enumeration declaration

An *enumeration* is a distinct type whose value is restricted to one of 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 defined by *enum-specifier*, which appears in *decl-specifier-seq* of the declaration syntax. The *enum-specifier* has the following syntax:

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

1) Definition of an enumeration type.

2) *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*** - one of **enum**, **enum class**(since C++11), or **enum struct**(since C++11)
- ***attr***(C++11) - optional sequence of any number of attributes
- ***identifier*** - the name of the enumeration that's being declared. If present, and if this declaration is a re-declaration, it may be preceded by *nested-name-specifier*(since C++11): sequence of names and scope-resolution operators **::**, ending with scope-resolution operator. The name can be omitted only in unscoped enumeration declarations
- ***enum-base***(C++11) - colon (**:**), followed by a *type-specifier-seq* that names an integral type (if it is cv-qualified, qualifications are ignored)
- ***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 enumeration

enum *name* **{** *enumerator* **=** *constexpr* **,** *enumerator* **=** *constexpr* **,** ... **}**	(1)	
enum *name* **:** *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 either `int` or, if not all enumerator values can be represented as `int`, an implementation-defined larger integral type that can represent all enumerator values. If the enumerator-list is empty, the underlying type is as if the enumeration had a single enumerator with value 0).

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

3) Opaque enum declaration for an unscoped enumeration must specify 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`. If the underlying type is fixed, the values can be converted to their promoted underlying type.

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

Values of integer, floating-point, and other enumeration types can be converted, such as by `static_cast`, to any enumeration type. The result is unspecified (until C++17) undefined behavior (since C++17) if the value, converted to the enumeration's underlying type, is out of this enumeration's *range*. If the underlying type is fixed, the *range* is the range of the underlying type. If the underlying type is not fixed, the *range* is all values possible for the smallest bit field large enough to hold all enumerators of the target enumeration. Note that the value after such conversion may not necessarily equal any of the named enumerators defined for the type.

```
enum access_t { READ = 1, WRITE = 2, EXEC = 4 }; // enumerators: 1, 2, 4 range: 0..7
access_t rw = static_cast<access_t>(3);
assert(rw & READ && rw & WRITE);
```

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;
```

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 an scoped enumeration whose underlying type is `int`
4) opaque enum declaration for an scoped enumeration whose underlying type is *type* (since C++11)

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.

```
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 scoped enum to int conversion
int n = static_cast<int>(r); // OK, n = 21
```

Example

Run this code

```
#include <iostream>

// enum that takes 16 bits
enum smallenum: 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', // C++11 allows the extra comma
};

// 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);
}

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

```
    a = altitude::low;

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

Output:

```
col = red
a = l
f = 3
```

See also

C documentation for **Enumerations**