Modern C++ Programming

4. Basic Concepts III

Entities, Control Flow, Namespaces, and Attributes

Federico Busato

2025-01-19

Table of Contents

Entities

Entities

A C++ program is set of language-specific *keywords* (`for`, `if`, `new`, `true`, etc.), *identifiers* (symbols for variables, functions, structures, namespaces, etc.), *expressions* defined as sequence of operators, and *literals* (constant value tokens)

C++ Entity

An **entity** is a value, object, reference, function, enumerator, type, class member, or template

Identifiers and *user-defined operators* are the names used to refer to *entities*

Entities also captures the result(s) of an *expression*

Preprocessor macros are not C++ entities

Declaration and Definition

Declaration/Definition

Declaration/Prototype

A **declaration** (or *prototype*) introduces an *entity* with an *identifier* describing its type and properties

A *declaration* is what the compiler and the linker needs to accept references (usage) to that identifier

Entities can be declared multiple times. All declarations are the same

Definition/Implementation

An entity **definition** is the implementation of a declaration. It defines the properties and the behavior of the entity

For each entity, only a single *definition* is allowed

Declaration/Definition Function Example

```
void f(int a, char* b);   // function declaration

void f(int a, char*) {    // function definition
    ...                   // "b" can be omitted if not used
}

void f(int a, char* b);   // function declaration
                          // multiple declarations is valid

f(3, "abc");              // usage
```

```
void g();   // function declaration

g();        // linking error "g" is not defined
```

Declaration/Definition `struct` Example

A declaration without a concrete implementation is an incomplete type (as `void`)

```
struct A;    // declaration 1
struct A;    // declaration 2 (ok)

struct B {   // declaration and definition
    int b;
//  A   x;   // compile error incomplete type
    A*  y;   // ok, pointer to incomplete type
};

struct A {   // definition
    char c;
}
```

Enumerators

Enumerator

An **enumerator** `enum` is a data type that groups a set of named integral constants

```
enum color_t { BLACK, BLUE, GREEN };

color_t color = BLUE;
cout << (color == BLACK); // print false
```

The problem:

```
enum color_t { BLACK, BLUE, GREEN };
enum fruit_t { APPLE, CHERRY };

color_t color = BLACK;    // int: 0
fruit_t fruit = APPLE;    // int: 0
bool    b     = (color == fruit); // print 'true'!!
// and, most importantly, does the match between a color and
// a fruit make any sense?
```

Strongly Typed Enumerator - `enum class`

`enum class` (C++11)

`enum class` (scoped enum) data type is a *type safe* enumerator that is not implicitly convertible to `int`

```
enum class Color { BLACK, BLUE, GREEN };
enum class Fruit { APPLE, CHERRY };

Color color = Color::BLUE;
Fruit fruit = Fruit::APPLE;

// bool b = (color == fruit) compile error we are trying to match colors with fruits
//                           BUT, they are different things entirely
// int a1 = Color::GREEN;              compile error
// int a2 = Color::RED + Color::GREEN; compile error
   int a3 = (int) Color::GREEN;        // ok, explicit conversion
```

`enum/enum class` Features

- `enum/enum class` can be compared

```
enum class Color   { RED, GREEN, BLUE };
cout << (Color::RED < Color::GREEN); // print true
```

- `enum/enum class` are automatically enumerated in increasing order

```
enum class Color { RED, GREEN = -1, BLUE, BLACK };
//                 (0)  (-1)         (0)   (1)
Color::RED == Color::BLUE; // true
```

- `enum/enum class` can contain alias

```
enum class Device { PC = 0, COMPUTER = 0, PRINTER };
```

- C++11 `enum/enum class` allows setting the underlying type

```
enum class Color : int8_t { RED, GREEN, BLUE };
```

`enum class` Features - C++17

- C++17 `enum class` supports *direct-list-initialization*

```
enum class Color { RED, GREEN, BLUE };
Color a{2}; // ok, equal to Color:BLUE
```

- C++17 `enum/enum class` support *attributes*

```
enum class Color { RED, GREEN, BLUE [[deprecated]] };
auto x = Color::BLUE; // compiler warning
```

`enum class` Features - C++20

- C++20 allows introducing the enumerator identifiers into the local scope to decrease the verbosity

```
enum class Color { RED, GREEN, BLUE };

switch (x) {
    using enum Color; // C++20
    case RED:
    case GREEN:
    case BLUE:
}
```

The same behavior can be emulated in older C++ versions with

```
enum class Color { RED, GREEN, BLUE };

constexpr auto RED = Color::RED;
```

`enum/enum class` - Common Errors

- `enum/enum class` should be always initialized

```
enum class Color { RED, GREEN, BLUE };

Color my_color;  // "my_color" may be outside RED, GREEN, BLUE!!
```

- C++17 Cast from *out-of-range values* respect to the *underlying type* of `enum/enum class` leads to undefined behavior

```
enum Color : uint8_t { RED, GREEN, BLUE };

Color value = 256; // undefined behavior
```

enum/enum class and constexpr*

- C++17 `constexpr` expressions don't allow *out-of-range values* for (only) `enum` without explicit *underlying type*

```
enum       Color       { RED };
enum       Fruit : int { APPLE };
enum class Device      { PC };

// constexpr Color  a1 = (Color)  -1; compile error
   const     Color  a2 = (Color)  -1; // ok
   constexpr Fruit  a3 = (Fruit)  -1; // ok
   constexpr Device a4 = (Device) -1; // ok
```

`struct`, Bitfield, and `union`

A `struct` (*structure*) aggregates different variables into a single unit

```
struct A {
    int   x;
    char  y;
};
```

It is possible to declare one or more variables after the definition of a `struct`

```
struct A {
    int x;
} a, b;
```

Enumerators can be declared within a `struct` without a name

```
struct A {
    enum {X, Y}
};
A::X;
```

It is possible to declare a `struct` in a local scope (with some restrictions), e.g. function scope

```
int f() {
    struct A {
        int x;
    } a;
    return a.x;
}
```

Anonymous and Unnamed `struct`★

Unnamed `struct`: a structure without a name, but with an associated type

Anonymous `struct`: a structure without a name and type

The C++ standard allows *unnamed* `struct` but, contrary to C, does not allow *anonymous* `struct` (i.e. without a name)

```
struct {
    int x;
} my_struct;          // unnamed struct, ok

struct S {
    int x;
    struct { int y; }; // anonymous struct, compiler warning with -Wpedantic
};                    // -Wpedantic: diagnose use of non-strict ISO C++ extensions
```

Bitfield

Bitfield

A **bitfield** is a variable of a structure with a predefined bit width. A bitfield can hold bits instead bytes

```
struct S1 {
    int b1 : 10;  // range [0, 1023]
    int b2 : 10;  // range [0, 1023]
    int b3 : 8;   // range [0, 255]
}; // sizeof(S1): 4 bytes

struct S2 {
    int b1 : 10;
    int    : 0;   // reset: force the next field
    int b2 : 10;  //        to start at bit 32
}; // sizeof(S2): 8 bytes
```

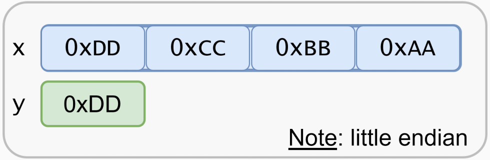
Union

A `union` is a special data type that allows to store different data types in the same memory location

- The `union` is only as big as necessary to hold its *largest* data member
- The `union` is a kind of *"overlapping"* storage

```cpp
union A {
    int  x;
    char y;
};
```

```cpp
A a;
a.x = 0xAABBCCDD
```



```
union A {
    int  x;
    char y;
};  // sizeof(A): 4

A a;
a.x = 1023;    // bits: 00..000001111111111
a.y = 0;       // bits: 00..000001100000000
cout << a.x;  // print 512 + 256 = 768
```

NOTE: Little-Endian encoding maps the bytes of a value in memory in the reverse order. `y` maps to the last byte of `x`

Contrary to `struct`, C++ allows *anonymous* `union` (i.e. without a name)

C++17 introduces `std::variant` to represent a *type-safe union*

Control Flow

`if` Statement

The `if` statement executes the first branch if the specified condition is evaluated to `true`, the second branch otherwise

- *Short-circuiting*:

```
int array[3];
if (<true expression> || array[2345324] == 0)
... // no error!! even though index is out-of-bound
    // left-to-right evaluation
```

- *Ternary operator*:

  `<cond> ? <expression1> : <expression2>`

  `<expression1>` and `<expression2>` must return a value of the same or convertible type

```
int value = (a == b) ? a : (b == c ? b : 3); // nested
```

`for` and `while` Loops

- `for`: use when number of iterations is known

```
for ([init]; [cond]; [increment]) {
    ...
}
```

- `while`: use when number of iterations is not known

```
while (cond) {
    ...
}
```

- `do while`: use when number of iterations is not known, but there is at least one iteration

```
do {
...
} while (cond);
```

`for` Loop Features and Jump Statements

- C++ allows multiple initializations and increments in the declaration:

```
for (int i = 0, k = 0; i < 10; i++, k += 2)
    ...
```

- Infinite loop:

```
for (;;)   // also while(true);
   ...
```

- Jump statements (`break`, `continue`, `return`):

```
for (int i = 0; i < 10; i++) {
    if (<condition>)
        break;    // exit from the loop
    if (<condition>)
        continue; // continue with a new iteration and exec. i++
    return;       // exit from the function
}
```

C++11 introduces the **range-based `for` loop** to simplify the verbosity of traditional `for` loop constructs. They are equivalent to the `for` loop operating over a range of values, but **safer**

The range-based `for` loop avoids the user to specify start, end, and increment of the loop

```
for (int v : { 3, 2, 1 }) // INITIALIZER LIST
    cout << v << " ";      // print: 3 2 1

int values[] = { 3, 2, 1 };
for (int v : values)      // ARRAY OF VALUES
    cout << v << " ";     // print: 3 2 1

for (auto c : "abcd")     // RAW STRING
    cout << c << " ";     // print: a b c d
```

Range-based for loop can be applied in three cases:

- Fixed-size array `int array[3]`, `"abcd"`
- Branch Initializer List `{1, 2, 3}`
- Any object with `begin()` and `end()` methods

```
std::vector vec{1, 2, 3, 4};

for (auto x : vec) {
   cout << x << ", ";
// print:  "1, 2, 3, 4"
```

```
int matrix[2][4];
for (auto& row : matrix) {
    for (auto element : row)
        cout << "@";
    cout << "\n";
}
// print:
//
```

C++17 extends the concept of **range-based loop** for *structure binding*

```
struct A {
    int x;
    int y;
};

A array[] = { {1,2}, {5,6}, {7,1} };
for (auto [x1, y1] : array)
    cout << x1 << "," << y1 << "  "; // print: 1,2  5,6  7,1
```

The `switch` statement evaluates an expression (`int`, `char`, `enum class`, `enum`) and executes the statement associated with the matching case value

```
char x = ...
switch (x) {
    case 'a': y = 1; break;
    default:  return -1;
}
return y;
```

Switch scope:

```
int x = 1;
switch (1) {
    case 0: int x;       // nearest scope
    case 1: cout << x;  // undefined!!
    case 2: { int y; }  // ok
//  case 3: cout << y;  // compile error
```

Fall-through:

```
MyEnum x
int y = 0;
switch (x) {
   case MyEnum::A:        // fall-through
   case MyEnum::B:        // fall-through
   case MyEnum::C: return 0;
   default: return -1;
}
```

C++17 `[[fallthrough]]` attribute

```
char x = ...
switch (x) {
    case 'a': x++;
              [[fallthrough]]; // C++17: avoid warning
    case 'b': return 0;
    default: return -1;
```

Control Flow with Initializing Statement

Control flow with **initializing statement** aims at simplifying complex actions before the condition evaluation and restrict the scope of a variable which is visible only in the control flow body

C++17 introduces `if` statement with initializer

```
if (int ret = x + y; ret < 10)
    cout << ret;
```

C++17 introduces `switch` statement with initializer

```
switch (auto i = f(); x) {
   case 1: return i + x;
```

C++20 introduces `range-for loop` statement with initializer

```
for (int i = 0; auto x : {'A', 'B', 'C'})
    cout << i++ << ":" << x << " ";  // print: 0:A  1:B  2:C
```

When goto could be useful:

```
bool flag = true;
for (int i = 0; i < N && flag; i++) {
    for (int j = 0; j < M && flag; j++) {
        if (<condition>)
            flag = false;
    }
}
```

become:

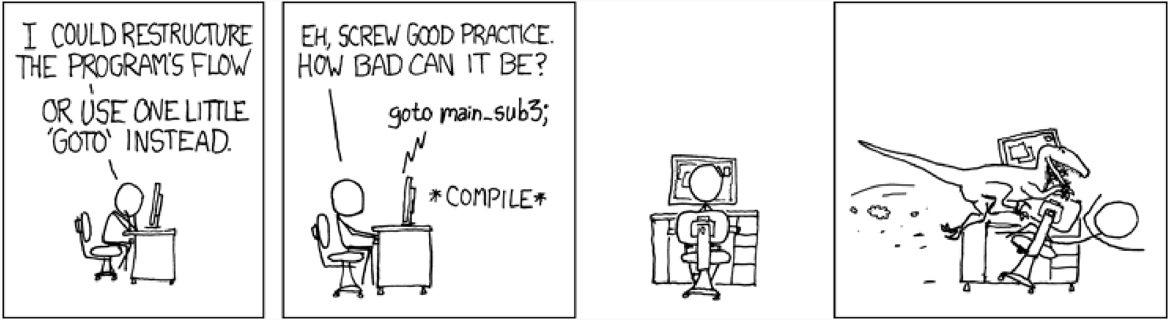
```
for (int i = 0; i < N; i++) {
    for (int j = 0; j < M; j++) {
        if (<condition>)
            goto LABEL;
    }
}
LABEL: ;
```

Best solution:

```
bool my_function(int M, int M) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < M; j++) {
            if (<condition>)
                return false;
        }
    }
    return true;
}
```

Junior: what's wrong with goto command?

goto command:


I COULD RESTRUCTURE THE PROGRAM'S FLOW
OR USE ONE LITTLE 'GOTO' INSTEAD.
EH, SCREW GOOD PRACTICE. HOW BAD CAN IT BE?
goto main_sub3;
COMPILE

Most compilers issue a warning when a variable is unused. There are different situations where a variable is expected to be unused

```
// EXAMPLE 1: macro dependency
int f(int value) {
    int x = value;
#if defined(ENABLE_SQUARE_PATH)
    return x * x;
#else
    return 0;
#endif
}
```

Avoid Unused Variable Warning★

```cpp
// EXAMPLE 2: constexpr dependency (MSVC)
template<typename T>
int f(T value) {
    if constexpr (sizeof(value) >= 4)
        return 1;
    else
        return 2;
}
```

```cpp
// EXAMPLE 3: decltype dependency (MSVC)
template<typename T>
int g(T value) {
    using R = decltype(value);
    return R{};
}
```

Avoid Unused Variable Warning

There are different ways to solve the problem depending on the standard used

- Before C++17: `static_cast<void>(var)`
- C++17 `[[maybe_unused]]` attribute
- C++26 `auto _`

```
[[maybe_unused]] int x = value;
int y  = 3;
static_cast<void>(y);
auto _ = 3;
auto _ = 4; // _ repetition is not an error

void f([[maybe_unused]] int x) {}
```

Namespace

Overview

The problem: Named entities, such as variables, functions, and compound types declared outside any block has *global scope*, meaning that its name/symbol is valid anywhere in the code

`Namespaces` *allow grouping named entities that otherwise would have global scope into narrower scopes, giving them* ***namespace scope***

Namespaces provide a method for preventing name conflicts in large projects. Symbols declared inside a *namespace block* are placed in a *named scope* that prevents them from being mistaken for symbols with identical names

Namespace Syntax

```
namespace [<name>] {

<identifier> // variable, function, struct, type, etc.

} // namespace <name>

<name>::<identifier> // use the identifier
```

The operator `::` is called **scope resolution operator** and it allows accessing identifiers that are defined in other namespaces

Namespace Example 1

```cpp
#include <iostream>
namespace my_namespace1 {
void f() {
    std::cout << "my_namespace1" << std::endl;
}
} // namespace my_namespace1

namespace my_namespace2 {
void f() {
    std::cout << "my_namespace2" << std::endl;
}
} // namespace my_namespace2

int main () {
    my_namespace1::f(); // print "my_namespace1"
    my_namespace2::f(); // print "my_namespace2"
//  f();                // compile error f() is not visible
}
```

Namespace - Alternative Syntax

It is also possible to declare entities in a preexisting namespace by adding the name as a prefix:

```
namespace <name> {}
<name>::<identifier>
```

```
#include <iostream>
namespace my_namespace1 {}

void my_namespace2::f() { std::cout << "my_namespace2" << std::endl; }

int main () {
    my_namespace1::f(); // print "my_namespace1"
}
```

Special Namespaces

- All functionalities and data types provided with the **standard library** (distributed along with the compiler) are declared within the `std` namespace

- The global namespace can be specified with `::identifier` and can be useful to prevent conflicts with surrounding namespaces

- It is also possible to define a namespace without a name. The concept refers to *anonymous (or unnamed) namespace*
  See "Translation Unit I" lecture for more details

Nested Namespaces

```
namespace my_namespace1 {
void f() { cout << "my_namespace1::f()"; }

namespace my_namespace2 {

void f() { cout << "my_namespace1::my_namespace2::f()"; }

} // namespace my_namespace2
} // namespace my_namespace1

my_namespace1::my_namespace2::f();
```

C++17 allows *nested namespace* definitions with a less verbose syntax:

```
namespace my_namespace1::my_namespace2 {
    void h();
}
```

Explicit Global Namespace

The explicit global namespace syntax `::identifier` can be useful to prevent conflicts with surrounding namespaces

```
void f() { cout << "global::f()"; }

namespace my_namespace {

void f() { cout << "my_namespace::f()"; }

void g() {
    f();   // print "my_namespace::f()"
    ::f(); // print "global::f()"
}

} // namespace my_namespace
```

Namespace Alias

Namespace alias allows declaring an alternate name for an existing namespace

```
namespace very_long_namespace {
namespace even_longer {
    void g() {}
} // namespace even_longer
} // namespace very_long_namespace

namespace ns1 = very_long_namespace::even_longer;     // namespace alias

int main() {
    namespace ns2 = very_long_namespace::even_longer; // namespace alias
                                                    // available only in this scope
    ns1::g();
    ns2::g();
}
```

The `using`-*declaration* introduces a specific name/system from a namespace into the current scope. This is useful for improving code readability and reducing verbosity

The `using`-*declaration* is roughly equivalent of declaring the name/system in the current scope

Syntax:

```
namespace <name> {
    <identifier>
}

using <name>::<identifier>;
<identifier>;
```

`using`-Declaration 2/2

```
namespace my_namespace {

void f() { cout << "my_namespace::f()"; }

struct S {};

using T = int;

} // namespace my_namespace

using my_namespace::f;
using my_namespace::S;
using my_namespace::T;
f(); // print "my_namespace::f()"
S s;
T x;
// struct S {}; // compile error "struct S" already defined by my_namespace::T
```

`using namespace`-Directive

The **`using namespace`**-directive introduces all the identifiers in a *scope* without having to specify them explicitly with the namespace name

Similarly to `using`-declaration, it is useful for improving code readability and reducing verbosity. On the other hand, it could make the code bug-prone because of the complex name lookup rules, especially if coupled with function overloadding

It is generally recommended not to write `using namespace`, especially at the global level. Otherwise, it defeats the purpose of the namespace

using namespace-Directive

```
namespace my_namespace {

void f() { cout << "my_namespace::f()"; }

struct S {};

} // namespace my_namespace

int main () {
    using namespace my_namespace;
    f();        // print "my_namespace::f()"
    S s;
}
```

using namespace-Directive vs. using-declaration

```
namespace A { int x = 0; }

namespace B {
    int y = 3;
    int x = 7;
}
int main () {
    using namespace A;
    int x = 3;   // ok!! even if it is already defined in my_namespace
    using B::y;
//  int y = 5;   // compiler error!! "y" is already defined in this scope
}
void f() {
    using B::x;
    using namespace A;
    cout << x;   // print 7, B::x has higher priority
}
```

`using namespace`-directive has the transitive property for its identifiers when used into another namespace

```cpp
namespace A {
    void f() { cout << "A::f()"; }
}

namespace B {
    using namespace A;
}

int main() {
    using namespace B;
    f(); // ok, print "A::f()"
}
```

`using namespace`-Directive Transitive Property ★ 2/3

The **unqualified name lookup** is the mechanism by which the compiler searches for the declaration of an identifier without using any explicit scope qualifiers like the `::` operator

Unqualified name lookup and `using namespace`-Directive:

Every name from `namespace-name` is visible as if it is declared in the nearest enclosing namespace which contains both the `using`-directive and `namespace-name`

using namespace-Directive Transitive Property ★ 3/3

```cpp
namespace A { int i = 0; }

namespace C {

int i = 3;
namespace B {
    using namespace A; // unqualified name lookup of A within B:
    int x = i;         // it is the nearest enclosing namespace which contains
} // namespace B       // both A and B -> global namespace
                       // "int x = i" -> "int x = C::i" because C has higher
} // namespace C       // precedence than the global namespace

int main() {
    using namespace B;
    cout << C::B::x;   // print "3"
}
```

`inline` Namespace ★

`inline` namespace is a concept similar to library versioning. It is a mechanism that makes a nested namespace look and act as if all its declarations were in the surrounding namespace

```
namespace my_namespace1 {

inline namespace V99 { void f(int) {} }  // most recent version
namespace        V98 { void f(int) {} }

} // namespace my_namespace1

using namespace my_namespace1;
V98::f(1);  // call V98
V99::f(1);  // call V99
f(1);       // call default version (V99)
```

Attributes ★

C++ attributes provide additional information to the compiler to enforce constraints or enable code optimization

Attributes are *annotation* on top of standard code that can be applied to functions, variables, classes, enumerator, types, etc.

C++11 introduces a standardized syntax for *attributes*: `[[my-attribute]]`

```
__attribute__((always_inline))  // < C++11, GCC/Clang/GNU compilers
__forceinline                   // < C++11, MSVC

[[gnu::always_inline]]          // C++11, GCC/Clang/GNU compilers
[[msvc::forceinline]]           // C++11, MSVC
```

In addtion, C++11 and later add *standard attributes* such as `maybe_unused`, `deprecated`, `nodiscard` for functions

C++23 adds the such attributes to lambda expressions

```
auto lambda = [] [[nodiscard]] (){ return 4; };

lambda();           // compiler warning
auto x = lambda(); // ok
```

C++17 introduces the attribute `[[nodiscard]]` to issue a warning if the return value of a function is discarded (not handled)

C++20 extends the attribute by allowing to add a reason
`[[nodiscard("reason")]]`

```cpp
[[nodiscard]] bool empty();

empty(); // WARNING "discard return value"
```

`[[nodiscard]]` Attribute

`[[nodiscard]]` can be also be applied to enumerators `enum` / `enum class` and structures `struct` / `class`

```
enum class [[nodiscard]] MyEnum { EnumValue };

struct [[nodiscard]] MyStruct {};

MyEnum f() { return MyEnum::EnumValue; }

MyStruct g() {
    MyStruct s;
    return s;
}

f(); // WARNING "discard return value"
g(); // WARNING "discard return value"
```

`[[nodiscard]]` can be also be applied to class constructors

```
MyStruct g() {
    [[nodiscard]] MyStruct() {}
    [[nodiscard]] MyStruct(const MyStruct&) {}
}

MyStruct{};                 // WARNING "discard return value"
MyStruct s{};
static_cast<MyStruct>(s); // WARNING "discard return value" for
                            // MyStruct(const MyStruct&)
```

`[[maybe_unused]]` Attribute

`[[maybe_unused]]` applies to

- Variables
- Structure binding
- Functions parameters and return value
- Types
- Classes and structures
- Enumerators and single value enumerators

[[maybe_unused]] Attribute 2/2

```cpp
[[maybe_unused]] int x1;

[[maybe_unused]] auto [x2, x3] = ...;

[[maybe_unused]] int f([[maybe_unused]] int x4);

struct [[maybe_unused]] S {};

using MyInt [[maybe_unused]] = int;

enum [[maybe_unused]] Enum {
     E1 [[maybe_unused]];
};

enum class [[maybe_unused]] EnumClass {
     E2 [[maybe_unused]];
};
```

`[[deprecated]]` Attribute

C++14 allows to deprecate, namely discourage, use of entities by adding the `[[deprecated]]` attribute, optionally with a message `[[deprecated("reason")]]`. It applies to:

- Functions
- Variables
- Classes and structures
- Enumerators
- Single value enumerator in C++17
- Types
- Namespaces

```cpp
[[deprecated]] void f() {}

struct [[deprecated]] S1 {};

using MyInt [[deprecated]] = int;

struct S2 {
    [[deprecated]] int var = 3;
    [[deprecated]] static constexpr int var2 = 4;
};

f();        // compiler warning
S1    s1; // compiler warning
MyInt i;  // compiler warning
S2{}.var; // compiler warning
S2::var2; // compiler warning
```

C++17 allows to deprecate individual enumerator values

```
enum [[deprecated]] E { EnumValue };             // C++14

enum class MyEnum { A, B [[deprecated]] = 42 }; // C++17

auto x = EnumValue; // compiler warning
MyEnum::B;          // compiler warning
```

C++17 allows defining attribute on namespaces

```
namespace [[deprecated("please use my_namespace_v2")]] my_namespace {

void f() {}

} // namespace my_namespace

my_namespace::f(); // compiler warning
```

[[noreturn]] Attribute

`[[noreturn]]` indicates that a function *does not return* (e.g. program termination) and the compiler should issue a compiler warning if the code contains other statements that cannot be executed because it means a wrong user intention

```
[[noreturn]] void g() { std::exit(0); }

g(); // WARNING: no code should be exectuted after calling this function

y = x + 1;
```