert a SEI instruction.

interrupt (MIPS)

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
X	X					X

Clang supports the GNU style __attribute__((interrupt("ARGUMENT"))) attribute on MIPS 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.

By default, the compiler will produce a function prologue and epilogue suitable for an interrupt service routine that handles an External Interrupt Controller (eic) generated interrupt. This behaviour can be explicitly requested with the “eic” argument.

Otherwise, for use with vectored interrupt mode, the argument passed should be of the form “vector=LEVEL” where LEVEL is one of the following values: “sw0”, “sw1”, “hw0”, “hw1”, “hw2”, “hw3”, “hw4”, “hw5”. The compiler will then set the interrupt mask to the corresponding level which will mask all interrupts up to and including the argument.

The semantics are as follows:

- The prologue is modified so that the Exception Program Counter (EPC) and Status coprocessor registers are saved to the stack. The interrupt mask is set so that the function can only be interrupted by a higher priority interrupt. The epilogue will restore the previous values of EPC and Status.
- The prologue and epilogue are modified to save and restore all non-kernel registers as necessary.
- The FPU is disabled in the prologue, as the floating pointer registers are not spilled to the stack.
- The function return sequence is changed to use an exception return instruction.

The parameter sets the interrupt mask for the function corresponding to the interrupt level specified. If no mask is specified the interrupt mask defaults to “eic”.

kernel

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
X						X

__attribute__((kernel)) is used to mark a kernel function in RenderScript.

In RenderScript, kernel functions are used to express data-parallel computations. The RenderScript runtime efficiently parallelizes kernel functions to run on computational resources such as multi-core CPUs and GPUs. See the **RenderScript** documentation for more information.

long_call (gnu::long_call, gnu::far)

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
X	X					X

Clang supports the __attribute__((long_call)), __attribute__((far)), and __attribute__((near)) attributes on MIPS targets. These attributes may only be added to function declarations and change the code generated by the compiler when directly calling the function. The near attribute allows calls to the function to be made using the jal instruction, which requires the function to be located in the same naturally aligned 256MB segment as the caller. The long_call and far attributes are synonyms and require the use of a different call sequence that works regardless of the distance between the functions.

These attributes have no effect for position-independent code.

These attributes take priority over command line switches such as -mlong-calls and -mno-long-calls.

micromips (gnu::micromips)

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
X	X					X

Clang supports the GNU style __attribute__((micromips)) and __attribute__((nomicromips)) attributes on MIPS targets. These attributes may be attached to a function definition and instructs the backend to generate or not to generate microMIPS code for that function.

These attributes override the *-mmicromips* and *-mno-micromips* options on the command line.

no_caller_saved_registers (gnu::no_caller_saved_registers)

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
X	X					

Use this attribute to indicate that the specified function has no caller-saved registers. That is, all registers are callee-saved except for registers used for passing parameters to the function or returning parameters from the function. The compiler saves and restores any modified registers that were not used for passing or returning arguments to the function.

The user can call functions specified with the 'no_caller_saved_registers' attribute from an interrupt handler without saving and restoring all call-clobbered registers.

Note that 'no_caller_saved_registers' attribute is not a calling convention. In fact, it only overrides the decision of which registers should be saved by the caller, but not how the parameters are passed from the caller to the callee.

For example:

```
__attribute__ ((no_caller_saved_registers, fastcall))
void f (int arg1, int arg2) {
  ...
}
```

In this case parameters ‘arg1’ and ‘arg2’ will be passed in registers. In this case, on 32-bit x86 targets, the function ‘f’ will use ECX and EDX as register parameters. However, it will not assume any scratch registers and should save and restore any modified registers except for ECX and EDX.

no_sanitize (clang::no_sanitize)

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
X	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)

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
X	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_memory

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
X	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_sanitize_thread

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
X	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_split_stack (gnu::no_split_stack)

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
X	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.

noalias

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
			X			

The noalias attribute indicates that the only memory accesses inside function are loads and stores from objects pointed to by its pointer-typed arguments, with arbitrary offsets.

nodiscard, warn_unused_result, clang::warn_unused_result, gnu::warn_unused_result

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
X	X	X				X

Clang supports the ability to diagnose when the results of a function call expression are discarded under suspicious circumstances. A diagnostic is generated when a function or its return type is marked with [[nodiscard]] (or __attribute__((warn_unused_result))) and the function call appears as a potentially-evaluated discarded-value expression that is not explicitly cast to *void*.

noduplicate (clang::noduplicate)

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
X	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.

nomicromips (gnu::nomicromips)

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
X	X					X

Clang supports the GNU style __attribute__((micromips)) and __attribute__((nomicromips)) attributes on MIPS targets. These attributes may be attached to a function definition and instructs the backend to generate or not to generate microMIPS code for that function.

These attributes override the *-mmicromips* and *-mno-micromips* options on the command line.

noreturn

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
	X					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.

not_tail_called (clang::not_tail_called)

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
X	X					X

The not_tail_called attribute prevents tail-call optimization on statically bound calls. It has no effect on indirect calls. Virtual functions, objective-c methods, and functions marked as always_inline cannot be marked as not_tail_called.

For example, it prevents tail-call optimization in the following case:

```
int __attribute__((not_tail_called)) foo1(int);

int foo2(int a) {
  return foo1(a); // No tail-call optimization on direct calls.
}
```

However, it doesn't prevent tail-call optimization in this case:

```
int __attribute__((not_tail_called)) foo1(int);

int foo2(int a) {
  int (*fn)(int) = &foo1;

  // not_tail_called has no effect on an indirect call even if the call can be
  // resolved at compile time.
  return (*fn)(a);
}
```

Marking virtual functions as not_tail_called is an error:

```
class Base {
public:
  // not_tail_called on a virtual function is an error.
  [[clang::not_tail_called]] virtual int foo1();
```

```
  virtual int foo2();

  // Non-virtual functions can be marked ``not_tail_called``.
  [[clang::not_tail_called]] int foo3();
};

class Derived1 : public Base {
public:
  int foo1() override;

  // not_tail_called on a virtual function is an error.
  [[clang::not_tail_called]] int foo2() override;
};
```

nothrow (gnu::nothrow)

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
X	X		X			X

Clang supports the GNU style __attribute__((nothrow)) and Microsoft style __declspec(nothrow) attribute as an equivilent of *noexcept* on function declarations. This attribute informs the compiler that the annotated function does not throw an exception. This prevents exception-unwinding. This attribute is particularly useful on functions in the C Standard Library that are guaranteed to not throw an exception.

objc_boxable

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
X						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

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
X						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

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
X						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

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
X						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
```

objc_runtime_visible

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
X						X

This attribute specifies that the Objective-C class to which it applies is visible to the Objective-C runtime but not to the linker. Classes annotated with this attribute cannot be subclassed and cannot have categories defined for them.

optnone (clang::optnone)

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
X	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

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
X						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.

If no viable candidates are otherwise available, we allow a conversion from a pointer of type `T*` to a pointer of type `U*`, where `T` and `U` are incompatible. This conversion is ranked below all other types of conversions. Please note: `U` lacking qualifiers that are present on `T` is sufficient for `T` and `U` to be incompatible.

The declaration of `overloadable` functions is restricted to function declarations and definitions. If a function is marked with the `overloadable` attribute, then all declarations and definitions of functions with that name, except for at most one (see the note below about unmarked overloads), must have the `overloadable` attribute. In addition, redeclarations of a function with the `overloadable` attribute must have the `overloadable` attribute, and redeclarations of a function without the `overloadable` attribute must *not* have the `overloadable` attribute. e.g.,

```
int f(int) __attribute__((overloadable));
float f(float); // error: declaration of "f" must have the "overloadable" attribute
int f(int); // error: redeclaration 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

int h(int);
int h(int) __attribute__((overloadable)); // error: declaration of "h" must not
                                          // have the "overloadable" 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 a 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
```

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, it's name *will* be mangled in the same way as it would in C.

For the purpose of backwards compatibility, at most one function with the same name as other overloadable functions may omit the overloadable attribute. In this case, the function without the overloadable attribute will not have its name mangled.

For example:

```
// Notes with mangled names assume Itanium mangling.
int f(int);
int f(double) __attribute__((overloadable));
void foo() {
  f(5); // Emits a call to f (not _Z1fi, as it would with an overload that
        // was marked with overloadable).
  f(1.0); // Emits a call to _Z1fd.
}
```

Support for unmarked overloads is not present in some versions of clang. You may query for it using __has_extension(overloadable_unmarked).

Query for this attribute with __has_attribute(overloadable).

release_capability (release_shared_capability, clang::release_capability, clang::release_shared_capability)

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
X	X					

Marks a function as releasing a capability.

short_call (gnu::short_call, gnu::near)

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
X	X					X

Clang supports the __attribute__((long_call)), __attribute__((far)), __attribute__((short_call)), and __attribute__((near)) attributes on MIPS targets. These attributes may only be added to function declarations and change the code generated by the compiler when directly calling the function. The short_call and near attributes are synonyms and allow calls to the function to be made using the jal instruction, which requires the function to be located in the same naturally aligned 256MB segment as the caller. The long_call and far attributes are synonyms and require the use of a different call sequence that works regardless of the distance between the functions.

These attributes have no effect for position-independent code.

These attributes take priority over command line switches such as -mlong-calls and -mno-long-calls.

signal (gnu::signal)

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
X	X					X

Clang supports the GNU style __attribute__((signal)) attribute on AVR 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.

Interrupt handler functions defined with the signal attribute do not re-enable interrupts.

target (gnu::target)

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
X	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)

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
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).

xray_always_instrument (clang::xray_always_instrument), xray_never_instrument (clang::xray_never_instrument), xray_log_args (clang::xray_log_args)

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
X	X					X

__attribute__((xray_always_instrument)) or [[clang::xray_always_instrument]] is used to mark member functions (in C++), methods (in Objective C), and free functions (in C, C++, and Objective C) to be instrumented with XRay. This will cause the function to always have space at the beginning and exit points to allow for runtime patching.

Conversely, __attribute__((xray_never_instrument)) or [[clang::xray_never_instrument]] will inhibit the insertion of these instrumentation points.

If a function has neither of these attributes, they become subject to the XRay heuristics used to determine whether a function should be instrumented or otherwise.

__attribute__((xray_log_args(N))) or [[clang::xray_log_args(N)]] is used to preserve N function arguments for the logging function. Currently, only N==1 is supported.

xray_always_instrument (clang::xray_always_instrument), xray_never_instrument (clang::xray_never_instrument), xray_log_args (clang::xray_log_args)

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
X	X					X

__attribute__((xray_always_instrument)) or [[clang::xray_always_instrument]] is used to mark member functions (in C++), methods (in Objective C), and free functions (in C, C++, and Objective C) to be instrumented with XRay. This will cause the function to always have space at the beginning and exit points to allow for runtime patching.

Conversely, __attribute__((xray_never_instrument)) or [[clang::xray_never_instrument]] will inhibit the insertion of these instrumentation points.

If a function has neither of these attributes, they become subject to the XRay heuristics used to determine whether a function should be instrumented or otherwise.

__attribute__((xray_log_args(N))) or [[clang::xray_log_args(N)]] is used to preserve N function arguments for the logging function. Currently, only N==1 is supported.

Variable Attributes

dllexport (gnu::dllexport)

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute

X	X	X	X

The __declspec(dllexport) attribute declares a variable, function, or Objective-C interface to be exported from the module. It is available under the -fdeclspec flag for compatibility with various compilers. The primary use is for COFF object files which explicitly specify what interfaces are available for external use. See the **dllexport** documentation on MSDN for more information.

dllimport (gnu::dllimport)

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
X	X		X			X

The __declspec(dllimport) attribute declares a variable, function, or Objective-C interface to be imported from an external module. It is available under the -fdeclspec flag for compatibility with various compilers. The primary use is for COFF object files which explicitly specify what interfaces are imported from external modules. See the **dllimport** documentation on MSDN for more information.

init_seg

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
					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.

maybe_unused, unused, gnu::unused

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
X	X	X				

When passing the -Wunused flag to Clang, entities that are unused by the program may be diagnosed. The [[maybe_unused]] (or __attribute__((unused))) attribute can be used to silence such diagnostics when the entity cannot be removed. For instance, a local variable may exist solely for use in an assert() statement, which makes the local variable unused when NDEBUG is defined.

The attribute may be applied to the declaration of a class, a typedef, a variable, a function or method, a function parameter, an enumeration, an enumerator, a non-static data member, or a label.

nodebug (gnu::nodebug)

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
X	X					X

The nodebug attribute allows you to suppress debugging information for a function or method, or for a variable that is not a parameter or a non-static data member.

noescape (clang::noescape)

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
X	X					X

noescape placed on a function parameter of a pointer type is used to inform the compiler that the pointer cannot escape: that is, no reference to the object the pointer points to that is derived from the parameter value will survive after the function returns. Users are responsible for making sure parameters annotated with noescape do not actuallly escape.

For example:

```
int *gp;

void nonescapingFunc(__attribute__((noescape)) int *p) {
  *p += 100; // OK.
}

void escapingFunc(__attribute__((noescape)) int *p) {
  gp = p; // Not OK.
}
```

Additionally, when the parameter is a *block pointer <https://clang.llvm.org/docs/BlockLanguageSpec.html>*, the same restriction applies to copies of the block. For example:

```
typedef void (^BlockTy)();
BlockTy g0, g1;

void nonescapingFunc(__attribute__((noescape)) BlockTy block) {
  block(); // OK.
}

void escapingFunc(__attribute__((noescape)) BlockTy block) {
  g0 = block; // Not OK.
  g1 = Block_copy(block); // Not OK either.
}
```

nosvm

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
X						X

OpenCL 2.0 supports the optional __attribute__((nosvm)) qualifier for pointer variable. It informs the compiler that the pointer does not refer to a shared virtual memory region. See OpenCL v2.0 s6.7.2 for details.

Since it is not widely used and has been removed from OpenCL 2.1, it is ignored by Clang.

pass_object_size

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
X						X

Note

The mangling of functions with parameters that are annotated with pass_object_size is subject to change. You can get around this by using __asm__("foo") to explicitly name your functions, thus preserving your ABI; also, non-overloadable C functions with pass_object_size are not mangled.

The pass_object_size(Type) attribute can be placed on function parameters to instruct clang to call __builtin_object_size(param, Type) at each callsite of said function, and implicitly pass the result of this call in as an invisible argument of type size_t directly after the parameter annotated with pass_object_size. Clang will also replace any calls to __builtin_object_size(param, Type) in the function by said implicit parameter.

Example usage:

```
int bzero1(char *const p __attribute__((pass_object_size(0))))
    __attribute__((noinline)) {
  int i = 0;
  for (/**/; i < (int)__builtin_object_size(p, 0); ++i) {
    p[i] = 0;
  }
  return i;
}

int main() {
  char chars[100];
  int n = bzero1(&chars[0]);
  assert(n == sizeof(chars));
```

```
  return 0;
}
```

If successfully evaluating __builtin_object_size(param, Type) at the callsite is not possible, then the "failed" value is passed in. So, using the definition of bzero1 from above, the following code would exit cleanly:

```
int main2(int argc, char *argv[]) {
  int n = bzero1(argv);
  assert(n == -1);
  return 0;
}
```

pass_object_size plays a part in overload resolution. If two overload candidates are otherwise equally good, then the overload with one or more parameters with pass_object_size is preferred. This implies that the choice between two identical overloads both with pass_object_size on one or more parameters will always be ambiguous; for this reason, having two such overloads is illegal. For example:

```
#define PS(N) __attribute__((pass_object_size(N)))
// OK
void Foo(char *a, char *b); // Overload A
// OK -- overload A has no parameters with pass_object_size.
void Foo(char *a PS(0), char *b PS(0)); // Overload B
// Error -- Same signature (sans pass_object_size) as overload B, and both
// overloads have one or more parameters with the pass_object_size attribute.
void Foo(void *a PS(0), void *b);

// OK
void Bar(void *a PS(0)); // Overload C
// OK
void Bar(char *c PS(1)); // Overload D

void main() {
  char known[10], *unknown;
  Foo(unknown, unknown); // Calls overload B
  Foo(known, unknown); // Calls overload B
  Foo(unknown, known); // Calls overload B
  Foo(known, known); // Calls overload B
```

```
  Bar(known); // Calls overload D
  Bar(unknown); // Calls overload D
}
```

Currently, pass_object_size is a bit restricted in terms of its usage:

- Only one use of pass_object_size is allowed per parameter.
- It is an error to take the address of a function with pass_object_size on any of its parameters. If you wish to do this, you can create an overload without pass_object_size on any parameters.
- It is an error to apply the pass_object_size attribute to parameters that are not pointers. Additionally, any parameter that pass_object_size is applied to must be marked const at its function's definition.

require_constant_initialization (clang::require_constant_initialization)

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
X	X					X

This attribute specifies that the variable to which it is attached is intended to have a **constant initializer** according to the rules of [basic.start.static]. The variable is required to have static or thread storage duration. If the initialization of the variable is not a constant initializer an error will be produced. This attribute may only be used in C++.

Note that in C++03 strict constant expression checking is not done. Instead the attribute reports if Clang can emit the variable as a constant, even if it's not technically a 'constant initializer'. This behavior is non-portable.

Static storage duration variables with constant initializers avoid hard-to-find bugs caused by the indeterminate order of dynamic initialization. They can also be safely used during dynamic initialization across translation units.

This attribute acts as a compile time assertion that the requirements for constant initialization have been met. Since these requirements change between dialects and have subtle pitfalls it's important to fail fast instead of silently falling back on dynamic initialization.

```
// -std=c++14
#define SAFE_STATIC [[clang::require_constant_initialization]]
struct T {
  constexpr T(int) {}
```

```
  ~T(); // non-trivial
};
SAFE_STATIC T x = {42}; // Initialization OK. Doesn't check destructor.
SAFE_STATIC T y = 42; // error: variable does not have a constant initializer
// copy initialization is not a constant expression on a non-literal type.
```

section (gnu::section, __declspec(allocate))

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
X	X		X			X

The section attribute allows you to specify a specific section a global variable or function should be in after translation.

swift_context

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
X						X

The swift_context attribute marks a parameter of a swiftcall function as having the special context-parameter ABI treatment.

This treatment generally passes the context value in a special register which is normally callee-preserved.

A swift_context parameter must either be the last parameter or must be followed by a swift_error_result parameter (which itself must always be the last parameter).

A context parameter must have pointer or reference type.

swift_error_result

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
X						X

The swift_error_result attribute marks a parameter of a swiftcall function as having the special error-result ABI treatment.

This treatment generally passes the underlying error value in and out of the function through a special register which is normally callee-preserved. This is modeled in C by pretending that the register is addressable memory:

> The caller appears to pass the address of a variable of pointer type. The current value of this variable is copied into the register before the call; if the call returns normally, the value is copied back into the variable.
> The callee appears to receive the address of a variable. This address is actually a hidden location in its own stack, initialized with the value of the register upon entry. When the function returns normally, the value in that hidden location is written back to the register.

A swift_error_result parameter must be the last parameter, and it must be preceded by a swift_context parameter.

A swift_error_result parameter must have type T** or T*& for some type T. Note that no qualifiers are permitted on the intermediate level.

It is undefined behavior if the caller does not pass a pointer or reference to a valid object.

The standard convention is that the error value itself (that is, the value stored in the apparent argument) will be null upon function entry, but this is not enforced by the ABI.

swift_indirect_result

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
X						X

The swift_indirect_result attribute marks a parameter of a swiftcall function as having the special indirect-result ABI treatment.

This treatment gives the parameter the target's normal indirect-result ABI treatment, which may involve passing it differently from an ordinary parameter. However, only the first indirect result will receive this treatment. Furthermore, low-level lowering may decide that a direct result must be returned indirectly; if so, this will take priority over the swift_indirect_result parameters.

A swift_indirect_result parameter must either be the first parameter or follow another swift_indirect_result parameter.

A swift_indirect_result parameter must have type T* or T& for some object type T. If T is a complete type at the point of definition of a function, it is undefined behavior if the argument value does not point to storage of adequate size and alignment for a value of type T.

Making indirect results explicit in the signature allows C functions to directly construct objects into them without relying on language optimizations like C++'s named return value optimization (NRVO).

swiftcall

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
X						

The swiftcall attribute indicates that a function should be called using the Swift calling convention for a function or function pointer.

The lowering for the Swift calling convention, as described by the Swift ABI documentation, occurs in multiple phases. The first, "high-level" phase breaks down the formal parameters and results into innately direct and indirect components, adds implicit paraameters for the generic signature, and assigns the context and error ABI treatments to parameters where applicable. The second phase breaks down the direct parameters and results from the first phase and assigns them to registers or the stack. The swiftcall convention only handles this second phase of lowering; the C function type must accurately reflect the results of the first phase, as follows:

- Results classified as indirect by high-level lowering should be represented as parameters with the swift_indirect_result attribute.
- Results classified as direct by high-level lowering should be represented as follows:
  - First, remove any empty direct results.
  - If there are no direct results, the C result type should be void.
  - If there is one direct result, the C result type should be a type with the exact layout of that result type.
  - If there are a multiple direct results, the C result type should be a struct type with the exact layout of a tuple of those results.
- Parameters classified as indirect by high-level lowering should be represented as parameters of pointer type.
- Parameters classified as direct by high-level lowering should be omitted if they are empty types; otherwise, they should be represented as a parameter type with a layout exactly matching the layout of the Swift parameter type.
- The context parameter, if present, should be represented as a trailing parameter with the swift_context attribute.
- The error result parameter, if present, should be represented as a trailing parameter (always following a context parameter) with the swift_error_result attribute.

swiftcall does not support variadic arguments or unprototyped functions.

The parameter ABI treatment attributes are aspects of the function type. A function type which which applies an ABI treatment attribute to a parameter is a different type from an otherwise-identical function type that does not. A single parameter may not have multiple ABI treatment attributes.

Support for this feature is target-dependent, although it should be supported on every target that Swift supports. Query for this support with __has_attribute(swiftcall). This implies support for the swift_context, swift_error_result, and swift_indirect_result attributes.

thread

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
			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.

tls_model (gnu::tls_model)

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
X	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.

Type Attributes

__single_inhertiance, __multiple_inheritance, __virtual_inheritance

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
				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 {};
```

align_value

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
X						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.

empty_bases

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
			X			

The empty_bases attribute permits the compiler to utilize the empty-base-optimization more frequently. This attribute only applies to struct, class, and union types. It is only supported when using the Microsoft C++ ABI.

enum_extensibility (clang::enum_extensibility)

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
X	X					X

Attribute enum_extensibility is used to distinguish between enum definitions that are extensible and those that are not. The attribute can take either closed or open as an argument. closed indicates a variable of the enum type takes a value that corresponds to one of the enumerators listed in the enum definition or, when the enum is annotated with flag_enum, a value that can be constructed using values corresponding to the enumerators. open indicates a variable of the enum type can take any values allowed by the standard and instructs clang to be more lenient when issuing warnings.

```
enum __attribute__((enum_extensibility(closed))) ClosedEnum {
  A0, A1
};

enum __attribute__((enum_extensibility(open))) OpenEnum {
  B0, B1
};

enum __attribute__((enum_extensibility(closed),flag_enum)) ClosedFlagEnum {
  C0 = 1 << 0, C1 = 1 << 1
};

enum __attribute__((enum_extensibility(open),flag_enum)) OpenFlagEnum {
  D0 = 1 << 0, D1 = 1 << 1
};

void foo1() {
  enum ClosedEnum ce;
  enum OpenEnum oe;
  enum ClosedFlagEnum cfe;
  enum OpenFlagEnum ofe;

  ce = A1;          // no warnings
  ce = 100;         // warning issued
```

```
  oe = B1;          // no warnings
  oe = 100;         // no warnings
  cfe = C0 | C1;    // no warnings
  cfe = C0 | C1 | 4; // warning issued
  ofe = D0 | D1;    // no warnings
  ofe = D0 | D1 | 4; // no warnings
}
```

flag_enum

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
X						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.

layout_version

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
			X			

The layout_version attribute requests that the compiler utilize the class layout rules of a particular compiler version. This attribute only applies to struct, class, and union types. It is only supported when using the Microsoft C++ ABI.

lto_visibility_public (clang::lto_visibility_public)

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute

	X					X

See **LTO Visibility**.

novtable

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
			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. It is only supported when using the Microsoft C++ ABI.

objc_subclassing_restricted

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
X						X

This attribute can be added to an Objective-C @interface declaration to ensure that this class cannot be subclassed.

selectany (gnu::selectany)

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
X	X		X			

This attribute appertains to a global symbol, causing it to have a weak definition (**linkonce**), allowing the linker to select any definition.

For more information see **gcc documentation** or **msvc documentation**.

transparent_union (gnu::transparent_union)

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
X	X					

This attribute can be applied to a union to change the behaviour of calls to functions that have an argument with a transparent union type. The compiler behaviour is changed in the following manner:

- A value whose type is any member of the transparent union can be passed as an argument without the need to cast that value.
- The argument is passed to the function using the calling convention of the first member of the transparent union. Consequently, all the members of the transparent union should have the same calling convention as its first member.

Transparent unions are not supported in C++.

Statement Attributes

#pragma clang loop

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
					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

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
					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 and attempt to partially unroll the loop if the trip count is not 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.

__attribute__((intel_reqd_sub_group_size))

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
X						X

The optional attribute intel_reqd_sub_group_size can be used to indicate that the kernel must be compiled and executed with the specified subgroup size. When this attribute is present, get_max_sub_group_size() is guaranteed to return the specified integer value. This is important for the correctness of many subgroup algorithms, and in some cases may be used by the compiler to generate more optimal code. See *cl_intel_required_subgroup_size <https://www.khronos.org/registry/OpenCL/extensions/intel/cl_intel_required_subgroup_size.txt>* for details.

__attribute__((opencl_unroll_hint))

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
X						

The opencl_unroll_hint attribute qualifier can be used to specify that a loop (for, while and do loops) can be unrolled. This attribute qualifier can be used to specify full unrolling or partial unrolling by a specified amount. This is a compiler hint and the compiler may ignore this directive. See **OpenCL v2.0** s6.11.5 for details.

__read_only, __write_only, __read_write (read_only, write_only, read_write)

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
				X		

The access qualifiers must be used with image object arguments or pipe arguments to declare if they are being read or written by a kernel or function.

The read_only/__read_only, write_only/__write_only and read_write/__read_write names are reserved for use as access qualifiers and shall not be used otherwise.

```
kernel void
foo (read_only image2d_t imageA,
    write_only image2d_t imageB) {
  ...
}
```

In the above example imageA is a read-only 2D image object, and imageB is a write-only 2D image object.

The read_write (or __read_write) qualifier can not be used with pipe.

More details can be found in the OpenCL C language Spec v2.0, Section 6.6.

fallthrough, clang::fallthrough

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
	X	X				

The fallthrough (or clang::fallthrough) attribute is used 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.

By default, Clang does not warn on unannotated fallthrough from one switch case to another. Diagnostics on fallthrough without a corresponding annotation can be enabled with the -Wimplicit-fallthrough argument.

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();
}
```

suppress (gsl::suppress)

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
	X					

The [[gsl::suppress]] attribute suppresses specific clang-tidy diagnostics for rules of the **C++ Core Guidelines** in a portable way. The attribute can be attached to declarations, statements, and at namespace scope.

```
[[gsl::suppress("Rh-public")]]
void f_() {
  int *p;
  [[gsl::suppress("type")]] {
    p = reinterpret_cast<int*>(7);
  }
}
namespace N {
  [[clang::suppress("type", "bounds")]];
  ...
}
```

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)

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
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)

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
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)

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
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”.

preserve_all

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
X						

On X86-64 and AArch64 targets, this attribute changes the calling convention of a function. The preserve_all calling convention attempts to make the code in the caller even less intrusive than the preserve_most calling convention. This calling convention also behaves identical to the C calling convention on how arguments and return values are passed, but it uses a different set of caller/callee-saved registers. This removes the burden of saving and recovering a large register set before and after the call in the caller. If the arguments are passed in callee-saved registers, then they will be preserved by the callee across the call. This doesn’t apply for values returned in callee-saved registers.

> On X86-64 the callee preserves all general purpose registers, except for R11. R11 can be used as a scratch register. Furthermore it also preserves all floating-point registers (XMMs/YMMs).

The idea behind this convention is to support calls to runtime functions that don't need to call out to any other functions.

This calling convention, like the preserve_most calling convention, will be used by a future version of the Objective-C runtime and should be considered experimental at this time.

preserve_most

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
X						

On X86-64 and AArch64 targets, this attribute changes the calling convention of a function. The preserve_most calling convention attempts to make the code in the caller as unintrusive as possible. This convention behaves identically to the C calling convention on how arguments and return values are passed, but it uses a different set of caller/callee-saved registers. This alleviates the burden of saving and recovering a large register set before and after the call in the caller. If the arguments are passed in callee-saved registers, then they will be preserved by the callee across the call. This doesn't apply for values returned in callee-saved registers.

> On X86-64 the callee preserves all general purpose registers, except for R11. R11 can be used as a scratch register. Floating-point registers (XMMs/YMMs) are not preserved and need to be saved by the caller.

The idea behind this convention is to support calls to runtime functions that have a hot path and a cold path. The hot path is usually a small piece of code that doesn't use many registers. The cold path might need to call out to another function and therefore only needs to preserve the caller-saved registers, which haven't already been saved by the caller. The *preserve_most* calling convention is very similar to the cold calling convention in terms of caller/callee-saved registers, but they are used for different types of function calls. coldcc is for function calls that are rarely executed, whereas *preserve_most* function calls are intended to be on the hot path and definitely executed a lot. Furthermore preserve_most doesn't prevent the inliner from inlining the function call.

This calling convention will be used by a future version of the Objective-C runtime and should therefore still be considered experimental at this time. Although this convention was created to optimize certain runtime calls to the Objective-C runtime, it is not limited to this runtime and might be used by other runtimes in the future too. The current implementation only supports X86-64 and AArch64, but the intention is to support more architectures in the future.

regcall (gnu::regcall, __regcall)

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
X	X			X		

On x86 targets, this attribute changes the calling convention to **__regcall** convention. This convention aims to pass as many arguments as possible in registers. It also tries to utilize registers for the return value whenever it is possible.

regparm (gnu::regparm)

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
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)

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
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)

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
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)

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
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.

AMD GPU Attributes

amdgpu_flat_work_group_size

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
X						X

The flat work-group size is the number of work-items in the work-group size specified when the kernel is dispatched. It is the product of the sizes of the x, y, and z dimension of the work-group.

Clang supports the __attribute__((amdgpu_flat_work_group_size(<min>, <max>))) attribute for the AMDGPU target. This attribute may be attached to a kernel function definition and is an optimization hint.

<min> parameter specifies the minimum flat work-group size, and <max> parameter specifies the maximum flat work-group size (must be greater than <min>) to which all dispatches of the kernel will conform. Passing 0, 0 as <min>, <max> implies the default behavior (128, 256).

If specified, the AMDGPU target backend might be able to produce better machine code for barriers and perform scratch promotion by estimating available group segment size.

An error will be given if:

Specified values violate subtarget specifications;
Specified values are not compatible with values provided through other attributes.

amdgpu_num_sgpr

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
X						X

Clang supports the __attribute__((amdgpu_num_sgpr(<num_sgpr>))) and __attribute__((amdgpu_num_vgpr(<num_vgpr>))) attributes for the AMDGPU target. These attributes may be attached to a kernel function definition and are an optimization hint.

If these attributes are specified, then the AMDGPU target backend will attempt to limit the number of SGPRs and/or VGPRs used to the specified value(s). The number of used SGPRs and/or VGPRs may further be rounded up to satisfy the allocation requirements or constraints of the subtarget. Passing 0 as num_sgpr and/or num_vgpr implies the default behavior (no limits).

These attributes can be used to test the AMDGPU target backend. It is recommended that the amdgpu_waves_per_eu attribute be used to control resources such as SGPRs and VGPRs since it is aware of the limits for different subtargets.

An error will be given if:

Specified values violate subtarget specifications;
Specified values are not compatible with values provided through other attributes;
The AMDGPU target backend is unable to create machine code that can meet the request.

amdgpu_num_vgpr

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
X						X

Clang supports the __attribute__((amdgpu_num_sgpr(<num_sgpr>))) and __attribute__((amdgpu_num_vgpr(<num_vgpr>))) attributes for the AMDGPU target. These attributes may be attached to a kernel function definition and are an optimization hint.

If these attributes are specified, then the AMDGPU target backend will attempt to limit the number of SGPRs and/or VGPRs used to the specified value(s). The number of used SGPRs and/or VGPRs may further be rounded up to satisfy the allocation requirements or constraints of the subtarget. Passing 0 as num_sgpr and/or num_vgpr implies the default behavior (no limits).

These attributes can be used to test the AMDGPU target backend. It is recommended that the amdgpu_waves_per_eu attribute be used to control resources such as SGPRs and VGPRs since it is aware of the limits for different subtargets.

An error will be given if:

Specified values violate subtarget specifications;
Specified values are not compatible with values provided through other attributes;
The AMDGPU target backend is unable to create machine code that can meet the request.

amdgpu_waves_per_eu

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
X						X

A compute unit (CU) is responsible for executing the wavefronts of a work-group. It is composed of one or more execution units (EU), which are responsible for executing the wavefronts. An EU can have enough resources to maintain the state of more than one executing wavefront. This allows an EU to hide latency by switching between wavefronts in a similar way to symmetric multithreading on a CPU. In order to allow the state for multiple wavefronts to fit on an EU, the resources used by a single wavefront have to be limited. For example, the number of SGPRs and VGPRs. Limiting such resources can allow greater latency hiding, but can result in having to spill some register state to memory.

Clang supports the __attribute__((amdgpu_waves_per_eu(<min>[, <max>]))) attribute for the AMDGPU target. This attribute may be attached to a kernel function definition and is an optimization hint.

<min> parameter specifies the requested minimum number of waves per EU, and *optional* <max> parameter specifies the requested maximum number of waves per EU (must be greater than <min> if specified). If <max> is omitted, then there is no restriction on the maximum number of waves per EU other than the one dictated by the hardware for which the kernel is compiled. Passing 0, 0 as <min>, <max> implies the default behavior (no limits).

If specified, this attribute allows an advanced developer to tune the number of wavefronts that are capable of fitting within the resources of an EU. The AMDGPU target backend can use this information to limit resources, such as number of SGPRs, number of VGPRs, size of available group and private memory segments, in such a way that guarantees that at least <min> wavefronts and at most <max> wavefronts are able to fit within the resources of an EU. Requesting more wavefronts can hide memory latency but limits available registers which can result in spilling. Requesting fewer wavefronts can help reduce cache thrashing, but can reduce memory latency hiding.

This attribute controls the machine code generated by the AMDGPU target backend to ensure it is capable of meeting the requested values. However, when the kernel is executed, there may be other reasons that prevent meeting the request, for example, there may be wavefronts from other kernels executing on the EU.

An error will be given if:

- Specified values violate subtarget specifications;
- Specified values are not compatible with values provided through other attributes;
- The AMDGPU target backend is unable to create machine code that can meet the request.

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

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
X						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

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
X						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

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
X						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

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
X						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

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
X						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_typestate

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute

X	X

Use __attribute__((test_typestate(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. To see warnings produced by these checks, ensure that -Wtype-safety is enabled. 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

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
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.

This attribute is primarily useful for checking arguments of variadic functions (pointer_with_type_tag can be used in most non-variadic cases).

In the attribute prototype above:

arg_kind is an identifier that should be used when annotating all applicable type tags.

arg_idx provides the position of a function argument. The expected type of this function argument will be determined by the function argument specified by type_tag_idx. In the code example below, “3” means that the type of the function’s third argument will be determined by type_tag_idx.

type_tag_idx provides the position of a function argument. This function argument will be a type tag. The type tag will determine the expected type of the argument specified by arg_idx. In the code example below, “2” means that the type tag associated with the function’s second argument should agree with the type of the argument specified by arg_idx.

For example:

```
int fcntl(int fd, int cmd, ...)
  __attribute__(( argument_with_type_tag(fcntl,3,2) ));
// The function's second argument will be a type tag; this type tag will
// determine the expected type of the function's third argument.
```

pointer_with_type_tag

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
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.

In the attribute prototype above:

ptr_kind is an identifier that should be used when annotating all applicable type tags.

ptr_idx provides the position of a function argument; this function argument will have a pointer type. The expected pointee type of this pointer type will be determined by the function argument specified by type_tag_idx. In the code example below, “1” means that the pointee type of the function’s first argument will be determined by type_tag_idx.

type_tag_idx provides the position of a function argument; this function argument will be a type tag. The type tag will determine the expected pointee type of the pointer argument specified by ptr_idx. In the code example below, “3” means that the type tag associated with the function’s third argument should agree with the pointee type of the pointer argument specified by ptr_idx.

For example:

```
typedef int MPI_Datatype;
int MPI_Send(void *buf, int count, MPI_Datatype datatype /*, other args omitted */)
    __attribute__(( pointer_with_type_tag(mpi,1,3) ));
// The function's 3rd argument will be a type tag; this type tag will
// determine the expected pointee type of the function's 1st argument.
```

type_tag_for_datatype

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
X						

When declaring a variable, use __attribute__((type_tag_for_datatype(kind, type))) to create a type tag that is tied to the type argument given to the attribute.

In the attribute prototype above:

kind is an identifier that should be used when annotating all applicable type tags.

type indicates the name of the type.

Clang supports annotating type tags of two forms.

Type tag that is a reference to a declared identifier. Use __attribute__((type_tag_for_datatype(kind, type))) when declaring that identifier:

```
typedef int MPI_Datatype;
extern struct mpi_datatype mpi_datatype_int
```

```
  __attribute__(( type_tag_for_datatype(mpi,int) ));
#define MPI_INT ((MPI_Datatype) &mpi_datatype_int)
// &mpi_datatype_int is a type tag. It is tied to type "int".
```

Type tag that is an integral literal. Declare a static const variable with an initializer value and attach __attribute__((type_tag_for_datatype(kind, type))) on that declaration:

```
typedef int MPI_Datatype;
static const MPI_Datatype mpi_datatype_int
  __attribute__(( type_tag_for_datatype(mpi,int) )) = 42;
#define MPI_INT ((MPI_Datatype) 42)
// The number 42 is a type tag. It is tied to type "int".
```

The type_tag_for_datatype attribute also accepts an optional third argument that determines how the type of the function argument specified by either arg_idx or ptr_idx is compared against the type associated with the type tag. (Recall that for the argument_with_type_tag attribute, the type of the function argument specified by arg_idx is compared against the type associated with the type tag. Also recall that for the pointer_with_type_tag attribute, the pointee type of the function argument specified by ptr_idx is compared against the type associated with the type tag.) There are two supported values for this optional third argument:

layout_compatible will cause types to be compared according to layout-compatibility rules (In C++11 [class.mem] p 17, 18, see the layout-compatibility rules for two standard-layout struct types and for two standard-layout union types). This is useful when creating a type tag associated with a struct or union type. For example:

```
/* In mpi.h */
typedef int MPI_Datatype;
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)

int MPI_Send(void *buf, int count, MPI_Datatype datatype, ...)
  __attribute__(( pointer_with_type_tag(mpi,1,3) ));

/* In user code */
struct my_pair { double a; int b; };
struct my_pair *buffer;
MPI_Send(buffer, 1, MPI_DOUBLE_INT /*, ... */); // no warning because the
                                  // layout of my_pair is
                                  // compatible with that of
                                  // internal_mpi_double_int
```

```
struct my_int_pair { int a; int b; }
struct my_int_pair *buffer2;
MPI_Send(buffer2, 1, MPI_DOUBLE_INT /*, ... */); // warning because the
                                    // layout of my_int_pair
                                    // does not match that of
                                    // internal_mpi_double_int
```

must_be_null specifies that the function argument specified by either arg_idx (for the argument_with_type_tag attribute) or ptr_idx (for the pointer_with_type_tag attribute) should be a null pointer constant. The second argument to the type_tag_for_datatype attribute is ignored. For example:

```
/* In mpi.h */
typedef int MPI_Datatype;
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)
int MPI_Send(void *buf, int count, MPI_Datatype datatype, ...)
  __attribute__(( pointer_with_type_tag(mpi,1,3) ));

/* In user code */
struct my_pair { double a; int b; };
struct my_pair *buffer;
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)

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
				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)

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
				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)

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
				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)

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
				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)

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
				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

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute

				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

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
				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

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
				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.

nonnull (gnu::nonnull)

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
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 (gnu::returns_nonnull)

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	Pragma	Pragma clang attribute
X	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