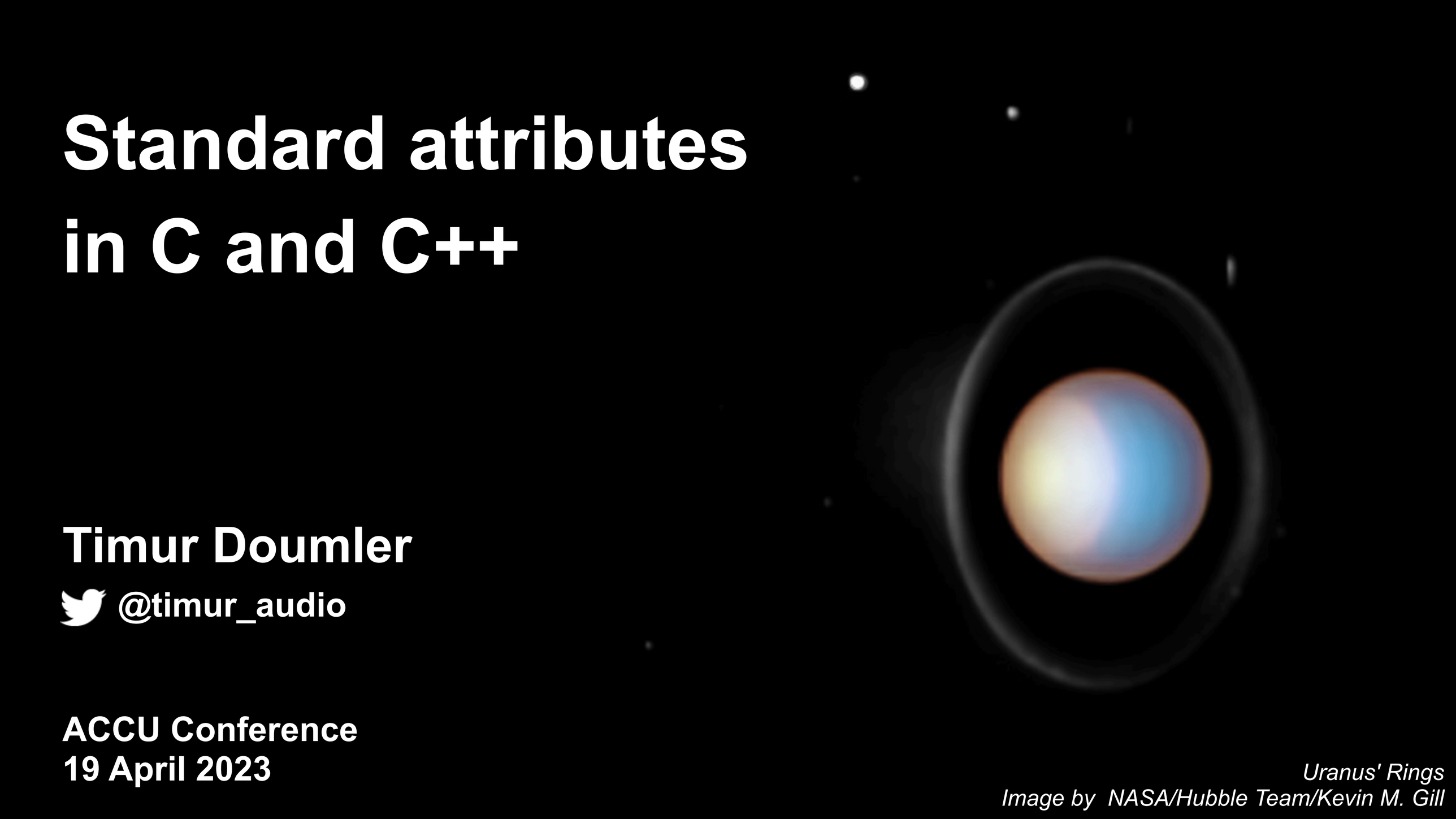
Standard attributes in C and C++

Timur Doumler

@timur_audio

ACCU Conference
19 April 2023



Uranus' Rings
Image by NASA/Hubble Team/Kevin M. Gill

- What are standard attributes?
- History of standardisation
- Standard attributes in C++
  - Classification
  - Usage
  - Caveats
  - Availability in major compilers
- Standard attributes in C
- Are standard attributes "ignorable"?
  - Syntactic ignorability
  - Semantic ignorability
  - Language design rule
- Future work

- What are standard attributes?
- History of standardisation
- Standard attributes in C++
  - Classification
  - Usage
  - Caveats
  - Availability in major compilers
- Standard attributes in C
- Are standard attributes "ignorable"?
  - Syntactic ignorability
  - Semantic ignorability
  - Language design rule
- Future work

[dcl.attr.grammar] p1

Attributes specify additional information for various source constructs such as types, variables, names, blocks, or translation units.

Towards support for attributes in C++ (Revision 6)

Jens Maurer, Michael Wong
jens.maurer@gmx.net
michaelw@ca.ibm.com

Document number: N2761
Date: 2008-09-18
Project: Programming Language C++, Core Working Group
Reply-to: Michael Wong (michaelw@ca.ibm.com)
Revision: 6

General Attributes for C++

1 Overview

The idea is to be able to annotate some entities in C++ with additional information. Currently, there is no means to do that short of inventing a new keyword and augmenting the grammar accordingly, thereby reserving yet another name of the user's namespace. This proposal will survey existing industry practice for extending the C++ syntax, and presents a general means for such annotations, including its integration into the C++ grammar. Specific attributes are not introduced in this proposal. It does not obviate the ability to add or overload keywords where appropriate, but it does reduce such need and add an ability to extend the language. This proposal will allow many C++0x proposals to move forward. A draft form of this proposal was presented in Oxford and received

Towards support for attributes in C++ (Revision 6)

Jens Maurer, Michael Wong
jens.maurer@gmx.net
michaelw@ca.ibm.com

Document number: N2761
Date: 2008-09-18
Project: Programming Language C++, Core Working Group
Reply-to: Michael Wong (michaelw@ca.ibm.com)
Revision: 6

General Attributes for C++

1 Overview

The idea is to be able to annotate some entities in C++ with additional information. Currently, there is no means to do that short of inventing a new keyword and augmenting the grammar accordingly, thereby reserving yet another name of the user's namespace. This proposal will survey existing industry practice for extending the C++ syntax, and presents a general means for such annotations, including its integration into the C++ grammar. Specific attributes are not introduced in this proposal. It does not obviate the ability to add or overload keywords where appropriate, but it does reduce such need and add an ability to extend the language. This proposal will allow many C++0x proposals to move forward. A draft form of this proposal was presented in Oxford and received

Towards support for attributes in C++ (Revision 6)

Jens Maurer, Michael Wong
jens.maurer@gmx.net
michaelw@ca.ibm.com

Document number: N2761
https://wg21.link/N2761
Date: 2008-09-18
Project: Programming Language C++, Core Working Group
Reply-to: Michael Wong (michaelw@ca.ibm.com)
Revision: 6

General Attributes for C++

1 Overview

The idea is to be able to annotate some entities in C++ with additional information. Currently, there is no means to do that short of inventing a new keyword and augmenting the grammar accordingly, thereby reserving yet another name of the user's namespace. This proposal will survey existing industry practice for extending the C++ syntax, and presents a general means for such annotations, including its integration into the C++ grammar. Specific attributes are not introduced in this proposal. It does not obviate the ability to add or overload keywords where appropriate, but it does reduce such need and add an ability to extend the language. This proposal will allow many C++0x proposals to move forward. A draft form of this proposal was presented in Oxford and received

Towards support for attributes in C++ (Revision 6)

Jens Maurer, Michael Wong
jens.maurer@gmx.net
michaelw@ca.ibm.com

Document number: N2761
https://wg21.link/N2761
Date: 2008-09-18
Project: Programming Language C++, Core Working Group
Reply-to: Michael Wong (michaelw@ca.ibm.com)
Revision: 6

General Attributes for C++

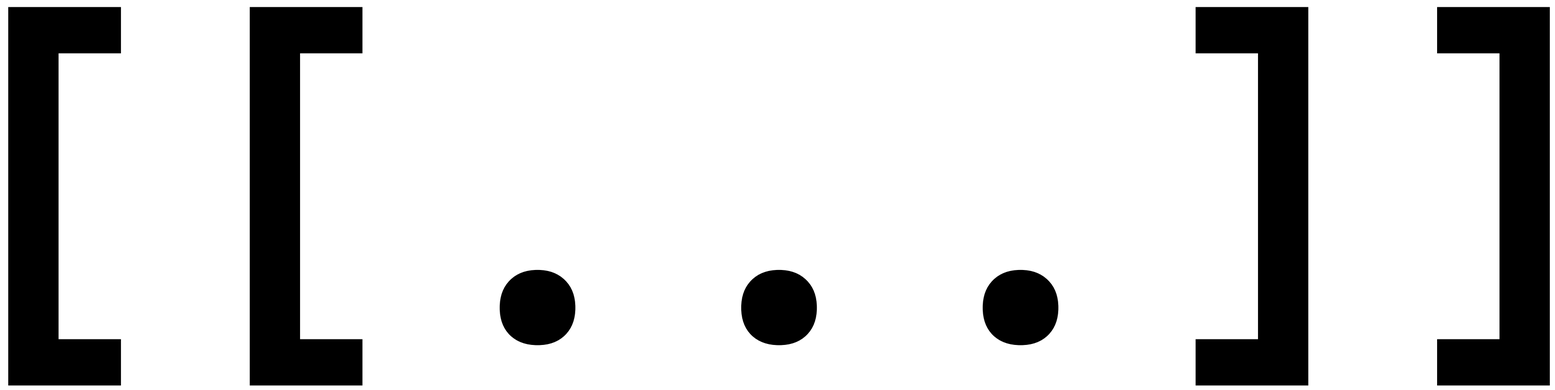
1 Overview

The idea is to be able to annotate some entities in C++ with additional information. Currently, there is no means to do that short of inventing a new keyword and augmenting the grammar accordingly, thereby reserving yet another name of the user's namespace. This proposal will survey existing industry practice for extending the C++ syntax, and presents a general means for such annotations, including its integration into the C++ grammar. Specific attributes are not introduced in this proposal. It does not obviate the ability to add or overload keywords where appropriate, but it does reduce such need and add an ability to extend the language. This proposal will allow many C++0x proposals to move forward. A draft form of this proposal was presented in Oxford and received

Certainly, we would advise anyone who propose an attribute to consider comments on the following area which will help guide them in making the decision of whether to use attributes or not:

- The feature is used in declarations or definitions only.
- Is the feature is of use to a limited audience only (e.g., alignment)?
- The feature does not modify the type system (e.g., thread_local) and hence does not require new mangling?
- The feature is a "minor annotation" to a declaration that does not alter its semantics significantly. (Test: Take away the annotation. Does the remaining declaration still make sense?
- Is it a vendor-specific extension?
- Is it a language Bindings on C++ that has no other way of tying to a type or scope(e.g. OpenMP)
- How does this change Overload resolution?
- What is the effect in typedefs, will it require cloning?

```cpp
// MSVC pre-C++11
__declspec(noreturn) void terminate() { /* ... */ }

// GCC pre-C++11
__attribute__((unused)) void test() { /* ... */ }
```

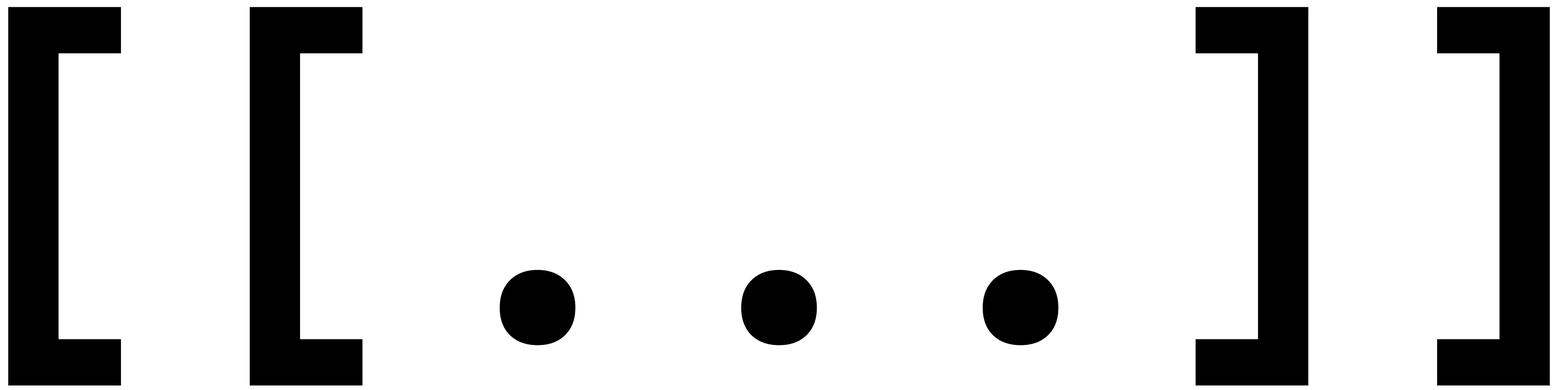


```cpp
// MSVC pre-C++11
__declspec(noreturn) void terminate() { /* ... */ }

// GCC pre-C++11
__attribute__((unused)) void test() { /* ... */ }
```

```cpp
// added in C++11:
[[noreturn]] void terminate() { /* ... */ }

// added in C++17:
[[maybe_unused]] void test() { /* ... */ }
```

```
// Standard attributes
[[nodiscard]]
[[fallthrough]]
[[deprecated("Use this other thing instead")]]
...


// Non-standard ("vendor-specific") attributes
[[clang::always_inline]]
[[gnu::pure]]
[[omp::sequence(directive(parallel), directive(for))]]
...
```

```
// Standard attributes
[[nodiscard]]
[[fallthrough]]
[[deprecated("Use this other thing instead")]]
...


// Non-standard ("vendor-specific") attributes
[[clang::always_inline]]
[[gnu::pure]]
[[omp::sequence(directive(parallel), directive(for))]]
...
```



```
int main() {
  int a[2] = {666, 42};
  return a[1];
}
```

```
int main() {
  int a[2] = {666, 42};
  return a[[]{ return 1; }()];
}
```

```
int main() {
  int a[2] = {666, 42};
  return a[[]{ return 1; }()]; // Error!
}
```

```
int main() {
  int a[2] = {666, 42};
  return a[ []{ return 1; }()]; // Error!
}
```

```
int main() {
  int a[2] = {666, 42};
  return a[([]{ return 1; }())]; // OK
}
```

[dcl.attr.grammar] p5

Each *attribute-specifier-seq* is said to appertain to some entity or statement, identified by the syntactic context where it appears.

```
// Attribute appertaining to class template declaration:
template <typename T>
class [[deprecated]] auto_ptr;
```

```
// Attribute appertaining to class template declaration:
template <typename T>
class [[deprecated]] auto_ptr;

// Attribute appertaining to function declaration:
[[nodiscard]] bool empty() const noexcept;
```

```cpp
// Attribute appertaining to class template declaration:
template <typename T>
class [[deprecated]] auto_ptr;

// Attribute appertaining to function declaration:
[[nodiscard]] bool empty() const noexcept;

// Attributes appertaining to statements and labels:
int f(int i) {
    switch(i) {
        case 1: [[fallthrough]];
        [[likely]] case 2: return 1;
    }
    return 2;
}
```

Towards support for attributes in C++ (Revision 6)

Jens Maurer, Michael Wong
jens.maurer@gmx.net
michaelw@ca.ibm.com

Document number: N2761=08-0271
Date: 2008-09-18
Project: Programming Language C++, Core Working Group
Reply-to: Michael Wong (michaelw@ca.ibm.com)
Revision: 6

General Attributes for C++

1 Overview

The idea is to be able to annotate some entities in C++ with additional information. Currently, there is no means to do that short of inventing a new keyword and augmenting the grammar accordingly, thereby reserving yet another name of the user's namespace. This proposal will survey existing industry practice for extending the C++ syntax, and presents a general means for such annotations, including its integration into the C++ grammar. Specific attributes are not introduced in this proposal. It does not obviate the ability to add or overload keywords where appropriate, but it does reduce such need and add an ability to extend the language. This proposal will allow many C++0x proposals to move forward. A draft form of this proposal was presented in Oxford and received

Certainly, we would advise anyone who propose an attribute to consider comments on the following area which will help guide them in making the decision of whether to use attributes or not:

- The feature is used in declarations or definitions only.
- Is the feature is of use to a limited audience only (e.g., alignment)?
- The feature does not modify the type system (e.g., thread_local) and hence does not require new mangling?
- The feature is a "minor annotation" to a declaration that does not alter its semantics significantly. (Test: Take away the annotation. Does the remaining declaration still make sense?
- Is it a vendor-specific extension?
- Is it a language Bindings on C++ that has no other way of tying to a type or scope(e.g. OpenMP)
- How does this change Overload resolution?
- What is the effect in typedefs, will it require cloning?

Certainly, we would advise anyone who propose an attribute to consider comments on the following area which will help guide them in making the decision of whether to use attributes or not:

- The feature is used in declarations or definitions only.
- Is the feature is of use to a limited audience only (e.g., alignment)?
- The feature does not modify the type system (e.g., thread_local) and hence does not require new mangling?
- The feature is a "minor annotation" to a declaration that does not alter its semantics significantly. (Test: Take away the annotation. Does the remaining declaration still make sense?
- Is it a vendor-specific extension?
- Is it a language Bindings on C++ that has no other way of tying to a type or scope(e.g. OpenMP)
- How does this change Overload resolution?
- What is the effect in typedefs, will it require cloning?

Certainly, we would advise anyone who propose an attribute to consider comments on the following area which will help guide them in making the decision of whether to use attributes or not:

- The feature is used in declarations or definitions only.
- Is the feature is of use to a limited audience only (e.g., alignment)?
- The feature does not modify the type system (e.g., thread_local) and hence does not require new mangling?
- The feature is a "minor annotation" to a declaration that does not alter its semantics significantly. (Test: Take away the annotation. Does the remaining declaration still make sense?
- Is it a vendor-specific extension?
- Is it a language Bindings on C++ that has no other way of tying to a type or scope(e.g. OpenMP)
- How does this change Overload resolution?
- What is the effect in typedefs, will it require cloning?

Some guidance for when not to use an attribute and use/reuse a keyword

- The feature is used in expressions as opposed to declarations.
- The feature is of use to a broad audience.
- The feature is a central part of the declaration that significantly affects its requirements/semantics (e.g., constexpr).
- The feature modifies the type system and/or overload resolution in a significant way (e.g., rvalue references). (However, something like near and far pointers should probably still be handled by attributes, although those do affect the type system.)
- The feature is used everywhere on every instance of class, or statements

Some guidance for when not to use an attribute and use/reuse a keyword

- The feature is used in expressions as opposed to declarations.
- The feature is of use to a broad audience.
- The feature is a central part of the declaration that significantly affects its requirements/semantics (e.g., constexpr).
- The feature modifies the type system and/or overload resolution in a significant way (e.g., rvalue references). (However, something like near and far pointers should probably still be handled by attributes, although those do affect the type system.)
- The feature is used everywhere on every instance of class, or statements

Good choices in attributes include:

- align(unsigned int)
- pure (promise that a function always returns the same value)
- probably(unsigned int) (hint for if, switch, ...)
    - if [[probably(true)]] (i == 42) { ... }
- noreturn (the function never returns)
- deprecated (functions)
- noalias (promises no other path to the object)
- unused (parameter name)
- final on virtual function declaration and on a class
- not_hiding (name of function does not hide something in a base class)
- register (if we had a time machine)
- owner (a pointer is owned and it is the owner's duty to delete it)

Bad choices in attributes include:

- C99 restrict (affects the type system)
- huge (really long long type, e.g. 256bits)
- C++ const

Good choices in attributes include:

- align(unsigned int)
- pure (promise that a function always returns the same value)
- probably(unsigned int) (hint for if, switch, ...)
  - if [[probably(true)]] (i == 42) { ... }
- noreturn (the function never returns)
- deprecated (functions)
- noalias (promises no other path to the object)
- unused (parameter name)
- final on virtual function declaration and on a class
- not_hiding (name of function does not hide something in a base class)
- register (if we had a time machine)
- owner (a pointer is owned and it is the owner's duty to delete it)

Bad choices in attributes include:

- C99 restrict (affects the type system)
- huge (really long long type, e.g. 256bits)
- C++ const

Good choices in attributes include:

- align(unsigned int)
- pure (promise that a function always returns the same value)
- probably(unsigned int) (hint for if, switch, ...)
  - if [[probably(true)]] (i == 42) { ... }
- noreturn (the function never returns)
- deprecated (functions)
- noalias (promises no other path to the object)
- unused (parameter name)
- final on virtual function declaration and on a class
- not_hiding (name of function does not hide something in a base class)
- register (if we had a time machine)
- owner (a pointer is owned and it is the owner's duty to delete it)

Bad choices in attributes include:

- C99 restrict (affects the type system)
- huge (really long long type, e.g. 256bits)
- C++ const

Good choices in attributes include:

- align(unsigned int)
- pure (promise that a function always returns the same value)
- probably(unsigned int) (hint for if, switch, ...)
    - if [[probably(true)]] (i == 42) { ... }
- noreturn (the function never returns)
- deprecated (functions)
- noalias (promises no other path to the object)
- unused (parameter name)
- final on virtual function declaration and on a class
- not_hiding (name of function does not hide something in a base class)
- register (if we had a time machine)
- owner (a pointer is owned and it is the owner's duty to delete it)

Bad choices in attributes include:

- C99 restrict (affects the type system)
- huge (really long long type, e.g. 256bits)
- C++ const

Good choices in attributes include:

- align(unsigned int)
- pure (promise that a function always returns the same value)
- probably(unsigned int) (hint for if, switch, ...)
    - if [[probably(true)]] (i == 42) { ... }
- noreturn (the function never returns)
- deprecated (functions)
- noalias (promises no other path to the object)
- unused (parameter name)
- final on virtual function declaration and on a class
- not_hiding (name of function does not hide something in a base class)
- register (if we had a time machine)
- owner (a pointer is owned and it is the owner's duty to delete it)

Bad choices in attributes include:

- C99 restrict (affects the type system)
- huge (really long long type, e.g. 256bits)
- C++ const

- What are standard attributes?
- History of standardisation
- Standard attributes in C++
  - Classification
  - Usage
  - Caveats
  - Availability in major compilers
- Standard attributes in C
- Are standard attributes "ignorable"?
  - Syntactic ignorability
  - Semantic ignorability
  - Language design rule
- Future work

```
// C++11
[[noreturn]]
[[carries_dependency]]
```

```
// C++11
[[noreturn]]
[[carries_dependency]]

// C++14
[[deprecated]]
[[deprecated("reason")]]
```

```
// C++11
[[noreturn]]
[[carries_dependency]]

// C++14
[[deprecated]]
[[deprecated("reason")]]

// C++17
[[fallthrough]]
[[nodiscard]]
[[maybe_unused]]
```

```
// C++11
[[noreturn]]
[[carries_dependency]]

// C++14
[[deprecated]]
[[deprecated("reason")]]

// C++17
[[fallthrough]]
[[nodiscard]]
[[maybe_unused]]
```

```
// C++20
[[nodiscard("reason")]]
[[likely]]
[[unlikely]]
[[no_unique_address]]
```

```
// C++11
[[noreturn]]
[[carries_dependency]]

// C++14
[[deprecated]]
[[deprecated("reason")]]

// C++17
[[fallthrough]]
[[nodiscard]]
[[maybe_unused]]
```

```
// C++20
[[nodiscard("reason")]]
[[likely]]
[[unlikely]]
[[no_unique_address]]

//C++23
[[assume(expression)]]
```

```
// C++11
[[noreturn]]
[[carries_dependency]]

// C++14
[[deprecated]]
[[deprecated("reason")]]

// C++17
[[fallthrough]]
[[nodiscard]]
[[maybe_unused]]
```

```
// C++20
[[nodiscard("reason")]]
[[likely]]
[[unlikely]]
[[no_unique_address]]

//C++23
[[assume(expression)]]
```

```
// C23
[[noreturn]]
[[deprecated]]
[[fallthrough]]
[[nodiscard]]
[[maybe_unused]]
```

```
// C23
[[noreturn]]
[[deprecated]]
[[fallthrough]]
[[nodiscard]]
[[maybe_unused]]
[[unsequenced]]
[[reproducible]]
```

```
// C23
[[noreturn]]
[[deprecated]]
[[fallthrough]]
[[nodiscard]]
[[maybe_unused]]
[[unsequenced]]
[[reproducible]]
```

- What are standard attributes?
- History of standardisation
- Standard attributes in C++
  - Classification
  - Usage
  - Caveats
  - Availability in major compilers
- Standard attributes in C
- Are standard attributes "ignorable"?
  - Syntactic ignorability
  - Semantic ignorability
  - Language design rule
- Future work

```
// Attributes to enable/disable warnings
[[deprecated]]
[[maybe_unused]]
[[fallthrough]]
[[nodiscard]]

// Attributes targeting backend (can trigger optimisations and UB)
[[noreturn]]
[[carries_dependency]]
[[likely]]
[[unlikely]]
[[assume]]

// Attribute targeting ABI
[[no_unique_address]]
```

1

1

2

1

2


3


CAUTION
WET
FLOOR
Rubbermaid

[[deprecated]]

[[maybe_unused]]

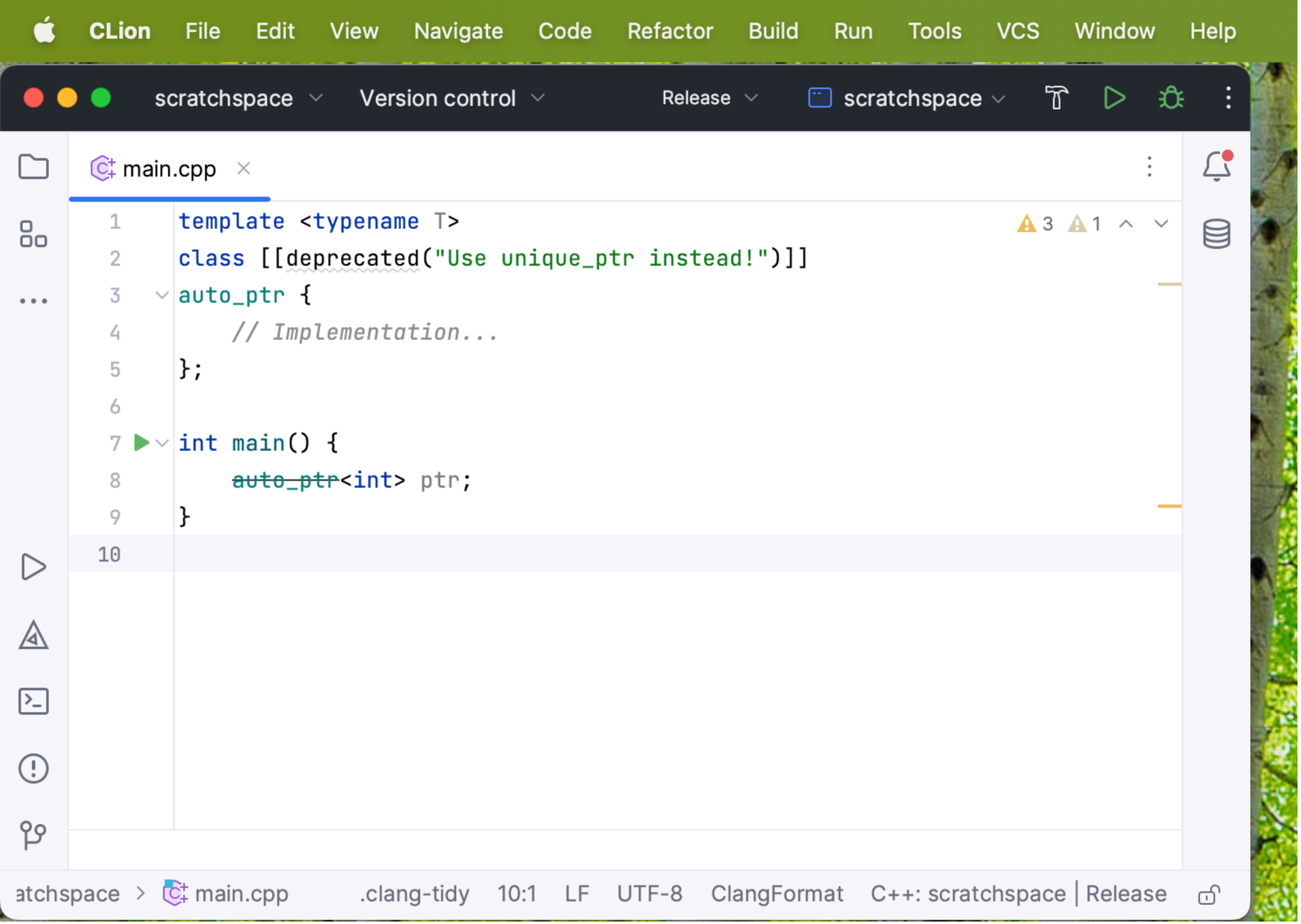
[[fallthrough]]

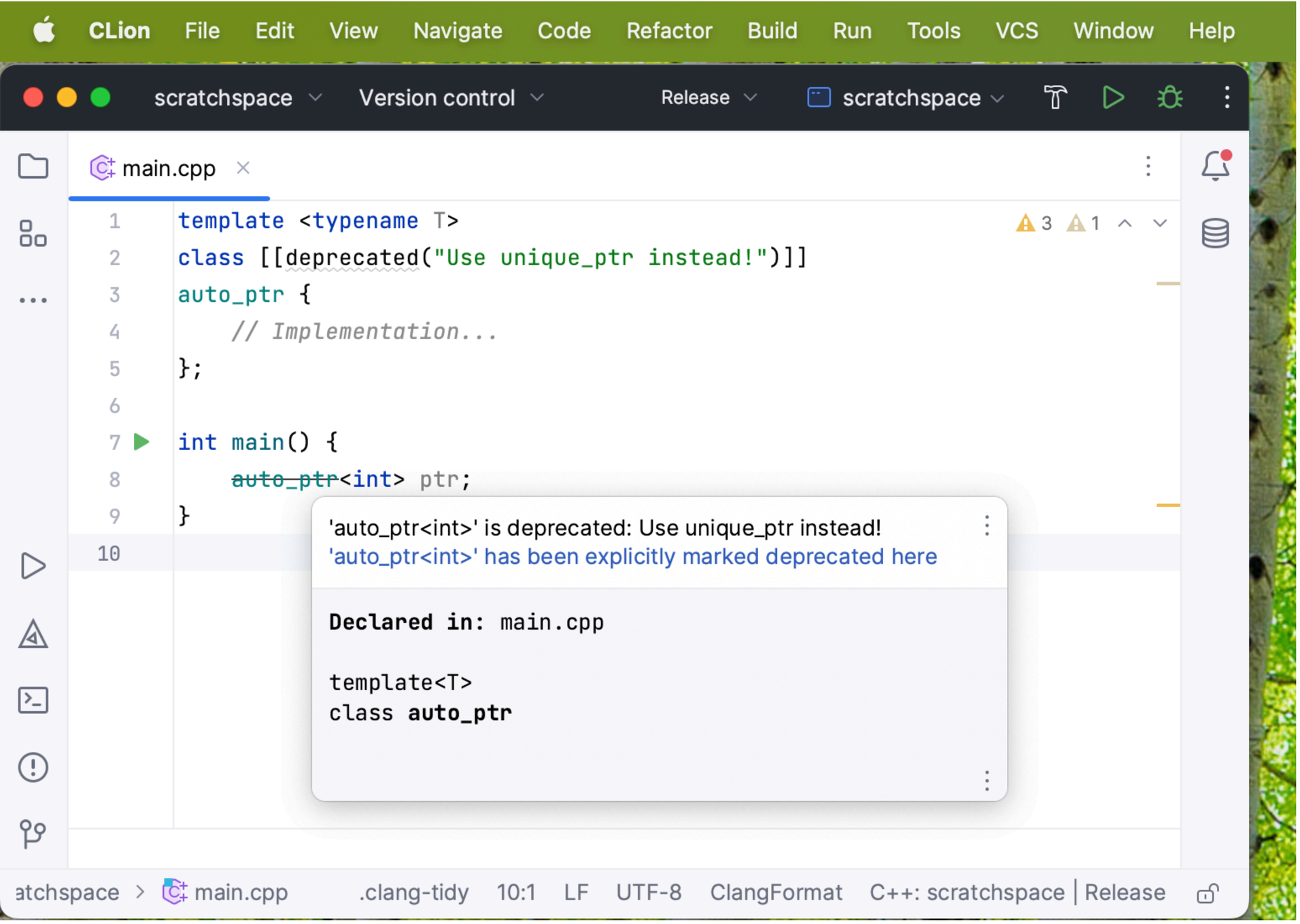
[[nodiscard]]

```
template <typename T>
class [[deprecated]]
auto_ptr {
    // Implementation...
};
```

```
template <typename T>
class [[deprecated]]
auto_ptr {
    // Implementation...
};

int main() {
    auto_ptr<int> ptr;   // Warning: auto_ptr is deprecated
}
```

```cpp
template <typename T>
class [[deprecated("Use unique_ptr instead!")]]
auto_ptr {
    // Implementation...
};

int main() {
    auto_ptr<int> ptr;   // Warning: auto_ptr is deprecated:
}                        // Use unique_ptr instead!
```


CLion File Edit View Navigate Code Refactor Build Run Tools VCS Window Help
scratchspace Version control Release scratchspace
main.cpp
template <typename T>
class [[deprecated("Use unique_ptr instead!")]]
auto_ptr {
// Implementation...
};
int main() {
auto_ptr<int> ptr;
}
3 1
atchspace main.cpp .clang-tidy 10:1 LF UTF-8 ClangFormat C++: scratchspace | Release


CLion File Edit View Navigate Code Refactor Build Run Tools VCS Window Help
scratchspace Version control Release scratchspace
main.cpp
template <typename T>
class [[deprecated("Use unique_ptr instead!")]]
auto_ptr {
// Implementation...
};
int main() {
auto_ptr<int> ptr;
}
'auto_ptr<int>' is deprecated: Use unique_ptr instead!
'auto_ptr<int>' has been explicitly marked deprecated here
Declared in: main.cpp
template<T>
class auto_ptr
atchspace main.cpp .clang-tidy 10:1 LF UTF-8 ClangFormat C++: scratchspace | Release

[[deprecated]]

- Syntax:
  - Has optional character literal argument
  - Appertains to declarations of a class, typedef, variable, data member, function, namespace, enum, template, or template specialisation
- Availability:
  - since C++14
  - MSVC, GCC, Clang, ICC
- Recommendation:
  - Use in your library when needed
  - Always use string argument to tell user what to use instead

[[deprecated]]

[[maybe_unused]]

[[fallthrough]]

[[nodiscard]]

```
void didFinishProcessingWidget(const Widget& widget, const Error& error)
{
    String errorString = error ? error.description : {};
    DEBUG_LOG("didFinishProcessingWidget, error: " + errorString);

    // do stuff..
}
```

```
void didFinishProcessingWidget(const Widget& widget, const Error& error)
{
    String errorString = error ? error.description : {};
    // Warning: unused variable: 'errorString'
    DEBUG_LOG("didFinishProcessingWidget, error: " + errorString);

    // do stuff..
}
```

```
#define UNUSED(x) (void)(x)

void didFinishProcessingWidget(const Widget& widget, const Error& error)
{
    String errorString = error ? error.description : {};
    UNUSED(errorString);
    DEBUG_LOG("didFinishProcessingWidget, error: " + errorString);

    // do stuff..
}
```

```
template <typename... Types>
void ignoreUnused(Types&&...) noexcept {}

void didFinishProcessingWidget(const Widget& widget, const Error& error)
{
    String errorString = error ? error.description : {};
    ignoreUnused(errorString);
    DEBUG_LOG("didFinishProcessingWidget, error: " + errorString);

    // do stuff..
}
```

```
void didFinishProcessingWidget(const Widget& widget, const Error& error)
{
    [[maybe_unused]] String errorString = error ? error.description : {};
    DEBUG_LOG("didFinishProcessingWidget, error: " + errorString);

    // do stuff..
}
```

[[maybe_unused]]

- Syntax:
  - Appertains to declarations of a class, typedef, variable, data member, function, enum, or structured binding
- Availability:
  - since C++17
  - MSVC, GCC, Clang, ICC
- Recommendation:
  - Use when needed
  - Consider replacing your existing solution with [[maybe_unused]]

```
struct A {};
struct [[maybe_unused]] B {};

int main() {
  A a1;
  [[maybe_unused]] A a2;

  B b1;
  [[maybe_unused]] B b2;
}
```

```
struct A {};
struct [[maybe_unused]] B {};

int main() {
  A a1;
  [[maybe_unused]] A a2;

  B b1;
  [[maybe_unused]] B b2;
}
```

```
struct A {};
struct [[maybe_unused]] B {};

int main() {
  A a1;   // MSVC/GCC/Clang: warning: a1 is unused
  [[maybe_unused]] A a2;  // No warning

  B b1;   // MSVC: warning: b1 is unused; GCC/Clang: no warning
  [[maybe_unused]] B b2;  // No warning
}
```

[[deprecated]]

[[maybe_unused]]

[[fallthrough]]

[[nodiscard]]

```
void f(int n) {
    switch (n) {
        case 0:
        case 1:
            doSomething();
        case 2:
            doSomethingElse();
            break;
        default:
            assert("Something went wrong");
            break;
    }
}
```

```
void f(int n) {
    switch (n) {
        case 0:
        case 1:
            doSomething();
        case 2:
            doSomethingElse();
            break;
        default:
            assert("Something went wrong");
            break;
    }
}
```

```
void f(int n) {
    switch (n) {
        case 0:
        case 1:
            doSomething();
        case 2:                         // Warning: implicit fallthrough
            doSomethingElse();
            break;
        default:
            assert("Something went wrong");
            break;
    }
}
```

```
void f(int n) {
    switch (n) {
        case 0:
        case 1:
            doSomething();
            doSomethingElse();
            break;
        case 2:
            doSomethingElse();
            break;
        default:
            assert("Something went wrong");
            break;
    }
}
```

```
void f(int n) {
    switch (n) {
        case 0:
        case 1:
            doSomething();
            [[fallthrough]];
        case 2:
            doSomethingElse();
            break;
        default:
            assert("Something went wrong");
            break;
    }
}
```

`[[fallthrough]]`

- Syntax:
  - Can appear only as a single statement: `[[fallthrough]];`
    ("Appertains to a null statement")
  - Only inside a switch statement, just before a case label
- Availability:
  - since C++17
  - MSVC, GCC, Clang, ICC
- Recommendation:
  - Use whenever needed

```
switch (n) {
  case 1:
  case 2:
    g();
    [[fallthrough]];
  case 3:                 // warning on fallthrough discouraged
    do {
      [[fallthrough]];    // error: next statement is not part of the same substatement execution
    } while (false);
  case 6:
    do {
      [[fallthrough]];    // error: next statement is not part of the same substatement execution
    } while (n--);
  case 7:
    while (false) {
      [[fallthrough]];    // error: next statement is not part of the same substatement execution
    }
  case 5:
    h();
  case 4:                 // implementation may warn on fallthrough
    i();
    [[fallthrough]];      // error
}
```

[[deprecated]]

[[maybe_unused]]

[[fallthrough]]

[[nodiscard]]

```
[[nodiscard]] int f() {
    return 42;
}
```

```
[[nodiscard]] int f() {
    return 42;
}

int main() {
    f();  // Warning: ignoring return value of nodiscard function
}
```

[[nodiscard]]

- Use when discarding return value would be a wrong use of the API, that can lead to bugs, for example:
  - memory leaks
  - race conditions
  - undefined behaviour
  - wrong functionality

```
template <typename T>
class vector {
public:
    bool empty() const noexcept;
    // Other stuff...
};
```

```
template <typename T>
class vector {
public:
    bool empty() const noexcept;
    // Other stuff...
};

int main() {
    vector<int> v = {1, 2, 3};
    v.empty();  // User expects that now vector.size() == 0 but it's not :(
}
```

```cpp
template <typename T>
class vector {
public:
    [[nodiscard]] bool empty() const noexcept;
    // Other stuff...
};

int main() {
    vector<int> v = {1, 2, 3};
    v.empty();  // Warning: ignoring return value of nodiscard function
}
```

```
template <typename T>
class vector {
public:
    [[nodiscard("Did you mean to call clear?")]] bool empty() const noexcept;
    // Other stuff...
};

int main() {
    vector<int> v = {1, 2, 3};
    v.empty();  // Warning: ignoring return value of nodiscard function:
}               // Did you mean to call clear?
```

```
// Other examples:

smartPtr.release();
seqLock.lock();
std::launder(ptr);
```

[[nodiscard]]

- Syntax:
  - Appertains to a function, enum or class declaration
- Availability:
  - since C++17 (without character literal argument)
  - since C++20 (with character literal argument)
  - MSVC, GCC, Clang, ICC
- Recommendation:
  - Use when discarding return value would be a wrong use of the API (wrong functionality, memory leak, UB, ...)
  - If on C++20 or above, use character literal argument

Project: ISO JTC1/SC22/WG21: Programming Language C++
Doc No: WG21 **P0600R1**
Date: 2017-11-09
Reply to: Nicolai Josuttis (nico@josuttis.de)
Audience: LEWG, LWG
Prev. Version: P0600R0

[[nodiscard]] in the Library, Rev1

Updates for Version R1:

- added empty()
- added launder() in wording
- no [[nodiscard]] for C functions (removed malloc())
- require also to add [[nodiscard]] in the definition
- fixed reference paper and section numbering
- reason for not having a feature test macro

C++17 introduced the [[nodiscard]] attribute.

The question is, where to apply it now in the standard library.

We suggest a conservative approach:

It should be added where:

It should be added where:

- For existing API's
  - not using the return value always is a "huge mistake" (e.g. always resulting in resource leak)
  - not using the return value is a source of trouble and easily can happen (not obvious that something is wrong)
- For new API's (not been in the C++ standard yet)
  - not using the return value is usually an error.

It should not be added when:

- For existing API's
  - not using the return value is a possible/common way of programming at least for some input
    - for example for realloc(), which acts like free when the new site is 0
  - not using the return value makes no sense but doesn't hurt and is usually not an error (e.g., because programmers meant to ask for a state change).
  - it is a C function, because their declaration might not be under control of the C++ implementation

For example:

Function	**[[nodiscard]] ?**	**Remark**
malloc()	no	expensive call, usually not using the return value is a resource leak. However, a C function.
realloc()	no	realloc() with new size 0 acts like free()
async()	yes	not using the return value makes the call synchronous, which might be hard to detect.
launder()	yes	new API, where not using the return value makes no sense, because launder() does not white-wash. It just the return value


CAUTION
WET
FLOOR
Rubbermaid


DANGER
BEAR
TRAP

[[noreturn]]

[[carries_dependency]]

[[likely]]

[[unlikely]]

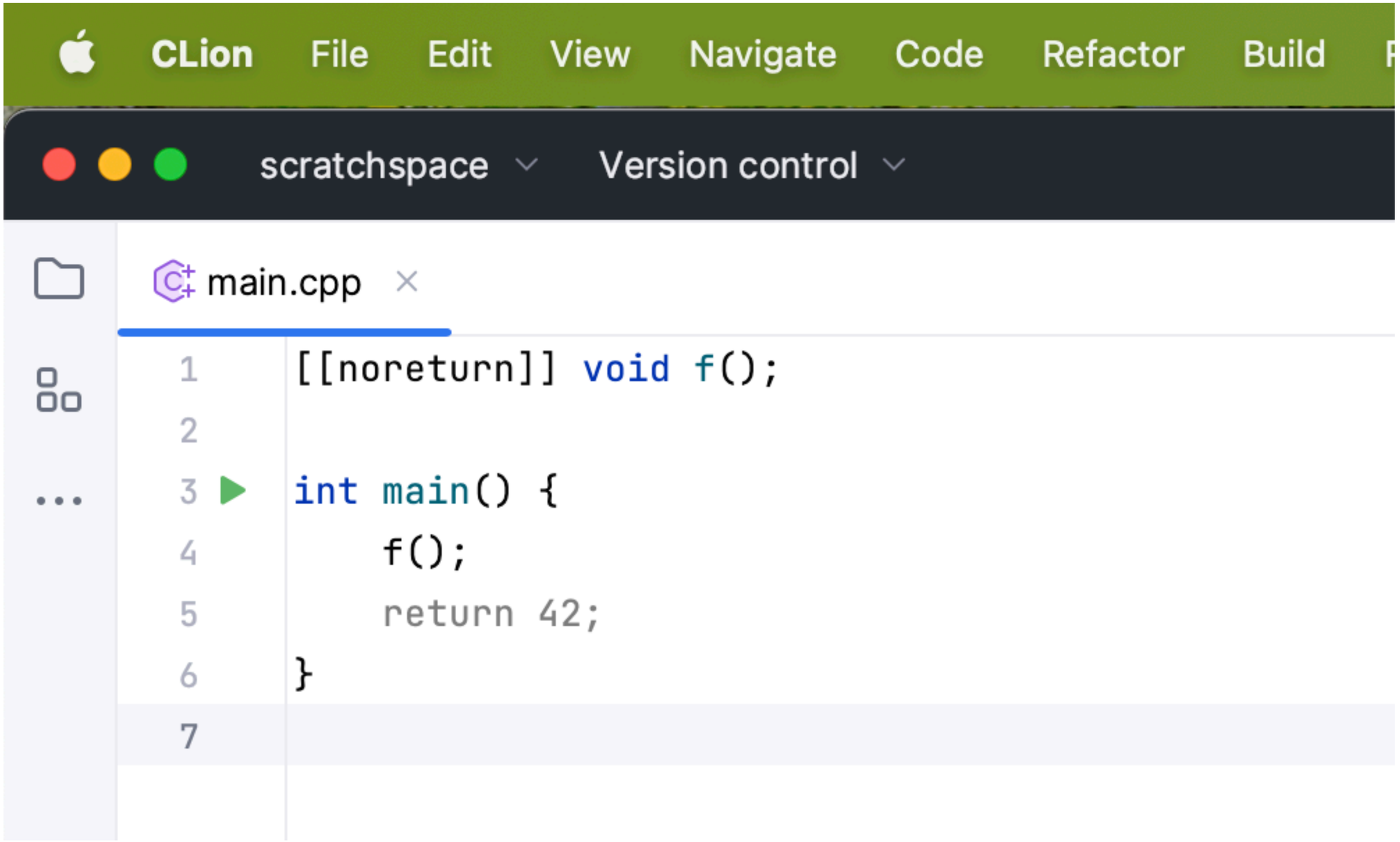
[[assume]]

```
[[noreturn]] void f() {
  throw "error";
}
```

[[noreturn]] in the C++ Standard Library

```
// Always throw:
std::rethrow_exception
std::rethrow_nested
std::throw_with_nested

// Always terminate the program:
std::abort
std::exit
std::quick_exit
std::terminate

// Always jump:
std::longjmp

// Always undefined behaviour:
std::unreachable
```

```
[[noreturn]] void f();

int main() {
  f();
  return 42;  // Dead code
}
```


CLion File Edit View Navigate Code Refactor Build
scratchspace
Version control
main.cpp
1 [[noreturn]] void f();
2
3 int main() {
4 f();
5 return 42;
6 }
7

```
[[noreturn]] void f() {}

int main() {
  f();  // Undefined behaviour!
  return 42;
}
```

[[noreturn]]

- Syntax:
  - Appertains to a function declaration
- Availability:
  - since C++11
  - MSVC, GCC, Clang, ICC
- Recommendation:
  - Expert-only usage (should be rare)
  - Keep in mind it can introduce UB

[[noreturn]]

[[carries_dependency]]

[[likely]]

[[unlikely]]

[[assume]]

```
void print(int* [[carries_dependency]] val);

std::atomic<int*> p;
// ...

int* local = p.load(std::memory_order_consume);
if(local)
    print(local);
```

```
void print(int* [[carries_dependency]] val);

std::atomic<int*> p;
// ...

int* local = p.load(std::memory_order_consume);
if(local)
    print(local);
```

p0750r1
Consume

Published Proposal, 11 February 2018

This version:
http://wg21.link/P0750

Authors:
JF Bastien (Apple)
Paul E. McKenney (IBM)

Audience:
SG1

Project:
ISO JTC1/SC22/WG21: Programming Language C++

Source:
github.com/jfbastien/papers/blob/master/source/P0750r1.bs

Implementation:
github.com/jfbastien/stdconsume

Abstract

Fixing memory order consume.

[[carries_dependency]]

- Syntax:
  - Appertains to a function parameter declaration
- Availability:
  - since C++11 (but will probably be deprecated/removed)
  - no compiler implements it
- Recommendation:
  - Never use

[[noreturn]]

[[carries_dependency]]

[[likely]]

[[unlikely]]

[[assume]]

```
if (condition) [[likely]] {
  f();
}
else {
  g();
}
```

```
if (shouldSendOrder) [[likely]]
  sendOrder(order);
else
  doSomethingElse();
```

Caveats

- Do not influence branch predictor, only code layout!
  → no CPU instructions to give hints to branch predictor

- Many caveats when using them
  → Aaron Ballman: *"Don’t use the [[likely]] or [[unlikely]] attributes"*

- Often it does not actually optimise, and sometimes it pessimises!
  → Amir Kirsh & Tomer Vromen: *"C++20’s [[likely]] Attribute - Optimizations, Pessimizations, and [[unlikely]] Consequences"*

Don’t use the [[likely]] or [[unlikely]] attributes

Posted on 2020-08-27 by Aaron Ballman

C++20 introduced the likelihood attributes `[[likely]]` and `[[unlikely]]` as a way for a programmer to give an optimization hint to their implementation that a given code path is more or less likely to be taken. On its face, this seems like a great set of attributes because you can give hints to the optimizer in a way that is hopefully understood by all implementations and will result in faster performance. What’s not to love?

The attribute is specified to appertain to arbitrary statements or labels with the recommended practice “to optimize for the case where paths of execution including it are arbitrarily more likely|unlikely than any alternative path of execution that does not include such an attribute on a statement or label.” Pop quiz, what does this code do?

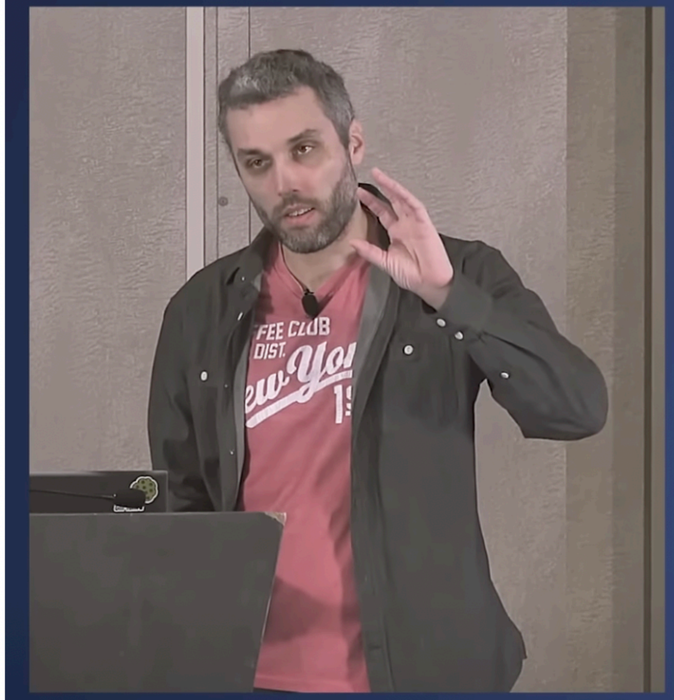
```
if (something) {
  [[likely]];
```

Let's review the rules we've got so far:

0) Never allow multiple likelihood attributes to appear in the same path of execution.
1) Only mark the dominating statement or label of the flow control path you want to optimize for.
2) Assume no two implementations will behave the same way for optimization behaviors with these attributes.
3) Prefer profile-guided optimization over likelihood attributes.
4) Not all flow control paths can be optimized.

These attributes are starting to look a bit more like some other code constructs we've seen in the past: the `register` keyword as an optimization hint to put things in registers and the `inline` keyword as an optimization hint to inline function bodies into the call site. Using `register` or `inline` for these purposes is often strongly discouraged because experience has shown that optimizer implementations eventually improved to the point where they

Cppcon 2022 The C++ Conference | September 12th-16th



Amir Kirsh & Tomer Vromen

C++20's [[likely]] Attribute: Optimizations, Pessimizations, and [[unlikely]] Consequences

Code layout

Case 5 is unlikely
=> put it further away

Case 7 is likely
=> put it closer

Default case is still last!

What if we mark default as [[likely]]?

https://godbolt.org/z/ceK9ddreb

```
int foo(int x)
{
    switch (x)
    {
        case 0:
            return foo0();
        case 1:
            return foo1();
        case 2:
            return foo2();
        case 3:
            return foo3();
        case 4:
            return foo4();
        [[unlikely]] case 5:
            return foo5();
        case 6:
            return foo6();
        [[likely]] case 7:
            return foo7();
        case 8:
            return foo8();
        case 9:
            return foo9();
        default:
            return bar(x);
    }
}
```

```
foo(int):
        cmp     edi, 9
        ja      .L2
        mov     edi, edi
        jmp     [QWORD PTR .L4[0+rdi*8]]
.L4:
        .quad   .L13
        .quad   .L12
        .quad   .L11
        .quad   .L10
        .quad   .L9
        .quad   .L8
        .quad   .L7
        .quad   .L6
        .quad   .L5
        .quad   .L3
.L6:
        jmp     foo7()
.L5:
        jmp     foo8()
.L7:
        jmp     foo6()
.L9:
        jmp     foo4()
.L10:
        jmp     foo3()
.L11:
        jmp     foo2()
.L12:
        jmp     foo1()
.L13:
        jmp     foo0()
.L3:
        jmp     foo9()
.L8:
        jmp     foo5()
.L2:
        jmp     bar(int)
```

Video Sponsorship Provided By:

think-cell

11:06 / 1:03:30 • Code Layout >

[[likely]], [[unlikely]]

- Syntax:
  - Appertain to labels and statements (other than declarations)
- Availability:
  - since C++20
  - MSVC, GCC, Clang, ICC
- Recommendation:
  - Avoid
  - Consider using only if it leads to a *measurable* perf improvement
  - Keep in mind this might change with a different compiler
  - Keep in mind non-straightforward semantic rules

```
if (shouldSendOrder) [[likely]]
  sendOrder(order);
else
  doSomethingElse();
```

[[noreturn]]
[[carries_dependency]]
[[likely]]
[[unlikely]]
[[assume]]

Portable assumptions

Timur Doumler (papers@timur.audio)

Document #: P1774R8
Date: 2022-06-14
Project: Programming Language C++
Audience: Core Working Group

Abstract

We propose a standard facility providing the semantics of existing compiler built-ins such as `__builtin_assume` (Clang) and `__assume` (MSVC, ICC). It gives the programmer a way to allow the compiler to assume that a given C++ expression is true, without evaluating it, and to optimise based on this assumption. This is very useful for high-performance and low-latency applications in order to generate both faster and smaller code.


DANGER
BEAR
TRAP

```
[[assume(expr)]];
// assume that expr == true, without checking it,
// and optimise based on that assumption.
```

```
[[assume(expr)]];
// assume that expr == true, without checking it,
// and optimise based on that assumption.
// If expr does not evaluate to true, you get UB.
```

```
int divide_by_32(int x) {
  return x/32;
}
```

```
// compiler output on x86:
mov eax, edi
sar eax, 31
shr eax, 27
add eax, edi
sar eax, 5
ret
```

```
int divide_by_32(int x) {
  return x/32;
}


// compiler output on x86:
mov eax, edi
sar eax, 31
shr eax, 27
add eax, edi
sar eax, 5
ret
```

```
int divide_by_32(int x) {
  [[assume(x >= 0)]];
  return x/32;
}


// compiler output on x86:
mov eax, edi
shr eax, 5
ret
```

```
void limiter(float* data, size_t size) {
  for (size_t i = 0; i < size; ++i)
    data[i] = std::clamp(data[i], -1.0f, 1.0f);
}
```

```
void limiter(float* data, size_t size) {
  [[assume(size > 0)]];
  [[assume(size % 32 == 0)]];
  for (size_t i = 0; i < size; ++i)
    [[assume(std::isfinite(data[i]))]];
    data[i] = std::clamp(data[i], -1.0f, 1.0f);
}
```

```
int f(int y) {
  [[assume(++y == 43)]];
  return y;
}
```

```
int f(int y) {
  [[assume(++y == 43)]];
  return y;
}

// can be optimised to:
int f(int) {
  return 42;
}
```

`[[assume]]`

- Syntax:
  - Argument: [[assume(*expr*)]]; where expr convertible to bool
- Availability:
  - since C++23
  - GCC13
  - MSVC and ICC have `__assume`, Clang has `__builtin_assume`
- Recommendation:
  - Expert-only usage (should be very rare)
  - `expr == true` *must* be an invariant in your code, otherwise UB
  - Consider using only if it leads to a *measurable* perf improvement

Document: P2646R0
Date: 2022-OCT-15
Project: Programming Language C++
Audience: EWG
Reply-to: Parsa Amini: me@parsaamini.net
Joshua Berne: jberne4@bloomberg.net
John Lakos: jlakos@bloomberg.net

Explicit Assumption Syntax *Can* Reduce *Run Time*

Abstract

Many compilers provide platform-specific *assumption* syntax, such as `__builtin_assume` in Clang or idiomatic use of `__builtin_unreachable()` in GCC. This augmented syntax can then indicate to the compiler that it is allowed but not required to assume that some condition — typically a Boolean-valued expression — is always true. Recently, after due consideration, the `[[assume]]` attribute was formally adopted into the C++ working draft (P1774R8) to provide a facility for expressing such assumptions *portably* in source code. As is well known and easily demonstrated, the use of such *compiler-accessible* assumption constructs can noticeably affect compile times as well as object-code and overall program sizes. On the other hand, some members of the C++ Standards Committee have suggested (wrongly) that modern compilers and CPUs conspire to realize essentially all runtime performance benefits available on modern architectures, thereby obviating use of explicit assumption constructs in source code.

[[no_unique_address]]

```
struct Empty {};
static_assert(sizeof(Empty) == 1);
```

```
struct Empty {};
static_assert(sizeof(Empty) == 1);

struct Derived : Empty {
  int i = 0;
};
static_assert(sizeof(Derived) == sizeof(int));
```

```
struct Empty {};
static_assert(sizeof(Empty) == 1);

struct Composed {
  Empty e;
  int i = 0;
};
static_assert(sizeof(Composed) == sizeof(int)); // fail
```

```
struct Empty {};
static_assert(sizeof(Empty) == 1);

struct Composed {
  [[no_unique_address]] Empty e;  // "potentially overlapping subobject"
  int i = 0;
};
static_assert(sizeof(Composed) == sizeof(int)); // OK if supported
                                                // by compiler
```

Why is [[no_unique_address]] odd?

- Modifies the class layout – but only optionally
- Adding it is a potential ABI break
- Not every major compiler supports it (MSVC doesn't)

Why is [[no_unique_address]] odd?

- Modifies the class layout – but only optionally
- Adding it is a potential ABI break
- Not every major compiler supports it (MSVC doesn't)
- "Remove it and declaration still makes sense" does not work:

```
struct Empty {};
struct Composed {
  [[no_unique_address]] Empty e;
  int i = 0;
};
static_assert(sizeof(Composed) == sizeof(int));
```

[[no_unique_address]]

- Syntax:
  - Appertains to a non-static data member
- Availability:
  - since C++20
  - GCC, Clang, and ICC (not enabled in MSVC by default)
- Recommendation:
  - Use if binary size is critical
  - Don't rely on the effects being reliable or portable

- What are standard attributes?
- History of standardisation
- Standard attributes in C++
  - Classification
  - Usage
  - Caveats
  - Availability in major compilers
- Standard attributes in C
- Are standard attributes "ignorable"?
  - Syntactic ignorability
  - Semantic ignorability
  - Language design rule
- Future work

```
// C23
[[noreturn]]
[[deprecated]]
[[fallthrough]]
[[nodiscard]]
[[maybe_unused]]
[[reproducible]]
[[unsequenced]]
```

```
// C23
[[noreturn]]
[[deprecated]]
[[fallthrough]]
[[nodiscard]]
[[maybe_unused]]
[[reproducible]]
[[unsequenced]]
```

```
// C23
[[noreturn]]
[[deprecated]]
[[fallthrough]]
[[nodiscard]]
[[maybe_unused]]
[[reproducible]]
[[unsequenced]]
```

[[reproducible]]
[[unsequenced]]

Unsequenced functions

Étienne Alepins (Thales Canada) and Jens Gustedt (INRIA France)

org:	ISO/IEC JCT1/SC22/WG14	document:	N2887
target:	IS 9899:2023	version:	4
date:	2021-12-31	license:	CC BY

1. Revision history

Paper number	Title	Changes	
N2477	Const functions	Initial version	
N2539	Unsequenced functions	Supersedes N2477	WG14 poll: 15-0-1
			new wording (extracted from N2522)
			no application to the C library
N2825	Unsequenced functions v3	Supersedes N2539	
			no attribute verification imposed
			support for function pointers
			optional text for inclusion of lambdas
N2887	Unsequenced functions v4	Supersedes N2825	refactoring of the properties
			regroup properties in general text
			attach properties to evaluations instead of syntax
			add a sentence to the wording for composite types
			editorial adjustments are collected in a note to the editors at the end
			emphasize on the relationship with existing implementations
			withdraw the special treatment of `call_once`

[[reproducible]] ≈ [[gnu::pure]]

[[unsequenced]] ≈ [[gnu::const]]

Sub-properties of reproducible & unsequenced

- **stateless:** function that does not define mutable static or thread-local objects (nor do functions that are called by it)
- **effectless:** function that does not have observable side effects
- **idempotent:** repeated evaluation gives the same result (hence may read global state)
- **independent:** does not depend on other state than the arguments or constants (hence may write to globals)

Sub-properties of reproducible & unsequenced

- **stateless:** function that does not define mutable static or thread-local objects (nor do functions that are called by it)
- **effectless:** function that does not have observable side effects
- **idempotent:** repeated evaluation gives the same result (hence may read global state)
- **independent:** does not depend on other state than the arguments or constants (hence may write to globals)
- **reproducible:** effectless and idempotent
- **unsequenced**: stateless, effectless, idempotent, and independent

```
double cos(double x);

if ((cos(angle1) > cos(angle2)) || (cos(angle1) > cos(angle3))) {
    ...
}
```

```
double cos(double x) [[unsequenced]];

if ((cos(angle1) > cos(angle2)) || (cos(angle1) > cos(angle3))) {
    ...
}
```

```
double cos(double x) [[unsequenced]];

if ((cos(angle1) > cos(angle2)) || (cos(angle1) > cos(angle3))) {
    ...
}
```

[[reproducible]], [[unsequenced]]

- The compiler is *not* required to check the declared property
- In general, it will be assumed and optimised on
- If the assumption does not hold, you get UB

[[reproducible]], [[unsequenced]]

- Syntax:
  - Appertain to function declarations
- Availability:
  - since C23 (C only – for now...)
  - No implementations yet afaik
  - But GCC and Clang have [[gnu::pure]] and [[gnu::const]]
- Recommendation:
  - Use if you're sure what you're doing
  - If you get it wrong, you get UB

- What are standard attributes?
- History of standardisation
- Standard attributes in C++
  - Classification
  - Usage
  - Caveats
  - Availability in major compilers
- Standard attributes in C
- Are standard attributes "ignorable"?
  - Syntactic ignorability
  - Semantic ignorability
  - Language design rule
- Future work

Towards support for attributes in C++ (Revision 6)

Jens Maurer, Michael Wong
jens.maurer@gmx.net
michaelw@ca.ibm.com

Document number: N2761
Date: 2008-09-18
Project: Programming Language C++, Core Working Group
Reply-to: Michael Wong (michaelw@ca.ibm.com)
Revision: 6

General Attributes for C++

1 Overview

The idea is to be able to annotate some entities in C++ with additional information. Currently, there is no means to do that short of inventing a new keyword and augmenting the grammar accordingly, thereby reserving yet another name of the user's namespace. This proposal will survey existing industry practice for extending the C++ syntax, and presents a general means for such annotations, including its integration into the C++ grammar. Specific attributes are not introduced in this proposal. It does not obviate the ability to add or overload keywords where appropriate, but it does reduce such need and add an ability to extend the language. This proposal will allow many C++0x proposals to move forward. A draft form of this proposal was presented in Oxford and received

Certainly, we would advise anyone who propose an attribute to consider comments on the following area which will help guide them in making the decision of whether to use attributes or not:

- The feature is used in declarations or definitions only.
- Is the feature is of use to a limited audience only (e.g., alignment)?
- The feature does not modify the type system (e.g., thread_local) and hence does not require new mangling?
- The feature is a "minor annotation" to a declaration that does not alter its semantics significantly. (Test: Take away the annotation. Does the remaining declaration still make sense?
- Is it a vendor-specific extension?
- Is it a language Bindings on C++ that has no other way of tying to a type or scope(e.g. OpenMP)
- How does this change Overload resolution?
- What is the effect in typedefs, will it require cloning?

On the ignorability of standard attributes

Timur Doumler (papers@timur.audio)

Document #: P2552R1
Date: 2022-11-15
Project: Programming Language C++
Audience: Evolution Working Group, Core Working Group

Abstract

There is a general notion in C++ that standard attributes should be *ignorable*. However, currently there does not seem to be a common understanding of what "ignorable" means, and the C++ standard itself is ambiguous on this matter. In this paper, we consider three aspects of ignorability: syntactic ignorability, semantic ignorability, and the behaviour of `__has_cpp_attribute`. We discuss where and how the C++ standard is underspecified and why that is problematic, survey existing implementation practice, and propose different options to resolve existing ambiguities.

Question 1:

Is an implementation allowed to ignore a standard attribute?

Question 2:

What does it mean to “ignore” a standard attribute?

C

A strictly conforming program using a standard attribute remains strictly conforming in the absence of that attribute. [...] Standard attributes specified by this document can be parsed but ignored by an implementation without changing the semantics of a correct program; the same is not true for attributes not specified by this document.

C

A strictly conforming program using a standard attribute remains strictly conforming in the absence of that attribute. [...] Standard attributes specified by this document can be parsed but ignored by an implementation without changing the semantics of a correct program; the same is not true for attributes not specified by this document.

C++20

[dcl.attr.grammar]/6

For an *attribute-token* (including an *attribute-scoped-token*) not specified in this document, the behavior is implementation-defined. Any *attribute-token* that is not recognized by the implementation is ignored.

C++20

[dcl.attr.grammar]/6

For an *attribute-token* (including an *attribute-scoped-token*) not specified in this document, the behavior is implementation-defined. Any *attribute-token* that is not recognized by the implementation is ignored.

For an *attribute-token* (including an *attribute-scoped-token*) not specified in this document, the behavior is implementation-defined. Any *attribute-token* that is not recognized by the implementation is ignored.

– Any *attribute-token* not specified in this document?

– Any *attribute-token*, including those specified in this document?

- Syntactic ignorability
- Semantic ignorability

- Syntactic ignorability
- Semantic ignorability

```
int main() {
  [[no_unique_address]] int i;   // Error or ignorable?
};
```

```
int main() {
  [[no_unique_address]] int i;   // Error or ignorable?
  [[assume(a %)]];  // Error or ignorable?
};
```

```
int main() {
  [[no_unique_address]] int i;   // Error or ignorable?
  [[assume(a %)]];  // Error or ignorable?
};

template <typename T>
struct X {
  static_assert(sizeof(T) > 1);
  bool f() { return true; }
};

int main() {
  [[assume(X<char>().f())]]; // Error or ignorable?
}
```

2538. Can standard attributes be syntactically ignored?

Section: 9.12.1 [dcl.attr.grammar] **Status:** open **Submitter:** Jens Maurer **Date:** 2021-12-02 **Liaison:** EWG

Subclause 9.12.1 [dcl.attr.grammar] paragraph 6 specifies that an unrecognized *attribute-token* is ignored:

> For an *attribute-token* (including an *attribute-scoped-token*) not specified in this document, the behavior is implementation-defined. Any *attribute-token* that is not recognized by the implementation is ignored.

The intent is that only non-standard unrecognized *attribute-token*s can be ignored; in particular, an implementation is required to syntax-check all standard attributes, even if the implementation then chooses not to effect any semantics for that attribute.

Proposed resolution (approved by CWG 2022-07-01):

Change in 9.12.1 [dcl.attr.grammar] paragraph 6 as follows:

> For an *attribute-token* (including an *attribute-scoped-token*) not specified in this document, the behavior is implementation-defined~~. Any~~ **; any such** *attribute-token* that is not recognized by the implementation is ignored. **[Note: A program is ill-formed if it contains an *attribute* specified in 9.12 [dcl.attr] that violates the rules to which entity or statement the attribute may apply or the syntax rules for the attribute's *attribute-argument-clause*, if any. -- end note]**

- Syntactic ignorability – no
- Semantic ignorability?

Semantic ignorability in C++ today is implicit

- Attribute only produces diagnostics (*"Recommended practice"*)

  `deprecated, fallthrough, maybe_unused, nodiscard`

- Attribute is an optimisation hint (*"Recommended practice"*)

  `likely, unlikely, noreturn, assume`

- Attribute turns well-defined into undefined behaviour

  `noreturn, assume`

- Attribute has explicitly optional semantics ("*potentially-overlapping subobject*")

  `no_unique_address`

What is semantic ignorability?

- "A program has the same behaviour/semantics with or without the attribute"

What is semantic ignorability?

- "A program has the same behaviour/semantics with or without the attribute"
  → No!

```
[[noreturn]] int f() { return 0; }
int main() { return f(); }
```

What is semantic ignorability?

- "Given a well-formed program with well-defined behaviour, omitting an attribute does not change the behaviour/semantics"

What is semantic ignorability?

- "Given a well-formed program with well-defined behaviour, omitting an attribute does not change the behaviour/semantics"
  → No!

```
struct X {};
struct Y {
  [[no_unique_address]] X x;
  int i;
};

int main() { return (sizeof(Y) == sizeof(int)); }
```

What is semantic ignorability?

- "Given a well-formed program with well-defined behaviour, omitting an attribute does not change the behaviour/semantics"
  → No!

```
struct X {};
struct Y {
  [[no_unique_address]] X x;
  int i;
};

static_assert(sizeof(Y) == sizeof(int));
```

What is semantic ignorability?

- "Given a well-formed program with well-defined behaviour, omitting an attribute does not change the behaviour/semantics"
  → No!

```
struct X {};
struct Y {
  [[no_unique_address]] X x;
  int i;
};

static_assert(sizeof(Y) == sizeof(int));
```

Our proposed rule of semantic ignorability

- "Given a well-formed program, removing all instances of a particular attribute results in a program whose observable behaviour is a conforming realisation of the original program."

Our proposed rule of semantic ignorability

- "Given a well-formed program, removing all instances of a particular attribute results in a program whose observable behaviour is a conforming realisation of the original program."

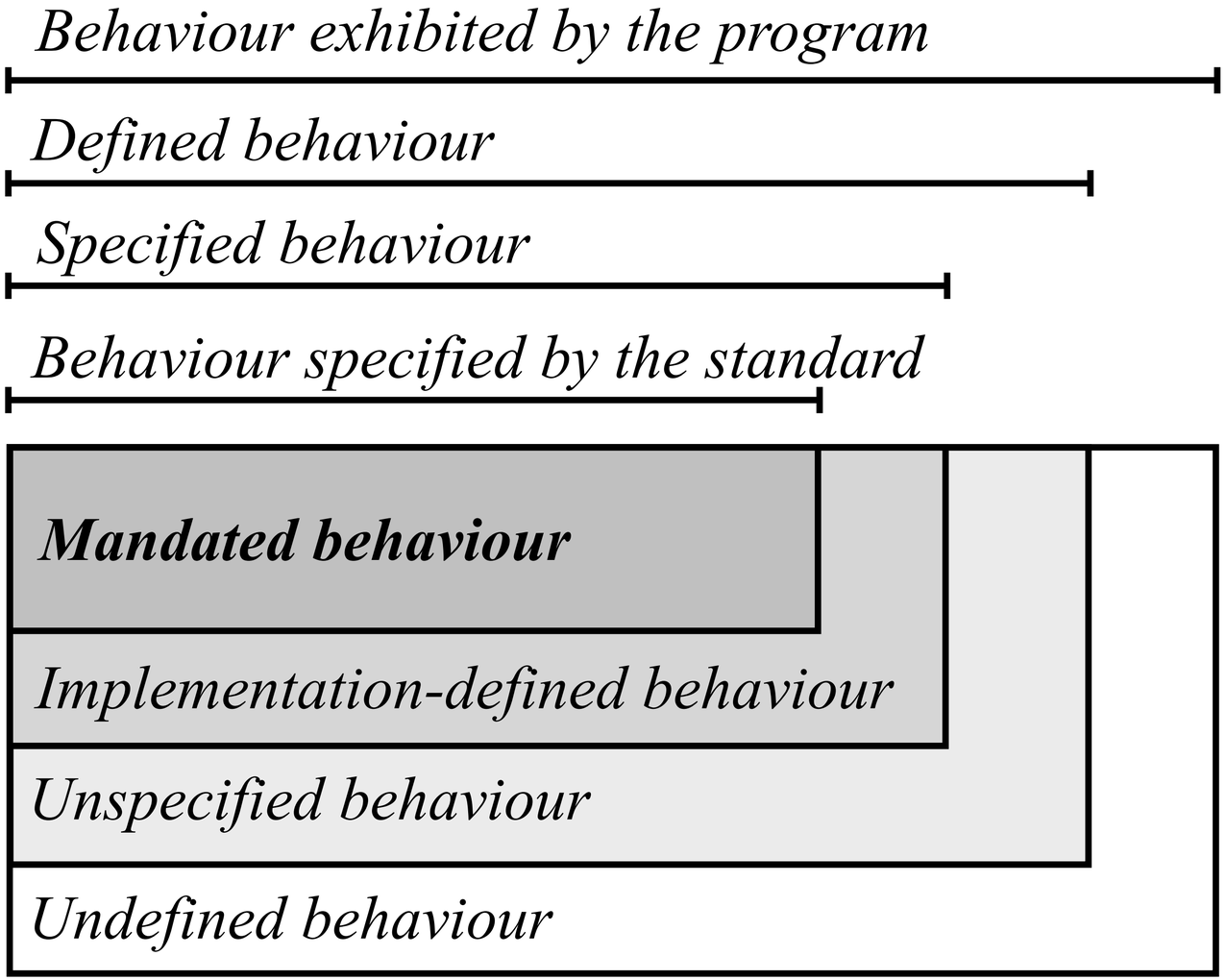
[dcl.attr.grammar] p6

6 For an *attribute-token* (including an *attribute-scoped-token*) not specified in this document, the behavior is implementation-defined; any such *attribute-token* that is not recognized by the implementation is ignored.

[*Note 4*: A program is ill-formed if it contains an *attribute* specified in [dcl.attr] that violates the rules specifying to which entity or statement the attribute can apply or the syntax rules for the attribute's *attribute-argument-clause*, if any. — *end note*]

[*Note 5*: The *attributes* specified in [dcl.attr] have optional semantics: given a well-formed program, removing all instances of any one of those *attributes* results in a program whose set of possible executions ([intro.abstract]) for a given input is a subset of those of the original program for the same input, absent implementation-defined guarantees with respect to that *attribute*. — *end note*]

Levels of program behaviour



Our proposed rule of semantic ignorability

- "Given a well-formed program, removing all instances of a particular attribute results in a program whose observable behaviour is a conforming realisation of the original program."

Needs to be fixed:

```
__has_cpp_attribute
```

```
#if __has_cpp_attribute(assume)
  #define ASSUME(expr) [[assume(expr)]]
#elif defined(__clang__)
  #define ASSUME(expr) __builtin_assume(expr)
#elif defined(_MSC_VER) || defined(__ICC)
  #define ASSUME(expr) __assume(expr)
#elif defined(__GNUC__)
  #define ASSUME(expr) if (expr) {} else { __builtin_unreachable(); }
#else
  #define ASSUME(expr) if (expr) {} else { *(char*)nullptr; }
#endif
```

Should `__has_cpp_attribute` have a non-zero value for an "ignored" attribute?

[cpp.cond]/6

For an attribute specified in this document, the value of the *has-attribute-expression* is given by Table 22. For other attributes recognized by the implementation, the value is implementation-defined.

Attribute	Value
carries_dependency	200809L
deprecated	201309L
fallthrough	201603L
likely	201803L
maybe_unused	201603L
no_unique_address	201803L
nodiscard	201907L
noreturn	200809L
unlikely	201803L

Should `__has_cpp_attribute` have a non-zero value for an "ignored" attribute?

[cpp.cond]/6

For an attribute specified in this document, the value of the *has-attribute-expression* is given by Table 22. For other attributes recognized by the implementation, the value is implementation-defined.

[cpp.cond]/5

Each *has-attribute-expression* is replaced by a non-zero *pp-number* matching the form of an *integer-literal* if the implementation supports an attribute with the name specified by interpreting the *pp-tokens*, after macro expansion, as an *attribute-token*, and by 0 otherwise.

Does the attribute "do anything"?

	Clang	GCC	ICC	MSVC
carries_dependency	❌	❌	❌	❌
deprecated	✅	✅	✅	✅
fallthrough	✅	✅	❓	✅
likely/unlikely	✅	✅	❓	❓
maybe_unused	✅	✅	✅	✅
no_unique_address	✅	✅	✅	❌
nodiscard	✅	✅	✅	✅
noreturn	✅	✅	✅	✅

Is `__has_cpp_attribute > 0`*?*

	Clang	GCC	ICC	MSVC
carries_dependency	✅	❌	✅	✅
deprecated	✅	✅	✅	✅
fallthrough	✅	✅	✅	✅
likely/unlikely	✅	✅	✅	✅
maybe_unused	✅	✅	✅	✅
no_unique_address	✅	✅	✅	❌
nodiscard	✅	✅	✅	✅
noreturn	✅	✅	✅	✅

Does the attribute "do anything"?

	Clang	GCC	ICC	MSVC
carries_dependency	❌	❌	❌	❌
deprecated	✅	✅	✅	✅
fallthrough	✅	✅	❓	✅
likely/unlikely	✅	✅	❓	❓
maybe_unused	✅	✅	✅	✅
no_unique_address	✅	✅	✅	❌
nodiscard	✅	✅	✅	✅
noreturn	✅	✅	✅	✅

Is `__has_cpp_attribute > 0`*?*

	Clang	GCC	ICC	MSVC
carries_dependency	✅	❌	✅	✅
deprecated	✅	✅	✅	✅
fallthrough	✅	✅	✅	✅
likely/unlikely	✅	✅	✅	✅
maybe_unused	✅	✅	✅	✅
no_unique_address	✅	✅	✅	❌
nodiscard	✅	✅	✅	✅
noreturn	✅	✅	✅	✅

Does the attribute "do anything"?

	Clang	GCC	ICC	MSVC
carries_dependency	❌	❌	❌	❌
deprecated	✅	✅	✅	✅
fallthrough	✅	✅	❓	✅
likely/unlikely	✅	✅	❓	❓
maybe_unused	✅	✅	✅	✅
no_unique_address	✅	✅	✅	❌
nodiscard	✅	✅	✅	✅
noreturn	✅	✅	✅	✅

Is `__has_cpp_attribute > 0`*?*

	Clang	GCC	ICC	MSVC
carries_dependency	✅	❌	✅	✅
deprecated	✅	✅	✅	✅
fallthrough	✅	✅	✅	✅
likely/unlikely	✅	✅	✅	✅
maybe_unused	✅	✅	✅	✅
no_unique_address	✅	✅	✅	❌
nodiscard	✅	✅	✅	✅
noreturn	✅	✅	✅	✅

```
#if __has_cpp_attribute(assume)
  #define ASSUME(expr) [[assume(expr)]]
#elif defined(__clang__)
  #define ASSUME(expr) __builtin_assume(expr)
#elif defined(_MSC_VER) || defined(__ICC)
  #define ASSUME(expr) __assume(expr)
#elif defined(__GNUC__)
  #define ASSUME(expr) if (expr) {} else { __builtin_unreachable(); }
#else
  #define ASSUME(expr) if (expr) {} else { *(char*)nullptr; }
#endif
```

Does the attribute "do anything"?

	Clang	GCC	ICC	MSVC
carries_dependency	❌	❌	❌	❌
deprecated	✅	✅	✅	✅
fallthrough	✅	✅	❓	✅
likely/unlikely	✅	✅	❓	❓
maybe_unused	✅	✅	✅	✅
no_unique_address	✅	✅	✅	❌
nodiscard	✅	✅	✅	✅
noreturn	✅	✅	✅	✅

Is `__has_cpp_attribute > 0`*?*

These should be compiler bugs!

	Clang	GCC	ICC	MSVC
carries_dependency	✅	❌	✅	✅
deprecated	✅	✅	✅	✅
fallthrough	✅	✅	✅	✅
likely/unlikely	✅	✅	✅	✅
maybe_unused	✅	✅	✅	✅
no_unique_address	✅	✅	✅	❌
nodiscard	✅	✅	✅	✅
noreturn	✅	✅	✅	✅

On the ignorability of standard attributes

Timur Doumler (papers@timur.audio)

Document #: D2552R2
Date: 2023-04-19
Project: Programming Language C++
Audience: Evolution Working Group, Core Working Group

Abstract

There is a general notion in C++ that standard attributes should be *ignorable*. However, currently there does not seem to be a common understanding of what "ignorable" means, and the C++ standard itself is ambiguous on this matter. In this paper, we consider three aspects of ignorability: syntactic ignorability, semantic ignorability, and the behaviour of `__has_cpp_attribute`. We discuss where and how the C++ standard is underspecified and why that is problematic, survey existing implementation practice, and propose different options to resolve existing ambiguities.

P2552R2

- will propose that `__has_cpp_attribute` should have a positive value only if the attribute is *supported* in the sense of *"the compiler makes its best effort to provide the optional semantics".*

```
#if __has_cpp_attribute(assume)
  #define ASSUME(expr) [[assume(expr)]]
#elif defined(__clang__)
  #define ASSUME(expr) __builtin_assume(expr)
#elif defined(_MSC_VER) || defined(__ICC)
  #define ASSUME(expr) __assume(expr)
#elif defined(__GNUC__)
  #define ASSUME(expr) if (expr) {} else { __builtin_unreachable(); }
#else
  #define ASSUME(expr) if (expr) {} else { *(char*)nullptr; }
#endif
```

- What are standard attributes?
- History of standardisation
- Standard attributes in C++
  - Classification
  - Usage
  - Caveats
  - Availability in major compilers
- Standard attributes in C
- Are standard attributes "ignorable"?
  - Syntactic ignorability
  - Semantic ignorability
  - Language design rule
- Future work

Future standard committee work

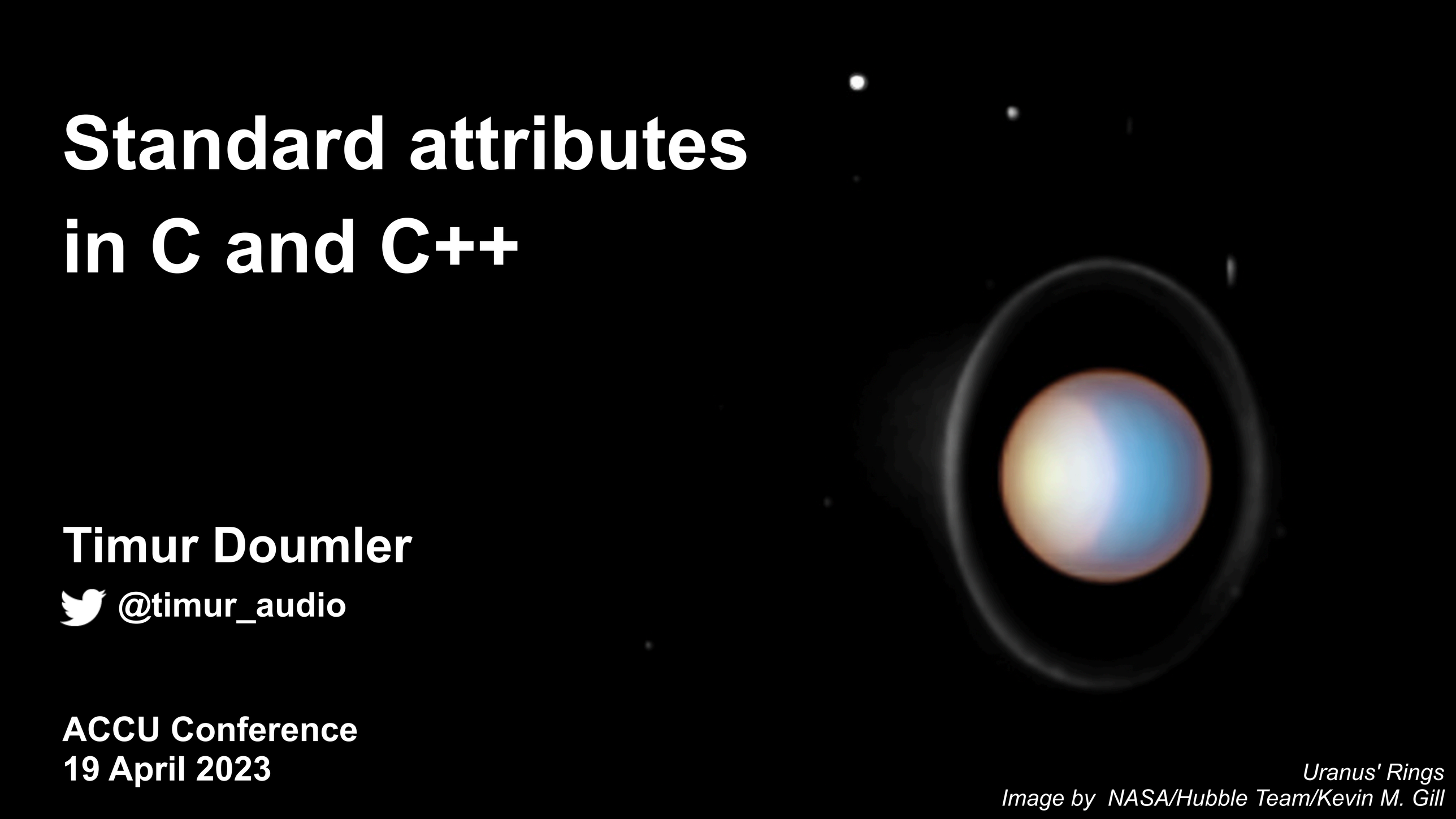
- Fix `__has_cpp_attribute` (P2552R2)
- Get `[[assume]]` into C
- Get `[[unsequenced]]` and `[[reproducible]]` into C++26
- Propose `[[noalias]]` for C++26

Standard attributes in C and C++

Timur Doumler

@timur_audio

ACCU Conference
19 April 2023



Uranus' Rings
Image by NASA/Hubble Team/Kevin M. Gill


The international
C++ conference
in the UK, by the sea
28th - 30th June 2023
Workshops: 27th June

C++ and Safety

Timur Doumler

60 minute session | beginner | intermediate | advanced

13:45-14:45, Thursday, 29th June 2023

Organisations such as the National Security Angency (NSA) and the National Institute of Standards and Techology (NIST) are currently urging developers to move away from programming languages that are not memory safe. C++ is arguably not a "safe" programming language in its current form. Why is that? And should we do anything about it? If yes, what, and how? Have we arrived at a crossroads for the future evolution of C++? What does "safety" even mean, and how is it different from "security" and "correctness"?

In this talk, we attempt to give useful definitions for these terms. For safety in particular, we can distinguish between functional safety and language safety, and identify different aspects of language safety (of which memory safety is one). We discuss how and why C++ is considered "unsafe" and what consequences follow from that for different domains and use cases. We look at how other programming languages, such as Java, Rust, and Val avoid such safety issues, what tradeoffs are involved in these strategies, and why we can't easily adopt any of them for C++. We consider the tooling available today to mitigate safety issues in C++, such as sanitisers and static analysers, and their limitations. Finally, we look at the future evolution of C++ and discuss the current work on C++ Contracts and other recent proposals targeted at making C++ more safe.

The C++ Undefined Behaviour Survey

- 3 simple questions (one of which is optional)
- anonymous

The C++ Undefined Behaviour Survey

- 3 simple questions (one of which is optional)
- anonymous
- https://timur.audio/survey