Enumeration declaration

An *enumeration* is a distinct type whose value is restricted to a range of values (see below for details), which may include several explicitly named constants ("*enumerators*"). The values of the constants are values of an integral type known as the *underlying type* of the enumeration.

An enumeration is (re)declared using the following syntax:

enum-key attr(optional) *enum-head-name*(optional) *enum-base*(optional) **{** *enumerator-list*(optional) **}**	(1)	
enum-key attr(optional) *enum-head-name*(optional) *enum-base*(optional) **{** *enumerator-list* **,** **}**	(2)	
enum-key attr(optional) *enum-head-name enum-base*(optional) **;**	(3)	(since C++11)

1) *enum-specifier*, which appears in *decl-specifier-seq* of the declaration syntax: defines the enumeration type and its enumerators.

2) A trailing comma can follow the *enumerator-list*.

3) *Opaque enum declaration*: defines the enumeration type but not its enumerators: after this declaration, the type is a complete type and its size is known.

`enum-key` -

enum	(until C++11)
one of **enum**, **enum class**, or **enum struct**	(since C++11)

`attr` - (since C++11) optional sequence of any number of attributes

`enum-head-name` -

the name of the enumeration that's being declared, it can be omitted.	(until C++11)
the name of the enumeration that's being declared, optionally preceded by a *nested-name-specifier*: sequence of names and scope-resolution operators `::`, ending with scope-resolution operator. It can only be omitted in unscoped non-opaque enumeration declarations. *nested-name-specifier* may only appear if the enumeration name is present and this declaration is a redeclaration. For opaque enumeration declarations, *nested-name-specifier* can only appear before the name of the enumeration in explicit specialization declarations. If *nested-name-specifier* is present, the *enum-specifier* cannot refer to an enumeration merely inherited or introduced by a using-declaration, and the *enum-specifier* can only appear in a namespace enclosing the previous declaration. In such cases, *nested-name-specifier* cannot begin with a decltype specifier.	(since C++11)

`enum-base` - (since C++11) colon (**:**), followed by a *type-specifier-seq* that names an integral type (if it is cv-qualified, qualifications are ignored) that will serve as the fixed underlying type for this enumeration type

`enumerator-list` - comma-separated list of enumerator definitions, each of which is either simply an *identifier*, which becomes the name of the enumerator, or an identifier with an initializer: *identifier* **=** *constexpr*. In either case, the *identifier* can be directly followed by an optional attribute specifier sequence. (since C++17)

There are two distinct kinds of enumerations: *unscoped enumeration* (declared with the *enum-key* **enum**) and *scoped enumeration* (declared with the *enum-key* **enum class** or **enum struct**).

Unscoped enumerations

enum *name*(optional) **{** *enumerator* **=** *constexpr* **,** *enumerator* **=** *constexpr* **,** ... **}**	(1)	
enum *name*(optional) **:** *type* **{** *enumerator* **=** *constexpr* **,** *enumerator* **=** *constexpr* **,** ... **}**	(2)	(since C++11)
enum *name* **:** *type* **;**	(3)	(since C++11)

1) Declares an unscoped enumeration type whose underlying type is not fixed (in this case, the underlying type is an implementation-defined integral type that can represent all enumerator values; this type is not larger than `int` unless the value of an enumerator cannot fit in an `int` or `unsigned int`. If the *enumerator-list* is empty, the underlying type is as if the enumeration had a single enumerator with value 0. If no integral type can represent all the enumerator values, the enumeration is ill-formed).

2) Declares an unscoped enumeration type whose underlying type is fixed.

3) Opaque enum declaration for an unscoped enumeration must specify the name and the underlying type.

Each *enumerator* becomes a named constant of the enumeration's type (that is, *name*), visible in the enclosing scope, and can be used whenever constants are required.

```
enum Color { red, green, blue };
Color r = red;

switch(r)
{
    case red  : std::cout << "red\n";   break;
    case green: std::cout << "green\n"; break;
    case blue : std::cout << "blue\n";  break;
}
```

Each enumerator is associated with a value of the underlying type. When initializers are provided in the *enumerator-list*, the values of enumerators are defined by those initializers. If the first enumerator does not have an initializer, the associated value is zero. For any other enumerator whose definition does not have an initializer, the associated value is the value of the previous enumerator plus one.

```
enum Foo { a, b, c = 10, d, e = 1, f, g = f + c };
//a = 0, b = 1, c = 10, d = 11, e = 1, f = 2, g = 12
```

Values of unscoped enumeration type are implicitly-convertible to integral types. If the underlying type is not fixed, the value is convertible to the first type from the following list able to hold their entire value range: `int`, `unsigned int`, `long`, `unsigned long`, `long long`, or `unsigned long long`, extended integer types with higher conversion rank (in rank order, signed given preference over unsigned) (since C++11). If the underlying type is fixed, the values can be converted to their underlying type (preferred in overload resolution), which can then be promoted.

```
enum color { red, yellow, green = 20, blue };
color col = red;
int n = blue; // n == 21
```

Values of integer, floating-point, and enumeration types can be converted by `static_cast` or explicit cast, to any enumeration type. If the underlying type is not fixed and the source value is out of range, the behavior is undefined. (The source value, as converted to the enumeration's underlying type if floating-point, is in range if it would fit in the smallest bit field large enough to hold all enumerators of the target enumeration.) Otherwise, the result is the same as the result of implicit conversion to the underlying type.

Note that the value after such conversion may not necessarily equal any of the named enumerators defined for the enumeration.

```
enum access_t { read = 1, write = 2, exec = 4 }; // enumerators: 1, 2, 4 range: 0..7
access_t rwe = static_cast<access_t>(7);
assert((rwe & read) && (rwe & write) && (rwe & exec));

access_t x = static_cast<access_t>(8.0); // undefined behavior since CWG1766
access_t y = static_cast<access_t>(8);   // undefined behavior since CWG1766

enum foo { a = 0, b = UINT_MAX }; // range: [0, UINT_MAX]
foo x = foo(-1); // undefined behavior since CWG1766,
                 // even if foo's underlying type is unsigned int
```

The *name* of an unscoped enumeration may be omitted: such declaration only introduces the enumerators into the enclosing scope:

```
enum { a, b, c = 0, d = a + 2 }; // defines a = 0, b = 1, c = 0, d = 2
```

When an unscoped enumeration is a class member, its enumerators may be accessed using class member access operators **.** and **->**:

```
struct X
{
    enum direction { left = 'l', right = 'r' };
};
X x;
X* p = &x;

int a = X::direction::left; // allowed only in C++11 and later
int b = X::left;
int c = x.left;
int d = p->left;
```

In the declaration specifiers of a member declaration, the sequence

enum *enum-head-name* **:**

is always parsed as a part of enumeration declaration:

```
struct S
{
    enum E1 : int {};
    enum E1 : int {}; // error: redeclaration of enumeration,
                      // NOT parsed as a zero-length bit-field of type enum E1
};

enum E2 { e1 };

void f()
{
    false ? new enum E2 : int(); // OK: 'int' is NOT parsed as the underlying type
}
```

(since C++11)

Scoped enumerations

enum struct|class *name* **{** *enumerator* **=** *constexpr* **,** *enumerator* **=** *constexpr* **,** ... **}**	(1)
enum struct|class *name* **:** *type* **{** *enumerator* **=** *constexpr* **,** *enumerator* **=** *constexpr* **,** ... **}**	(2)
enum struct|class *name* **;**	(3)
enum struct|class *name* **:** *type* **;**	(4)

1) declares a scoped enumeration type whose underlying type is `int` (the keywords `class` and `struct` are exactly equivalent)
2) declares a scoped enumeration type whose underlying type is *type*
3) opaque enum declaration for a scoped enumeration whose underlying type is `int`
4) opaque enum declaration for a scoped enumeration whose underlying type is *type*

Each *enumerator* becomes a named constant of the enumeration's type (that is, *name*), which is contained within the scope of the enumeration, and can be accessed using scope resolution operator. There are no implicit conversions from the values of a scoped enumerator to integral types, although `static_cast` may be used to obtain the numeric value of the enumerator. (since C++11)

```
enum class Color { red, green = 20, blue };
Color r = Color::blue;

switch(r)
{
    case Color::red  : std::cout << "red\n";   break;
    case Color::green: std::cout << "green\n"; break;
    case Color::blue : std::cout << "blue\n";  break;
}

// int n = r; // error: no implicit conversion from scoped enum to int
int n = static_cast<int>(r); // OK, n = 21
```

An enumeration can be initialized from an integer without a cast, using list initialization, if all of the following are true:

- the initialization is direct-list-initialization
- the initializer list has only a single element
- the enumeration is either scoped or unscoped with underlying type fixed
- the conversion is non-narrowing

This makes it possible to introduce new integer types (e.g. `SafeInt`) that enjoy the same existing calling conventions as their underlying integer types, even on ABIs that penalize passing/returning structures by value.

```
enum byte : unsigned char {}; // byte is a new integer type; see also std::byte (C++17)
```

```
byte b{42};       // OK as of C++17 (direct-list-initialization)
byte c = {42};     // error
byte d = byte{42}; // OK as of C++17; same value as b
byte e{-1};       // error

struct A { byte b; };
A a1 = {{42}};     // error (copy-list-initialization of a constructor parameter)
A a2 = {byte{42}}; // OK as of C++17

void f(byte);
f({42}); // error (copy-list-initialization of a function parameter)

enum class Handle : std::uint32_t { Invalid = 0 };
Handle h{42}; // OK as of C++17
```

(since C++17)

Using-enum-declaration

using enum *nested-name-specifier*(optional) *name* **;** (since C++20)

where *nested-name-specifier*(optional) *name* must not name a dependent type and must name an enumeration type.

A using-enum-declaration introduces the enumerator names of the named enumeration as if by a using-declaration for each enumerator. When in class scope, a using-enum-declaration adds the enumerators of the named enumeration as members to the scope, making them accessible for member lookup.

```
enum class fruit { orange, apple };

struct S
{
    using enum fruit; // OK: introduces orange and apple into S
};

void f()
{
    S s;
    s.orange;  // OK: names fruit::orange
    S::orange; // OK: names fruit::orange
}
```

(since C++20)

Two using-enum-declarations that introduce two enumerators of the same name conflict.

```
enum class fruit { orange, apple };
enum class color { red, orange };

void f()
{
    using enum fruit;    // OK
    // using enum color; // error: color::orange and fruit::orange conflict
}
```

Notes

Although the conversion from an out-of-range to an enumeration without fixed underlying type is made undefined behavior by the resolution of CWG issue 1766 (https://cplusplus.github.io/CWG/issues/1766.html) , currently no compiler performs the required diagnostic for it in constant evaluation.

Feature-test macro	Value	Std	Comment
__cpp_enumerator_attributes	201411L	(C++17)	Attributes for enumerators
__cpp_using_enum	201907L	(C++20)	using enum

Example

Run this code

```
#include <iostream>
#include <cstdint>

// enum that takes 16 bits
enum smallenum: std::int16_t
{
    a,
    b,
    c
};

// color may be red (value 0), yellow (value 1), green (value 20), or blue (value 21)
enum color
{
    red,
    yellow,
    green = 20,
    blue
};

// altitude may be altitude::high or altitude::low
enum class altitude: char
{
    high = 'h',
    low = 'l', // trailing comma only allowed after CWG518
};

// the constant d is 0, the constant e is 1, the constant f is 3
enum
{
    d,
```

```
    e,
    f = e + 2
};

// enumeration types (both scoped and unscoped) can have overloaded operators
std::ostream& operator<<(std::ostream& os, color c)
{
    switch(c)
    {
        case red   : os << "red";    break;
        case yellow: os << "yellow"; break;
        case green : os << "green";  break;
        case blue  : os << "blue";   break;
        default    : os.setstate(std::ios_base::failbit);
    }
    return os;
}

std::ostream& operator<<(std::ostream& os, altitude al)
{
    return os << static_cast<char>(al);
}

// The scoped enum (C++11) can be partially emulated in earlier C++ revisions:

enum struct E11 { x, y }; // since C++11

struct E98 { enum { x, y }; }; // OK in pre-C++11

namespace N98 { enum { x, y }; } // OK in pre-C++11

struct S98 { static const int x = 0, y = 1; }; // OK in pre-C++11

void emu()
{
    std::cout << (static_cast<int>(E11::y) + E98::y + N98::y + S98::y) << '\n'; // 4
}

namespace cxx20
{
    enum class long_long_long_name { x, y };

    void using_enum_demo()
    {
        std::cout << "C++20 `using enum`: __cpp_using_enum == ";
        switch (auto rnd = []{return long_long_long_name::x;}; rnd())
        {
#if defined(__cpp_using_enum)
            using enum long_long_long_name;
            case x: std::cout << __cpp_using_enum << "; x\n"; break;
            case y: std::cout << __cpp_using_enum << "; y\n"; break;
#else
            case long_long_long_name::x: std::cout << "?; x\n"; break;
            case long_long_long_name::y: std::cout << "?; y\n"; break;
#endif
        }
    }
}

int main()
{
    color col = red;
    altitude a;
    a = altitude::low;

    std::cout << "col = " << col << '\n'
              << "a = "   << a   << '\n'
              << "f = "   << f   << '\n';

    cxx20::using_enum_demo();
}
```

Possible output:

```
col = red
a = l
f = 3
C++20 `using enum`: __cpp_using_enum == 201907; x
```

Defect reports

The following behavior-changing defect reports were applied retroactively to previously published C++ standards.

DR	Applied to	Behavior as published	Correct behavior
CWG 377 (https://cplusplus.github.io/CWG/issues/377.html)	C++98	the behavior was unspecified when no integral type can represent all the enumerator values	the enumeration is ill-formed in this case
CWG 518 (https://cplusplus.github.io/CWG/issues/518.html)	C++98	a trailing comma was not allowed after the enumerator list	allowed
CWG 1514 (https://cplusplus.github.io/CWG/issues/1514.html)	C++11	an redefinition of enumeration with fixed underlying type could be parsed as a bit-field in a class member declaration	always parsed as a redefinition
CWG 1638 (https://cplusplus.github.io/CWG/issues/1638.html)	C++11	grammar of opaque enumeration declaration prohibited use for template specializations	nested-name-specifier permitted
CWG 1766 (https://cplusplus.github.io/CWG/issues/1766.html)	C++98	casting an out-of-range value to an enumeration without fixed underlying type had an unspecified result	the behavior is undefined
CWG 1966 (https://cplusplus.github.io/CWG/issues/1966.html)	C++11	the resolution of CWG issue 1514 (https://cplusplus.github.io/CWG/issues/1514.html) made the `:` of a conditional expression part of *enum-base*	only apply the resolution to member declaration specifiers
CWG 2156 (https://cplusplus.github.io/CWG/issues/2156.html)	C++11	enum definitions could define enumeration types by using-declarations	prohibited
CWG 2157 (https://cplusplus.github.io/CWG/issues/2157.html)	C++11	the resolution of CWG issue 1966 (https://cplusplus.github.io/CWG/issues/1966.html) did not cover qualified enumeration names	covered

See also

is_enum (C++11)	checks if a type is an enumeration type (class template)

is_scoped_enum (C++23)	checks if a type is a scoped enumeration type (class template)
underlying_type (C++11)	obtains the underlying integer type for a given enumeration type (class template)
to_underlying (C++23)	converts an enumeration to its underlying type (function template)

C documentation for **Enumerations**