

# Welcome back to C++ - Modern C++

---

Since its creation, C++ has become one of the most widely used programming languages in the world. Well-written C++ programs are fast and efficient. The language is more flexible than other languages: It can work at the highest levels of abstraction, and down at the level of the silicon. C++ supplies highly optimized standard libraries. It enables access to low-level hardware features, to maximize speed and minimize memory requirements. Using C++, you can create a wide range of apps. Games, device drivers, and high-performance scientific software. Embedded programs. Windows client apps. Even libraries and compilers for other programming languages get written in C++.

One of the original requirements for C++ was backward compatibility with the C language. As a result, C++ has always permitted C-style programming, with raw pointers, arrays, null-terminated character strings, and other features. They may enable great performance, but can also spawn bugs and complexity. The evolution of C++ has emphasized features that greatly reduce the need to use C-style idioms. The old C-programming facilities are there when you need them, but with modern C++ code you should need them less and less. Modern C++ code is simpler, safer, more elegant, and still as fast as ever.

The following sections provide an overview of the main features of modern C++. Unless noted otherwise, the features listed here are available in C++11 and later. In the Microsoft C++ compiler, you can set the `/std` compiler option to specify which version of the standard to use for your project.

## Resources and smart pointers

One of the major classes of bugs in C-style programming is the *memory leak*. Leaks are often caused by a failure to call **delete** for memory that was allocated with **new**. Modern C++ emphasizes the principle of *resource acquisition is initialization* (RAII). The idea is simple. Resources (heap memory, file handles, sockets, and so on) should be *owned* by an object. That object creates, or receives, the newly allocated resource in its constructor, and deletes it in its destructor. The principle of RAII guarantees that all resources get properly returned to the operating system when the owning object goes out of scope.

To support easy adoption of RAII principles, the C++ Standard Library provides three smart pointer types: `std::unique_ptr`, `std::shared_ptr`, and `std::weak_ptr`. A smart pointer handles the allocation and deletion of the memory it owns. The following example shows a class with an array member that is allocated on the heap in the call to `make_unique()`. The calls to **new** and **delete** are encapsulated by the `unique_ptr` class. When a `widget` object goes out of scope, the `unique_ptr` destructor will be invoked and it will release the memory that was allocated for the array.

```
#include <memory>
class widget
{
private:
    std::unique_ptr<int> data;
public:
    widget(const int size) { data = std::make_unique<int>(size); }
    void do_something() {}
};
```

```

void functionUsingWidget() {
    widget w(1000000);    // lifetime automatically tied to enclosing scope
                        // constructs w, including the w.data gadget member
    // ...
    w.do_something();
    // ...
} // automatic destruction and deallocation for w and w.data

```

Whenever possible, use a smart pointer when allocating heap memory. If you must use the new and delete operators explicitly, follow the principle of RAII. For more information, see [Object lifetime and resource management \(RAII\)](#).

## std::string and std::string\_view

C-style strings are another major source of bugs. By using [std::string](#) and [std::wstring](#), you can eliminate virtually all the errors associated with C-style strings. You also gain the benefit of member functions for searching, appending, prepending, and so on. Both are highly optimized for speed. When passing a string to a function that requires only read-only access, in C++17 you can use [std::string\\_view](#) for even greater performance benefit.

## std::vector and other Standard Library containers

The Standard Library containers all follow the principle of RAII. They provide iterators for safe traversal of elements. And, they're highly optimized for performance and have been thoroughly tested for correctness. By using these containers, you eliminate the potential for bugs or inefficiencies that might be introduced in custom data structures. Instead of raw arrays, use [vector](#) as a sequential container in C++.

```

vector<string> apples;
apples.push_back("Granny Smith");

```

Use [map](#) (not [unordered\\_map](#)) as the default associative container. Use [set](#), [multimap](#), and [multiset](#) for degenerate and multi cases.

```

map<string, string> apple_color;
// ...
apple_color["Granny Smith"] = "Green";

```

When performance optimization is needed, consider using:

- The [array](#) type when embedding is important, for example, as a class member.
- Unordered associative containers such as [unordered\\_map](#). These have lower per-element overhead and constant-time lookup, but they can be harder to use correctly and efficiently.
- Sorted [vector](#). For more information, see [Algorithms](#).

Don't use C-style arrays. For older APIs that need direct access to the data, use accessor methods such as `f(vec.data(), vec.size());` instead. For more information about containers, see [C++ Standard Library Containers](#).

## Standard Library algorithms

Before you assume that you need to write a custom algorithm for your program, first review the C++ Standard Library [algorithms](#). The Standard Library contains an ever-growing assortment of algorithms for many common operations such as searching, sorting, filtering, and randomizing. The math library is extensive. Starting in C++17, parallel versions of many algorithms are provided.

Here are some important examples:

- **for\_each**, the default traversal algorithm (along with range-based for loops).
- **transform**, for not-in-place modification of container elements
- **find\_if**, the default search algorithm.
- **sort**, **lower\_bound**, and the other default sorting and searching algorithms.

To write a comparator, use strict `<` and use *named lambdas* when you can.

```
auto comp = [](const widget& w1, const widget& w2)
    { return w1.weight() < w2.weight(); }

sort( v.begin(), v.end(), comp );

auto i = lower_bound( v.begin(), v.end(), comp );
```

## auto instead of explicit type names

C++11 introduced the [auto](#) keyword for use in variable, function, and template declarations. **auto** tells the compiler to deduce the type of the object so that you don't have to type it explicitly. **auto** is especially useful when the deduced type is a nested template:

```
map<int, list<string>>>::iterator i = m.begin(); // C-style
auto i = m.begin(); // modern C++
```

## Range-based for loops

C-style iteration over arrays and containers is prone to indexing errors and is also tedious to type. To eliminate these errors, and make your code more readable, use range-based for loops with both Standard Library containers and raw arrays. For more information, see [Range-based for statement](#).

```
#include <iostream>
#include <vector>
```

```

int main()
{
    std::vector<int> v {1,2,3};

    // C-style
    for(int i = 0; i < v.size(); ++i)
    {
        std::cout << v[i];
    }

    // Modern C++:
    for(auto& num : v)
    {
        std::cout << num;
    }
}

```

## constexpr expressions instead of macros

Macros in C and C++ are tokens that are processed by the preprocessor before compilation. Each instance of a macro token is replaced with its defined value or expression before the file is compiled. Macros are commonly used in C-style programming to define compile-time constant values. However, macros are error-prone and difficult to debug. In modern C++, you should prefer [constexpr](#) variables for compile-time constants:

```

#define SIZE 10 / C-style
constexpr int size = 10; // modern C++

```

## Uniform initialization

In modern C++, you can use brace initialization for any type. This form of initialization is especially convenient when initializing arrays, vectors, or other containers. In the following example, **v2** is initialized with three instances of **S**. **v3** is initialized with three instances of **S** that are themselves initialized using braces. The compiler infers the type of each element based on the declared type of **v3**.

```

#include <vector>

struct S
{
    std::string name;
    float num;
    S(std::string s, float f) : name(s), num(f) {}
};

int main()
{
    // C-style initialization

```

```

std::vector<S> v;
S s1("Norah", 2.7);
S s2("Frank", 3.5);
S s3("Jeri", 85.9);

v.push_back(s1);
v.push_back(s2);
v.push_back(s3);

// Modern C++:
std::vector<S> v2 {s1, s2, s3};

// or...
std::vector<S> v3{ {"Norah", 2.7}, {"Frank", 3.5}, {"Jeri", 85.9} };

}

```

For more information, see [Brace initialization](#).

## Move semantics

Modern C++ provides *move semantics*, which make it possible to eliminate unnecessary memory copies. In earlier versions of the language, copies were unavoidable in certain situations. A *move* operation transfers ownership of a resource from one object to the next without making a copy. When implementing a class that owns a resource (such as heap memory, file handles, and so on), you can define a *move constructor* and *move assignment operator* for it. The compiler will choose these special members during overload resolution in situations where a copy isn't needed. The Standard Library container types invoke the move constructor on objects if one is defined. For more information, see [Move Constructors and Move Assignment Operators \(C++\)](#).

## Lambda expressions

In C-style programming, a function can be passed to another function by using a *function pointer*. Function pointers are inconvenient to maintain and understand. The function they refer to may be defined elsewhere in the source code, far away from the point at which it's invoked. Also, they're not type-safe. Modern C++ provides *function objects*, classes that override the `()` operator, which enables them to be called like a function. The most convenient way to create function objects is with inline [lambda expressions](#). The following example shows how to use a lambda expression to pass a function object, that the `for_each` function will invoke on each element in the vector:

```

std::vector<int> v {1,2,3,4,5};
int x = 2;
int y = 4;
auto result = find_if(begin(v), end(v), [](int i) { return i > x && i < y;
});

```

The lambda expression `[](int i) { return i > x && i < y; }` can be read as "function that takes a single argument of type `int` and returns a boolean that indicates whether the argument is greater than `x` and

less than `y`." Notice that the variables `x` and `y` from the surrounding context can be used in the lambda. The `[=]` specifies that those variables are *captured* by value; in other words, the lambda expression has its own copies of those values.

## Exceptions

Modern C++ emphasizes exceptions rather than error codes as the best way to report and handle error conditions. For more information, see [Modern C++ best practices for exceptions and error handling](#).

## `std::atomic`

Use the C++ Standard Library `std::atomic` struct and related types for inter-thread communication mechanisms.

## `std::variant` (C++17)

Unions are commonly used in C-style programming to conserve memory by enabling members of different types to occupy the same memory location. However, unions aren't type-safe and are prone to programming errors. C++17 introduces the `std::variant` class as a more robust and safe alternative to unions. The `std::visit` function can be used to access the members of a `variant` type in a type-safe manner.

## See also

[C++ Language Reference](#)

[Lambda Expressions](#)

[C++ Standard Library](#)

[Microsoft C++ language conformance table](#)

# Lexical conventions

---

This section introduces the fundamental elements of a C++ program. You use these elements, called "lexical elements" or "tokens" to construct statements, definitions, declarations, and so on, which are used to construct complete programs. The following lexical elements are discussed in this section:

- [Tokens and character sets](#)
- [Comments](#)
- [Identifiers](#)
- [Keywords](#)
- [Punctuators](#)
- [Numeric, boolean, and pointer literals](#)
- [String and character literals](#)
- [User-defined literals](#)

For more information about how C++ source files are parsed, see [Phases of translation](#).

## See also

[C++ Language Reference](#)

[Translation units and linkage](#)

# Tokens and character sets

---

The text of a C++ program consists of tokens and *white space*. A token is the smallest element of a C++ program that is meaningful to the compiler. The C++ parser recognizes these kinds of tokens:

- [Keywords](#)
- [Identifiers](#)
- [Numeric, Boolean and Pointer Literals](#)
- [String and Character Literals](#)
- [User-Defined Literals](#)
- [Operators](#)
- [Punctuators](#)

Tokens are usually separated by *white space*, which can be one or more:

- Blanks
- Horizontal or vertical tabs
- New lines
- Form feeds
- Comments


## Basic source character set

The C++ standard specifies a *basic source character set* that may be used in source files. To represent characters outside of this set, additional characters can be specified by using a *universal character name*. The MSVC implementation allows additional characters. The *basic source character set* consists of 96 characters that may be used in source files. This set includes the space character, horizontal tab, vertical tab, form feed and new-line control characters, and this set of graphical characters:

a b c d e f g h i j k l m n o p q r s t u v w x y z

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

0 1 2 3 4 5 6 7 8 9

 # ( ) < > % ; : . ? \* + - / ^ & | ~ ! = , \ " ' `

### Microsoft Specific

MSVC includes the `$` character as a member of the basic source character set. MSVC also allows an additional set of characters to be used in source files, based on the file encoding. By default, Visual Studio stores source files by using the default codepage. When source files are saved by using a locale-specific codepage or a Unicode codepage, MSVC allows you to use any of the characters of that code page in your source code, except for the control codes not explicitly allowed in the basic source character set. For example, you can put Japanese characters in comments, identifiers, or string literals if you save the file using a Japanese codepage. MSVC does not allow character sequences that cannot be translated into valid multibyte characters or Unicode code points. Depending on compiler options, not all allowed characters may appear in identifiers. For more information, see [Identifiers](#).



## END Microsoft Specific

### Universal character names

Because C++ programs can use many more characters than the ones specified in the basic source character set, you can specify these characters in a portable way by using *universal character names*. A universal character name consists of a sequence of characters that represent a Unicode code point. These take two forms. Use `\UNNNNNNNNN` to represent a Unicode code point of the form U+NNNNNNNN, where NNNNNNNN is the eight-digit hexadecimal code point number. Use four-digit `\uNNNN` to represent a Unicode code point of the form U+0000NNNN.

Universal character names can be used in identifiers and in string and character literals. A universal character name cannot be used to represent a surrogate code point in the range 0xD800-0xDFFF. Instead, use the desired code point; the compiler automatically generates any required surrogates. Additional restrictions apply to the universal character names that can be used in identifiers. For more information, see [Identifiers](#) and [String and Character Literals](#).

## Microsoft Specific

The Microsoft C++ compiler treats a character in universal character name form and literal form interchangeably. For example, you can declare an identifier using universal character name form, and use it in literal form:

```
auto \u30AD = 42; // \u30AD is 'キ'
if (キ == 42) return true; // \u30AD and キ are the same to the compiler
```

The format of extended characters on the Windows clipboard is specific to application locale settings. Cutting and pasting these characters into your code from another application may introduce unexpected character encodings. This can result in parsing errors with no visible cause in your code. We recommend that you set your source file encoding to a Unicode codepage before pasting extended characters. We also recommend that you use an IME or the Character Map app to generate extended characters.

## END Microsoft Specific

### Execution character sets

The *execution character sets* represent the characters and strings that can appear in a compiled program. These character sets consist of all the characters permitted in a source file, and also the control characters that represent alert, backspace, carriage return, and the null character. The execution character set has a locale-specific representation.

# Comments (C++)

---

A comment is text that the compiler ignores but that is useful for programmers. Comments are normally used to annotate code for future reference. The compiler treats them as white space. You can use comments in testing to make certain lines of code inactive; however, `#if/#endif` preprocessor directives work better for this because you can surround code that contains comments but you cannot nest comments.

A C++ comment is written in one of the following ways:

- The `/*` (slash, asterisk) characters, followed by any sequence of characters (including new lines), followed by the `*/` characters. This syntax is the same as ANSI C.
- The `//` (two slashes) characters, followed by any sequence of characters. A new line not immediately preceded by a backslash terminates this form of comment. Therefore, it is commonly called a "single-line comment."

The comment characters (`/*`, `*/`, and `//`) have no special meaning within a character constant, string literal, or comment. Comments using the first syntax, therefore, cannot be nested.

## See also

[Lexical Conventions](#)

# Identifiers (C++)

---

An identifier is a sequence of characters used to denote one of the following:

- Object or variable name
- Class, structure, or union name
- Enumerated type name
- Member of a class, structure, union, or enumeration
- Function or class-member function
- typedef name
- Label name
- Macro name
- Macro parameter

The following characters are allowed as any character of an identifier:

```
_ a b c d e f g h i j k l m  
n o p q r s t u v w x y z  
A B C D E F G H I J K L M  
N O P Q R S T U V W X Y Z
```

Certain ranges of universal character names are also allowed in an identifier. A universal character name in an identifier cannot designate a control character or a character in the basic source character set. For more information, see [Character Sets](#). These Unicode code point number ranges are allowed as universal character names for any character in an identifier:

- 00A8, 00AA, 00AD, 00AF, 00B2-00B5, 00B7-00BA, 00BC-00BE, 00C0-00D6, 00D8-00F6, 00F8-00FF, 0100-02FF, 0370-167F, 1681-180D, 180F-1DBF, 1E00-1FFF, 200B-200D, 202A-202E, 203F-2040, 2054, 2060-206F, 2070-20CF, 2100-218F, 2460-24FF, 2776-2793, 2C00-2DFF, 2E80-2FFF, 3004-3007, 3021-302F, 3031-303F, 3040-D7FF, F900-FD3D, FD40-FDCF, FDF0-FE1F, FE30-FE44, FE47-FFFF, 10000-1FFFF, 20000-2FFFF, 30000-3FFFF, 40000-4FFFF, 50000-5FFFF, 60000-6FFFF, 70000-7FFFF, 80000-8FFFF, 90000-9FFFF, A0000-AFFFF, B0000-BFFFF, C0000-CFFFF, D0000-DFFFF, E0000-EFFFF

The following characters are allowed as any character in an identifier except the first:

```
0 1 2 3 4 5 6 7 8 9
```

These Unicode code point number ranges are also allowed as universal character names for any character in an identifier except the first:

- 0300-036F, 1DC0-1DFF, 20D0-20FF, FE20-FE2F

## Microsoft Specific

Only the first 2048 characters of Microsoft C++ identifiers are significant. Names for user-defined types are "decorated" by the compiler to preserve type information. The resultant name, including the type information, cannot be longer than 2048 characters. (See [Decorated Names](#) for more information.) Factors that can influence the length of a decorated identifier are:

- Whether the identifier denotes an object of user-defined type or a type derived from a user-defined type.
- Whether the identifier denotes a function or a type derived from a function.
- The number of arguments to a function.

The dollar sign `$` is a valid identifier character in the Microsoft C++ compiler (MSVC). MSVC also allows you to use the actual characters represented by the allowed ranges of universal character names in identifiers. To use these characters, you must save the file by using a file encoding codepage that includes them. This example shows how both extended characters and universal character names can be used interchangeably in your code.

```
// extended_identifier.cpp
// In Visual Studio, use File, Advanced Save Options to set
// the file encoding to Unicode codepage 1200
struct テスト          // Japanese 'test'
{
    void トスト() {}    // Japanese 'toast'
};

int main() {
    テスト \u30D1\u30F3; // Japanese パン 'bread' in UCN form
    パン.トスト();       // compiler recognizes UCN or literal form
}
```

The range of characters allowed in an identifier is less restrictive when compiling C++/CLI code. Identifiers in code compiled by using `/clr` should follow [Standard ECMA-335: Common Language Infrastructure \(CLI\)](#).

## END Microsoft Specific

The first character of an identifier must be an alphabetic character, either uppercase or lowercase, or an underscore (`_`). Because C++ identifiers are case sensitive, `fileName` is different from `FileName`.

Identifiers cannot be exactly the same spelling and case as keywords. Identifiers that contain keywords are legal. For example, `Pint` is a legal identifier, even though it contains `int`, which is a keyword.

Use of two sequential underscore characters (`__`) in an identifier, or a single leading underscore followed by a capital letter, is reserved for C++ implementations in all scopes. You should avoid using one leading underscore followed by a lowercase letter for names with file scope because of possible conflicts with current or future reserved identifiers.

## See also

[Lexical Conventions](#)

# Keywords (C++)

---

Keywords are predefined reserved identifiers that have special meanings. They cannot be used as identifiers in your program. The following keywords are reserved for Microsoft C++. Names with leading underscores, and names followed by (C++/CLI) are Microsoft extensions.

<a href="#">__abstract</a> <sup>2</sup>	<a href="#">__alignof Operator</a> <sup>4</sup>	<a href="#">__asm</a> <sup>4</sup>	<a href="#">__assume</a> <sup>4</sup>
<a href="#">__based</a> <sup>4</sup>	<a href="#">__box</a> <sup>2</sup>	<a href="#">__cdecl</a> <sup>4</sup>	<a href="#">__declspec</a> <sup>4</sup>
<a href="#">__delegate</a> <sup>2</sup>	<a href="#">__event</a>	<a href="#">__except</a> <sup>4</sup>	<a href="#">__fastcall</a> <sup>4</sup>
<a href="#">__finally</a> <sup>4</sup>	<a href="#">__forceinline</a> <sup>4</sup>	<a href="#">__gc</a> <sup>2</sup>	<a href="#">__hook</a> <sup>3</sup>
<a href="#">__identifier</a>	<a href="#">__if_exists</a>	<a href="#">__if_not_exists</a>	<a href="#">__inline</a> <sup>4</sup>
<a href="#">__int16</a> <sup>4</sup>	<a href="#">__int32</a> <sup>4</sup>	<a href="#">__int64</a> <sup>4</sup>	<a href="#">__int8</a> <sup>4</sup>
<a href="#">__interface</a>	<a href="#">__leave</a> <sup>4</sup>	<a href="#">__m128</a>	<a href="#">__m128d</a>
<a href="#">__m128i</a>	<a href="#">__m64</a>	<a href="#">__multiple_inheritance</a> <sup>4</sup>	<a href="#">__nogc</a> <sup>2</sup>
<a href="#">__noop</a>	<a href="#">__pin</a> <sup>2</sup>	<a href="#">__property</a> <sup>2</sup>	<a href="#">__ptr32</a> <sup>4</sup>
<a href="#">__ptr64</a> <sup>4</sup>	<a href="#">__raise</a>	<a href="#">__restrict</a> <sup>4</sup>	<a href="#">__sealed</a> <sup>2</sup>
<a href="#">__single_inheritance</a> <sup>4</sup>	<a href="#">__sptr</a> <sup>4</sup>	<a href="#">__stdcall</a> <sup>4</sup>	<a href="#">__super</a>
<a href="#">__thiscall</a>	<a href="#">__try_cast</a> <sup>2</sup>	<a href="#">__unaligned</a> <sup>4</sup>	<a href="#">__unhook</a> <sup>3</sup>
<a href="#">__uptr</a> <sup>4</sup>	<a href="#">__uuidof</a> <sup>4</sup>	<a href="#">__value</a> <sup>2</sup>	<a href="#">__vectorcall</a> <sup>4</sup>
<a href="#">__virtual_inheritance</a> <sup>4</sup>	<a href="#">__w64</a> <sup>4</sup>	<a href="#">__wchar_t</a>	<a href="#">abstract(C++/CLI)</a>
<a href="#">alignas</a>	<a href="#">array(C++/CLI)</a>	<a href="#">auto</a>	<a href="#">bool</a>
<a href="#">break</a>	<a href="#">case</a>	<a href="#">catch</a>	<a href="#">char</a>
<a href="#">char16_t</a>	<a href="#">char32_t</a>	<a href="#">class</a>	<a href="#">const</a>
<a href="#">const_cast</a>	<a href="#">constexpr</a>	<a href="#">continue</a>	<a href="#">decltype</a>
<a href="#">default</a>	<a href="#">delegate(C++/CLI)</a>	<a href="#">delete</a>	<a href="#">deprecated</a> <sup>1</sup>
<a href="#">dllexport</a> <sup>1</sup>	<a href="#">dllimport</a> <sup>1</sup>	<a href="#">do</a>	<a href="#">double</a>
<a href="#">dynamic_cast</a>	<a href="#">else</a>	<a href="#">enum</a>	<a href="#">enum class</a>
<a href="#">enum struct</a>	<a href="#">event(C++/CLI)</a>	<a href="#">explicit</a>	<a href="#">extern</a>
<a href="#">false</a>	<a href="#">finally</a>	<a href="#">float</a>	<a href="#">for</a>

<a href="#">for each in</a>	<a href="#">friend</a>	<a href="#">friend_as</a>	<a href="#">gcnew(C++/CLI)</a>
<a href="#">generic(C++/CLI)</a>	<a href="#">goto</a>	<a href="#">if</a>	<a href="#">initonly</a>
<a href="#">inline</a>	<a href="#">int</a>	<a href="#">interface class(C++/CLI)</a>	<a href="#">interface struct(C++/CLI)</a>
<a href="#">interior_ptr(C++/CLI)</a>	<a href="#">literal(C++/CLI)</a>	<a href="#">long</a>	<a href="#">mutable</a>
<a href="#">naked<sup>1</sup></a>	<a href="#">namespace</a>	<a href="#">new(C++/CLI)</a>	<a href="#">new</a>
<a href="#">noexcept</a>	<a href="#">noinline<sup>1</sup></a>	<a href="#">noreturn<sup>1</sup></a>	<a href="#">nothrow<sup>1</sup></a>
<a href="#">novtable<sup>1</sup></a>	<a href="#">nullptr</a>	<a href="#">operator</a>	<a href="#">private</a>
<a href="#">property(C++/CLI)</a>	<a href="#">property<sup>1</sup></a>	<a href="#">protected</a>	<a href="#">public</a>
<a href="#">ref class</a>	<a href="#">ref struct</a>	<a href="#">register</a>	<a href="#">reinterpret_cast</a>
<a href="#">return</a>	<a href="#">safecast</a>	<a href="#">sealed(C++/CLI)</a>	<a href="#">selectany<sup>1</sup></a>
<a href="#">short</a>	<a href="#">signed</a>	<a href="#">sizeof</a>	<a href="#">static</a>
<a href="#">static_assert</a>	<a href="#">static_cast</a>	<a href="#">struct</a>	<a href="#">switch</a>
<a href="#">template</a>	<a href="#">this</a>	<a href="#">thread<sup>1</sup></a>	<a href="#">throw</a>
<a href="#">true</a>	<a href="#">try</a>	<a href="#">typedef</a>	<a href="#">typeid</a>
<a href="#">typeid</a>	<a href="#">typename</a>	<a href="#">union</a>	<a href="#">unsigned</a>
<a href="#">using declaration</a>	<a href="#">using directive</a>	<a href="#">uuid<sup>1</sup></a>	<a href="#">value class(C++/CLI)</a>
<a href="#">value struct(C++/CLI)</a>	<a href="#">virtual</a>	<a href="#">void</a>	<a href="#">volatile</a>
<a href="#">while</a>			

<sup>1</sup> Extended attributes for the **\_\_declspec** keyword.

<sup>2</sup> Applicable to Managed Extensions for C++ only. This syntax is now deprecated. See [Component Extensions for Runtime Platforms](#) for more information.

<sup>3</sup> Intrinsic function used in event handling.

<sup>4</sup> For backward compatibility with previous versions, these keywords are available both with two leading underscores and a single leading underscore when Microsoft extensions are enabled (the default).

## Microsoft Specific

In Microsoft C++, identifiers with two leading underscores are reserved for compiler implementations. Therefore, the Microsoft convention is to precede Microsoft-specific keywords with double underscores. These words cannot be used as identifier names.

Microsoft extensions are enabled by default. To ensure that your programs are fully portable, you can disable Microsoft extensions by specifying the [/Za \(Disable language extensions\)](#) option during compilation. When

you do this, some Microsoft-specific keywords are disabled.

When Microsoft extensions are enabled, you can use the Microsoft-specific keywords in your programs. For ANSI compliance, these keywords are prefaced by a double underscore. For backward compatibility, single-underscore versions of many of the double-underscored keywords are supported. In addition, **\_\_cdecl** is available with no leading underscore.

The **\_\_asm** keyword replaces C++ **asm** syntax. **asm** is reserved for compatibility with other C++ implementations, but not implemented. Use **\_\_asm**.

The **\_\_based** keyword has limited uses for 32-bit and 64-bit target compilations.

## **END Microsoft Specific**

## See also

[Lexical Conventions](#)

[C++ Built-in Operators, Precedence and Associativity](#)



# Punctuators (C++)

---

Punctuators in C++ have syntactic and semantic meaning to the compiler but do not, of themselves, specify an operation that yields a value. Some punctuators, either alone or in combination, can also be C++ operators or be significant to the preprocessor.

Any of the following characters are considered punctuators:

```
! % ^ & * ( ) - + = { } | ~  
[ ] \ ; ' : " < > ? , . / #
```

The punctuators `[ ]`, `( )`, and `{ }` must appear in pairs after [translation phase 4](#).

## See also

[Lexical Conventions](#)

# Numeric, boolean and pointer literals

---

A literal is a program element that directly represents a value. This article covers literals of type integer, floating-point, boolean and pointer. For information about string and character literals, see [String and Character Literals \(C++\)](#). You can also define your own literals based on any of these categories; for more information see [User-Defined Literals \(C++\)](#)

. You can use literals in many contexts, but most commonly to initialize named variables and to pass arguments to functions:

```
const int answer = 42; // integer literal
double d = sin(108.87); //floating point literal passed to sin function
bool b = true; // boolean literal
MyClass* mc = nullptr; // pointer literal
```

Sometimes it's important to tell the compiler how to interpret a literal, or what specific type to give to it. You do this by appending prefixes or suffixes to the literal. For example, the prefix `0x` tells the compiler to interpret the number that follows it as a hexadecimal value, for example `0x35`. The `ULL` suffix tells the compiler to treat the value as an **unsigned long long** type, as in `5894345ULL`. See the following sections for the complete list of prefixes and suffixes for each literal type.

## Integer literals

Integer literals begin with a digit and have no fractional parts or exponents. You can specify integer literals in decimal, octal, or hexadecimal form. They can specify signed or unsigned types and long or short types.

When no prefix or suffix is present, the compiler will give an integral literal value type **int** (32 bits), if the value will fit, otherwise it will give it type **long long** (64 bits).

To specify a decimal integral literal, begin the specification with a nonzero digit. For example:

```
int i = 157; // Decimal literal
int j = 0198; // Not a decimal number; erroneous octal literal
int k = 0365; // Leading zero specifies octal literal, not decimal
int m = 36'000'000 // digit separators make large values more readable
int
```

To specify an octal integral literal, begin the specification with `0`, followed by a sequence of digits in the range 0 through 7. The digits 8 and 9 are errors in specifying an octal literal. For example:

```
int i = 0377; // Octal literal
int j = 0397; // Error: 9 is not an octal digit
```

To specify a hexadecimal integral literal, begin the specification with `0x` or `0X` (the case of the "x" does not matter), followed by a sequence of digits in the range `0` through `9` and `a` (or `A`) through `f` (or `F`). Hexadecimal digits `a` (or `A`) through `f` (or `F`) represent values in the range 10 through 15. For example:

```
int i = 0x3fff;    // Hexadecimal literal
int j = 0X3FFF;    // Equal to i
```

To specify an unsigned type, use either the `u` or `U` suffix. To specify a long type, use either the `l` or `L` suffix. To specify a 64-bit integral type, use the `ll` or `ll` suffix. The `i64` suffix is still supported but should be avoided because it is specific to Microsoft and is not portable. For example:

```
unsigned val_1 = 328u;           // Unsigned value
long val_2 = 0x7FFFFFFL;         // Long value specified
                                // as hex literal
unsigned long val_3 = 0776745ul; // Unsigned long value
auto val_4 = 108LL;              // signed long long
auto val_4 = 0x8000000000000000ULL << 16; // unsigned long long
```

**Digit separators:** You can use the single-quote character (apostrophe) to separate place values in larger numbers to make them easier for humans to read. Separators have no effect on compilation.

```
long long i = 24'847'458'121
```

## Floating point literals

Floating-point literals specify values that must have a fractional part. These values contain decimal points (.) and can contain exponents.

Floating-point literals have a "mantissa," which specifies the value of the number, an "exponent," which specifies the magnitude of the number, and an optional suffix that specifies the literal's type. The mantissa is specified as a sequence of digits followed by a period, followed by an optional sequence of digits representing the fractional part of the number. For example:

```
18.46
38.
```

The exponent, if present, specifies the magnitude of the number as a power of 10, as shown in the following example:

```
18.46e0    // 18.46
18.46e1    // 184.6
```

The exponent may be specified using **e** or **E**, which have the same meaning, followed by an optional sign (+ or -) and a sequence of digits. If an exponent is present, the trailing decimal point is unnecessary in whole numbers such as **18E0**.

Floating-point literals default to type **double**. By using the suffixes **f** or **l** (or **F** or **L** — the suffix is not case sensitive), the literal can be specified as **float** or **long double**, respectively.

Although **long double** and **double** have the same representation, they are not the same type. For example, you can have overloaded functions like

```
void func( double );
```

and

```
void func( long double );
```

## Boolean literals

The boolean literals are **true** and **false**.

## Pointer literal (C++11)

C++ introduces the **nullptr** literal to specify a zero-initialized pointer. In portable code, **nullptr** should be used instead of integral-type zero or macros such as **NULL**.

## Binary literals (C++14)

A binary literal can be specified by the use of the **0B** or **0b** prefix, followed by a sequence of 1's and 0's:

```
auto x = 0B001101 ; // int
auto y = 0b000001 ; // int
```

## Avoid using literals as "magic constants"

You can use literals directly in expressions and statements although it's not always good programming practice:

```
if (num < 100)
    return "Success";
```

In the previous example, it might be better to use a named constant that conveys a clear meaning, for example **"MAXIMUM\_ERROR\_THRESHOLD"**. And if the return value **"Success"** is seen by end users, then it might be better to use a named string constant that can be stored in a single location in a file from where it

can be localized into other languages. Using named constants helps others as well as yourself to understand the intent of the code.

## See also

[Lexical Conventions](#)

[C++ String Literals](#)

[C++ User-Defined Literals](#)

# String and character literals (C++)

---

C++ supports various string and character types, and provides ways to express literal values of each of these types. In your source code, you express the content of your character and string literals using a character set. Universal character names and escape characters allow you to express any string using only the basic source character set. A raw string literal enables you to avoid using escape characters, and can be used to express all types of string literals. You can also create `std::string` literals without having to perform extra construction or conversion steps.

```
#include <string>
using namespace std::string_literals; // enables s-suffix for std::string literals

int main()
{
    // Character literals
    auto c0 = 'A'; // char
    auto c1 = u8'A'; // char
    auto c2 = L'A'; // wchar_t
    auto c3 = u'A'; // char16_t
    auto c4 = U'A'; // char32_t

    // Multicharacter literals
    auto m0 = 'abcd'; // int, value 0x61626364

    // String literals
    auto s0 = "hello"; // const char*
    auto s1 = u8"hello"; // const char*, encoded as UTF-8
    auto s2 = L"hello"; // const wchar_t*
    auto s3 = u"hello"; // const char16_t*, encoded as UTF-16
    auto s4 = U"hello"; // const char32_t*, encoded as UTF-32

    // Raw string literals containing unescaped \ and "
    auto R0 = R("Hello \ world"); // const char*
    auto R1 = u8R("Hello \ world"); // const char*, encoded as UTF-8
    auto R2 = LR("Hello \ world"); // const wchar_t*
    auto R3 = uR("Hello \ world"); // const char16_t*, encoded as UTF-16
    auto R4 = UR("Hello \ world"); // const char32_t*, encoded as UTF-32

    // Combining string literals with standard s-suffix
    auto S0 = "hello"s; // std::string
    auto S1 = u8"hello"s; // std::string
    auto S2 = L"hello"s; // std::wstring
    auto S3 = u"hello"s; // std::u16string
    auto S4 = U"hello"s; // std::u32string

    // Combining raw string literals with standard s-suffix
    auto S5 = R("Hello \ world")s; // std::string from a raw const char*
    auto S6 = u8R("Hello \ world")s; // std::string from a raw const char*,
    encoded as UTF-8
    auto S7 = LR("Hello \ world")s; // std::wstring from a raw const wchar_t*
```

```

    auto S8 = uR("Hello \ world")s; // std::u16string from a raw const
    char16_t*, encoded as UTF-16
    auto S9 = UR("Hello \ world")s; // std::u32string from a raw const
    char32_t*, encoded as UTF-32
}

```

String literals can have no prefix, or `u8`, `L`, `u`, and `U` prefixes to denote narrow character (single-byte or multi-byte), UTF-8, wide character (UCS-2 or UTF-16), UTF-16 and UTF-32 encodings, respectively. A raw string literal can have `R`, `u8R`, `LR`, `uR`, and `UR` prefixes for the raw version equivalents of these encodings. To create temporary or static `std::string` values, you can use string literals or raw string literals with an `s` suffix. For more information, see the [String literals](#) section below. For more information on the basic source character set, universal character names, and using characters from extended codepages in your source code, see [Character sets](#).

## Character literals

A *character literal* is composed of a constant character. It's represented by the character surrounded by single quotation marks. There are five kinds of character literals:

- Ordinary character literals of type **char**, for example `'a'`
- UTF-8 character literals of type **char** (**char8\_t** in C++20), for example `u8'a'`
- Wide-character literals of type **wchar\_t**, for example `L'a'`
- UTF-16 character literals of type **char16\_t**, for example `u'a'`
- UTF-32 character literals of type **char32\_t**, for example `U'a'`

The character used for a character literal may be any character, except for the reserved characters backslash (`\`), single quotation mark (`'`), or newline. Reserved characters can be specified by using an escape sequence. Characters may be specified by using universal character names, as long as the type is large enough to hold the character.

## Encoding

Character literals are encoded differently based their prefix.

- A character literal without a prefix is an ordinary character literal. The value of an ordinary character literal containing a single character, escape sequence, or universal character name that can be represented in the execution character set has a value equal to the numerical value of its encoding in the execution character set. An ordinary character literal that contains more than one character, escape sequence, or universal character name is a *multicharacter literal*. A multicharacter literal or an ordinary character literal that can't be represented in the execution character set has type **int**, and its value is implementation-defined. For MSVC, see the **Microsoft-specific** section below.
- A character literal that begins with the `L` prefix is a wide-character literal. The value of a wide-character literal containing a single character, escape sequence, or universal character name has a value equal to the numerical value of its encoding in the execution wide-character set unless the character literal has no representation in the execution wide-character set, in which case the value is implementation-

defined. The value of a wide-character literal containing multiple characters, escape sequences, or universal character names is implementation-defined. For MSVC, see the **Microsoft-specific** section below.

- A character literal that begins with the `u8` prefix is a UTF-8 character literal. The value of a UTF-8 character literal containing a single character, escape sequence, or universal character name has a value equal to its ISO 10646 code point value if it can be represented by a single UTF-8 code unit (corresponding to the C0 Controls and Basic Latin Unicode block). If the value can't be represented by a single UTF-8 code unit, the program is ill-formed. A UTF-8 character literal containing more than one character, escape sequence, or universal character name is ill-formed.
- A character literal that begins with the `u` prefix is a UTF-16 character literal. The value of a UTF-16 character literal containing a single character, escape sequence, or universal character name has a value equal to its ISO 10646 code point value if it can be represented by a single UTF-16 code unit (corresponding to the basic multi-lingual plane). If the value can't be represented by a single UTF-16 code unit, the program is ill-formed. A UTF-16 character literal containing more than one character, escape sequence, or universal character name is ill-formed.
- A character literal that begins with the `U` prefix is a UTF-32 character literal. The value of a UTF-32 character literal containing a single character, escape sequence, or universal character name has a value equal to its ISO 10646 code point value. A UTF-32 character literal containing more than one character, escape sequence, or universal character name is ill-formed.

## Escape sequences

There are three kinds of escape sequences: simple, octal, and hexadecimal. Escape sequences may be any of the following values:

Value	Escape sequence
newline	<code>\n</code>
backslash	<code>\\</code>
horizontal tab	<code>\t</code>
question mark	<code>?</code> or <code>\?</code>
vertical tab	<code>\v</code>
single quote	<code>\'</code>
backspace	<code>\b</code>
double quote	<code>\"</code>
carriage return	<code>\r</code>
the null character	<code>\0</code>
form feed	<code>\f</code>
octal	<code>\ooo</code>



Value	Escape sequence
alert (bell)	\a
hexadecimal	\xhhh

An octal escape sequence is a backslash followed by a sequence of one to three octal digits. An octal escape sequence terminates at the first character that's not an octal digit, if encountered sooner than the third digit. The highest possible octal value is `\377`.

A hexadecimal escape sequence is a backslash followed by the character `x`, followed by a sequence of one or more hexadecimal digits. Leading zeroes are ignored. In an ordinary or `u8`-prefixed character literal, the highest hexadecimal value is `0xFF`. In an `L`-prefixed or `u`-prefixed wide character literal, the highest hexadecimal value is `0xFFFF`. In a `U`-prefixed wide character literal, the highest hexadecimal value is `0xFFFFFFFF`.

This sample code shows some examples of escaped characters using ordinary character literals. The same escape sequence syntax is valid for the other character literal types.

```
#include <iostream>
using namespace std;

int main() {
    char newline = '\n';
    char tab = '\t';
    char backspace = '\b';
    char backslash = '\\';
    char nullChar = '\0';

    cout << "Newline character: " << newline << "ending" << endl;
    cout << "Tab character: " << tab << "ending" << endl;
    cout << "Backspace character: " << backspace << "ending" << endl;
    cout << "Backslash character: " << backslash << "ending" << endl;
    cout << "Null character: " << nullChar << "ending" << endl;
}
/* Output:
Newline character:
ending
Tab character:  ending
Backspace character:ending
Backslash character: \ending
Null character:  ending
*/
```

The backslash character (`\`) is a line-continuation character when it's placed at the end of a line. If you want a backslash character to appear as a character literal, you must type two backslashes in a row (`\\`). For more information about the line continuation character, see [Phases of Translation](#).

## Microsoft-specific

To create a value from a narrow multicharacter literal, the compiler converts the character or character sequence between single quotes into 8-bit values within a 32-bit integer. Multiple characters in the literal fill corresponding bytes as needed from high-order to low-order. The compiler then converts the integer to the destination type following the usual rules. For example, to create a **char** value, the compiler takes the low-order byte. To create a **wchar\_t** or **char16\_t** value, the compiler takes the low-order word. The compiler warns that the result is truncated if any bits are set above the assigned byte or word.

```
char c0    = 'abcd';    // C4305, C4309, truncates to 'd'
wchar_t w0 = 'abcd';    // C4305, C4309, truncates to '\x6364'
int i0     = 'abcd';    // 0x61626364
```

An octal escape sequence that appears to contain more than three digits is treated as a 3-digit octal sequence, followed by the subsequent digits as characters in a multicharacter literal, which can give surprising results. For example:

```
char c1 = '\100';    // '@'
char c2 = '\1000';   // C4305, C4309, truncates to '0'
```

Escape sequences that appear to contain non-octal characters are evaluated as an octal sequence up to the last octal character, followed by the remaining characters as the subsequent characters in a multicharacter literal. Warning C4125 is generated if the first non-octal character is a decimal digit. For example:

```
char c3 = '\009';    // '9'
char c4 = '\089';    // C4305, C4309, truncates to '9'
char c5 = '\qrs';    // C4129, C4305, C4309, truncates to 's'
```

An octal escape sequence that has a higher value than `\377` causes error C2022: *'value-in-decimal': too big for character*.

An escape sequence that appears to have hexadecimal and non-hexadecimal characters is evaluated as a multicharacter literal that contains a hexadecimal escape sequence up to the last hexadecimal character, followed by the non-hexadecimal characters. A hexadecimal escape sequence that contains no hexadecimal digits causes compiler error C2153: "hex literals must have at least one hex digit".

```
char c6 = '\x0050';  // 'P'
char c7 = '\x0pqr';  // C4305, C4309, truncates to 'r'
```

If a wide character literal prefixed with **L** contains a multicharacter sequence, the value is taken from the first character, and the compiler raises warning C4066. Subsequent characters are ignored, unlike the behavior of the equivalent ordinary multicharacter literal.

```
wchar_t w1 = L'\100';    // L'@'
wchar_t w2 = L'\1000';   // C4066 L'@', 0 ignored
wchar_t w3 = L'\009';    // C4066 L'\0', 9 ignored
wchar_t w4 = L'\089';    // C4066 L'\0', 89 ignored
wchar_t w5 = L'\qrs';    // C4129, C4066 L'q' escape, rs ignored
wchar_t w6 = L'\x0050';  // L'P'
wchar_t w7 = L'\x0pqr';  // C4066 L'\0', pqr ignored
```

The **Microsoft-specific** section ends here.

## Universal character names

In character literals and native (non-raw) string literals, any character may be represented by a universal character name. Universal character names are formed by a prefix `\U` followed by an eight-digit Unicode code point, or by a prefix `\u` followed by a four-digit Unicode code point. All eight or four digits, respectively, must be present to make a well-formed universal character name.

```
char u1 = 'A';           // 'A'
char u2 = '\101';        // octal, 'A'
char u3 = '\x41';        // hexadecimal, 'A'
char u4 = '\u0041';      // \u UCN 'A'
char u5 = '\U00000041';  // \U UCN 'A'
```

## Surrogate Pairs

Universal character names can't encode values in the surrogate code point range D800-DFFF. For Unicode surrogate pairs, specify the universal character name by using `\UNNNNNNNNN`, where NNNNNNNNN is the eight-digit code point for the character. The compiler generates a surrogate pair if necessary.

In C++03, the language only allowed a subset of characters to be represented by their universal character names, and allowed some universal character names that didn't actually represent any valid Unicode characters. This mistake was fixed in the C++11 standard. In C++11, both character and string literals and identifiers can use universal character names. For more information on universal character names, see [Character Sets](#). For more information about Unicode, see [Unicode](#). For more information about surrogate pairs, see [Surrogate Pairs and Supplementary Characters](#).

## String literals

A string literal represents a sequence of characters that together form a null-terminated string. The characters must be enclosed between double quotation marks. There are the following kinds of string literals:

### Narrow string literals

A narrow string literal is a non-prefixed, double-quote delimited, null-terminated array of type `const char[n]`, where `n` is the length of the array in bytes. A narrow string literal may contain any graphic character except the double quotation mark (`"`), backslash (`\`), or newline character. A narrow string literal may also contain the escape sequences listed above, and universal character names that fit in a byte.

```
const char *narrow = "abcd";

// represents the string: yes\no
const char *escaped = "yes\\no";
```

## UTF-8 encoded strings

A UTF-8 encoded string is a u8-prefixed, double-quote delimited, null-terminated array of type `const char[n]`, where  $n$  is the length of the encoded array in bytes. A u8-prefixed string literal may contain any graphic character except the double quotation mark (`"`), backslash (`\`), or newline character. A u8-prefixed string literal may also contain the escape sequences listed above, and any universal character name.

```
const char* str1 = u8"Hello World";
const char* str2 = u8"\U0001F607 is 0:-)";
```

## Wide string literals

A wide string literal is a null-terminated array of constant `wchar_t` that is prefixed by `'L'` and contains any graphic character except the double quotation mark (`"`), backslash (`\`), or newline character. A wide string literal may contain the escape sequences listed above and any universal character name.

```
const wchar_t* wide = L"zyxw";
const wchar_t* newline = L"hello\ngoodbye";
```

## char16\_t and char32\_t (C++11)

C++11 introduces the portable `char16_t` (16-bit Unicode) and `char32_t` (32-bit Unicode) character types:

```
auto s3 = u"hello"; // const char16_t*
auto s4 = U"hello"; // const char32_t*
```

## Raw string literals (C++11)

A raw string literal is a null-terminated array—of any character type—that contains any graphic character, including the double quotation mark (`"`), backslash (`\`), or newline character. Raw string literals are often used in regular expressions that use character classes, and in HTML strings and XML strings. For examples, see the following article: [Bjarne Stroustrup's FAQ on C++11](#).

```
// represents the string: An unescaped \ character
const char* raw_narrow = R"(An unescaped \ character)";
const wchar_t* raw_wide = LR"(An unescaped \ character)";
```

```
const char*      raw_utf8  = u8R"(An unescaped \ character)";
const char16_t* raw_utf16 = uR"(An unescaped \ character)";
const char32_t* raw_utf32 = UR"(An unescaped \ character)";
```

A delimiter is a user-defined sequence of up to 16 characters that immediately precedes the opening parenthesis of a raw string literal, and immediately follows its closing parenthesis. For example, in `R"abc(Hello"\"()abc"` the delimiter sequence is `abc` and the string content is `Hello"\"()`. You can use a delimiter to disambiguate raw strings that contain both double quotation marks and parentheses. This string literal causes a compiler error:

```
// meant to represent the string: )"
const char* bad_parens = R"()"); // error C2059
```

But a delimiter resolves it:

```
const char* good_parens = R"xyz()")xyz";
```

You can construct a raw string literal that contains a newline (not the escaped character) in the source:

```
// represents the string: hello
//goodbye
const wchar_t* newline = LR"(hello
goodbye)";
```

## std::string literals (C++14)

`std::string` literals are Standard Library implementations of user-defined literals (see below) that are represented as `"xyz"s` (with a `s` suffix). This kind of string literal produces a temporary object of type `std::string`, `std::wstring`, `std::u32string`, or `std::u16string`, depending on the prefix that is specified. When no prefix is used, as above, a `std::string` is produced. `L"xyz"s` produces a `std::wstring`. `u"xyz"s` produces a `std::u16string`, and `U"xyz"s` produces a `std::u32string`.

```
//#include <string>
//using namespace std::string_literals;
string str{ "hello"s };
string str2{ u8"Hello World" };
wstring str3{ L"hello"s };
u16string str4{ u"hello"s };
u32string str5{ U"hello"s };
```

The `s` suffix may also be used on raw string literals:

```
u32string str6{ UR"(She said "hello.")"s };
```

`std::string` literals are defined in the namespace `std::literals::string_literals` in the `<string>` header file. Because `std::literals::string_literals`, and `std::literals` are both declared as `inline namespaces`, `std::literals::string_literals` is automatically treated as if it belonged directly in namespace `std`.

## Size of string literals

For ANSI `char*` strings and other single-byte encodings (but not UTF-8), the size (in bytes) of a string literal is the number of characters plus 1 for the terminating null character. For all other string types, the size isn't strictly related to the number of characters. UTF-8 uses up to four **char** elements to encode some *code units*, and `char16_t` or `wchar_t` encoded as UTF-16 may use two elements (for a total of four bytes) to encode a single *code unit*. This example shows the size of a wide string literal in bytes:

```
const wchar_t* str = L"Hello!";  
const size_t byteSize = (wcslen(str) + 1) * sizeof(wchar_t);
```

Notice that `strlen()` and `wcslen()` don't include the size of the terminating null character, whose size is equal to the element size of the string type: one byte on a `char*` or `char8_t*` string, two bytes on `wchar_t*` or `char16_t*` strings, and four bytes on `char32_t*` strings.

The maximum length of a string literal is 65,535 bytes. This limit applies to both narrow string literals and wide string literals.

## Modifying string literals

Because string literals (not including `std::string` literals) are constants, trying to modify them—for example, `str[2] = 'A'`—causes a compiler error.

### Microsoft-specific

In Microsoft C++, you can use a string literal to initialize a pointer to non-const **char** or **wchar\_t**. This non-const initialization is allowed in C99 code, but is deprecated in C++98 and removed in C++11. An attempt to modify the string causes an access violation, as in this example:

```
wchar_t* str = L"hello";  
str[2] = L'a'; // run-time error: access violation
```

You can cause the compiler to emit an error when a string literal is converted to a non-const character pointer when you set the `/Zc:strictStrings (Disable string literal type conversion)` compiler option. We recommend it for standards-compliant portable code. It's also a good practice to use the **auto** keyword to declare string literal-initialized pointers, because it resolves to the correct (const) type. For example, this code example catches an attempt to write to a string literal at compile time:

```
auto str = L"hello";  
str[2] = L'a'; // C3892: you cannot assign to a variable that is const.
```

In some cases, identical string literals may be pooled to save space in the executable file. In string-literal pooling, the compiler causes all references to a particular string literal to point to the same location in memory, instead of having each reference point to a separate instance of the string literal. To enable string pooling, use the `/GF` compiler option.

The **Microsoft-specific** section ends here.

## Concatenating adjacent string literals

Adjacent wide or narrow string literals are concatenated. This declaration:

```
char str[] = "12" "34";
```

is identical to this declaration:

```
char atr[] = "1234";
```

and to this declaration:

```
char atr[] = "12\  
34";
```

Using embedded hexadecimal escape codes to specify string literals can cause unexpected results. The following example seeks to create a string literal that contains the ASCII 5 character, followed by the characters f, i, v, and e:

```
"\x05five"
```

The actual result is a hexadecimal 5F, which is the ASCII code for an underscore, followed by the characters i, v, and e. To get the correct result, you can use one of these escape sequences:

```
"\005five"    // Use octal literal.  
"\x05" "five" // Use string splicing.
```

`std::string` literals, because they're `std::string` types, can be concatenated with the `+` operator that is defined for `basic_string` types. They can also be concatenated in the same way as adjacent string literals. In

both cases, the string encoding and the suffix must match:

```
auto x1 = "hello" " " " world"; // OK
auto x2 = U"hello" " " L"world"; // C2308: disagree on prefix
auto x3 = u8"hello" " "s u8"world"s; // OK, agree on prefixes and suffixes
auto x4 = u8"hello" " "s u8"world"z; // C3688, disagree on suffixes
```

## String literals with universal character names

Native (non-raw) string literals may use universal character names to represent any character, as long as the universal character name can be encoded as one or more characters in the string type. For example, a universal character name representing an extended character can't be encoded in a narrow string using the ANSI code page, but it can be encoded in narrow strings in some multi-byte code pages, or in UTF-8 strings, or in a wide string. In C++11, Unicode support is extended by the `char16_t*` and `char32_t*` string types:

```
// ASCII smiling face
const char* s1 = ":-)";

// UTF-16 (on Windows) encoded WINKING FACE (U+1F609)
const wchar_t* s2 = L"😜 = \U0001F609 is ;-)";

// UTF-8 encoded SMILING FACE WITH HALO (U+1F607)
const char* s3 = u8"😄 = \U0001F607 is 0:-)";

// UTF-16 encoded SMILING FACE WITH OPEN MOUTH (U+1F603)
const char16_t* s4 = u"😃 = \U0001F603 is :-D";

// UTF-32 encoded SMILING FACE WITH SUNGLASSES (U+1F60E)
const char32_t* s5 = U"😎 = \U0001F60E is B-)";
```

## See also

[Character sets](#)

[Numeric, Boolean, and pointer literals](#)

[User-Defined Literals](#)



# User-defined literals

---

There are six major categories of literals in C++: integer, character, floating-point, string, boolean, and pointer. Starting in C++ 11, you can define your own literals based on these categories, to provide syntactic shortcuts for common idioms and increase type safety. For example, let's say you have a `Distance` class. You could define a literal for kilometers and another one for miles, and encourage the user to be explicit about the units of measure by writing: `auto d = 42.0_km` or `auto d = 42.0_mi`. There's no performance advantage or disadvantage to user-defined literals; they're primarily for convenience or for compile-time type deduction. The Standard Library has user-defined literals for `std::string`, for `std::complex`, and for units in time and duration operations in the `<chrono>` header:

```
Distance d = 36.0_mi + 42.0_km;           // Custom UDL (see below)
std::string str = "hello"s + "World"s;    // Standard Library <string> UDL
complex<double> num =
    (2.0 + 3.01i) * (5.0 + 4.3i);          // Standard Library <complex> UDL
auto duration = 15ms + 42h;               // Standard Library <chrono> UDLs
```

## User-defined literal operator signatures

You implement a user-defined literal by defining an **operator""** at namespace scope with one of the following forms:

```
ReturnType operator "" _a(unsigned long long int); // Literal operator for user-
defined INTEGRAL literal
ReturnType operator "" _b(long double);           // Literal operator for user-
defined FLOATING literal
ReturnType operator "" _c(char);                   // Literal operator for user-
defined CHARACTER literal
ReturnType operator "" _d(wchar_t);               // Literal operator for user-
defined CHARACTER literal
ReturnType operator "" _e(char16_t);              // Literal operator for user-
defined CHARACTER literal
ReturnType operator "" _f(char32_t);              // Literal operator for user-
defined CHARACTER literal
ReturnType operator "" _g(const char*, size_t);    // Literal operator for user-
defined STRING literal
ReturnType operator "" _h(const wchar_t*, size_t); // Literal operator for user-
defined STRING literal
ReturnType operator "" _i(const char16_t*, size_t); // Literal operator for user-
defined STRING literal
ReturnType operator "" _g(const char32_t*, size_t); // Literal operator for user-
defined STRING literal
ReturnType operator "" _r(const char*);           // Raw literal operator
template<char...> ReturnType operator "" _t();    // Literal operator template
```

The operator names in the previous example are placeholders for whatever name you provide; however, the leading underscore is required. (Only the Standard Library is allowed to define literals without the underscore.) The return type is where you customize the conversion or other operations done by the literal. Also, any of these operators can be defined as `constexpr`.

## Cooked literals

In source code, any literal, whether user-defined or not, is essentially a sequence of alphanumeric characters, such as `101`, or `54.7`, or `"hello"` or `true`. The compiler interprets the sequence as an integer, float, const char\* string, and so on. A user-defined literal that accepts as input whatever type the compiler assigned to the literal value is informally known as a *cooked literal*. All the operators above except `_r` and `_t` are cooked literals. For example, a literal `42.0_km` would bind to an operator named `_km` that had a signature similar to `_b` and the literal `42_km` would bind to an operator with a signature similar to `_a`.

The following example shows how user-defined literals can encourage callers to be explicit about their input. To construct a `Distance`, the user must explicitly specify kilometers or miles by using the appropriate user-defined literal. You can achieve the same result in other ways, but user-defined literals are less verbose than the alternatives.

```
// UDL_Distance.cpp

#include <iostream>
#include <string>

struct Distance
{
private:
    explicit Distance(long double val) : kilometers(val)
    {}

    friend Distance operator"" _km(long double val);
    friend Distance operator"" _mi(long double val);

    long double kilometers{ 0 };
public:
    const static long double km_per_mile;
    long double get_kilometers() { return kilometers; }

    Distance operator+(Distance other)
    {
        return Distance(get_kilometers() + other.get_kilometers());
    }
};

const long double Distance::km_per_mile = 1.609344L;

Distance operator"" _km(long double val)
{
    return Distance(val);
}
```

```

Distance operator"" _mi(long double val)
{
    return Distance(val * Distance::km_per_mile);
}

int main()
{
    // Must have a decimal point to bind to the operator we defined!
    Distance d{ 402.0_km }; // construct using kilometers
    std::cout << "Kilometers in d: " << d.get_kilometers() << std::endl; // 402

    Distance d2{ 402.0_mi }; // construct using miles
    std::cout << "Kilometers in d2: " << d2.get_kilometers() << std::endl;
    //646.956

    // add distances constructed with different units
    Distance d3 = 36.0_mi + 42.0_km;
    std::cout << "d3 value = " << d3.get_kilometers() << std::endl; // 99.9364

    // Distance d4(90.0); // error constructor not accessible

    std::string s;
    std::getline(std::cin, s);
    return 0;
}

```

The literal number must use a decimal. Otherwise, the number would be interpreted as an integer, and the type wouldn't be compatible with the operator. For floating point input, the type must be **long double**, and for integral types it must be **long long**.

## Raw literals

In a raw user-defined literal, the operator that you define accepts the literal as a sequence of char values. It's up to you to interpret that sequence as a number or string or other type. In the list of operators shown earlier in this page, `_r` and `_t` can be used to define raw literals:

```

ReturnType operator "" _r(const char*);           // Raw literal operator
template<char...> ReturnType operator "" _t();     // Literal operator template

```

You can use raw literals to provide a custom interpretation of an input sequence that's different than the compiler's normal behavior. For example, you could define a literal that converts the sequence `4.75987` into a custom `Decimal` type instead of an IEEE 754 floating point type. Raw literals, like cooked literals, can also be used for compile-time validation of input sequences.

### Example: Limitations of raw literals

The raw literal operator and literal operator template only work for integral and floating-point user-defined literals, as shown by the following example:

```

#include <cstdint>
#include <cstdio>

// Literal operator for user-defined INTEGRAL literal
void operator "" _dump(unsigned long long int lit)
{
    printf("operator \"\" _dump(unsigned long long int) : ===>%llu<===\n", lit);
};

// Literal operator for user-defined FLOATING literal
void operator "" _dump(long double lit)
{
    printf("operator \"\" _dump(long double) : ===>%Lf<===\n", lit);
};

// Literal operator for user-defined CHARACTER literal
void operator "" _dump(char lit)
{
    printf("operator \"\" _dump(char) : ===>%c<===\n", lit);
};

void operator "" _dump(wchar_t lit)
{
    printf("operator \"\" _dump(wchar_t) : ===>%d<===\n", lit);
};

void operator "" _dump(char16_t lit)
{
    printf("operator \"\" _dump(char16_t) : ===>%d<===\n", lit);
};

void operator "" _dump(char32_t lit)
{
    printf("operator \"\" _dump(char32_t) : ===>%d<===\n", lit);
};

// Literal operator for user-defined STRING literal
void operator "" _dump(const char* lit, size_t)
{
    printf("operator \"\" _dump(const char*, size_t): ===>%s<===\n", lit);
};

void operator "" _dump(const wchar_t* lit, size_t)
{
    printf("operator \"\" _dump(const wchar_t*, size_t): ===>%ls<===\n", lit);
};

void operator "" _dump(const char16_t* lit, size_t)
{
    printf("operator \"\" _dump(const char16_t*, size_t):\n" );
};

void operator "" _dump(const char32_t* lit, size_t)

```

```

{
    printf("operator \"\" _dump(const char32_t*, size_t):\n"
);
};

// Raw literal operator
void operator "" _dump_raw(const char* lit)
{
    printf("operator \"\" _dump_raw(const char*) : ==>%s<==\n", lit);
};

template<char...> void operator "" _dump_template(); // Literal operator
template

int main(int argc, const char* argv[])
{
    42_dump;
    3.1415926_dump;
    3.14e+25_dump;
    'A'_dump;
    L'B'_dump;
    u'C'_dump;
    U'D'_dump;
    "Hello World"_dump;
    L"Wide String"_dump;
    u8"UTF-8 String"_dump;
    u"UTF-16 String"_dump;
    U"UTF-32 String"_dump;
    42_dump_raw;
    3.1415926_dump_raw;
    3.14e+25_dump_raw;

    // There is no raw literal operator or literal operator template support on
    these types:
    // 'A'_dump_raw;
    // L'B'_dump_raw;
    // u'C'_dump_raw;
    // U'D'_dump_raw;
    // "Hello World"_dump_raw;
    // L"Wide String"_dump_raw;
    // u8"UTF-8 String"_dump_raw;
    // u"UTF-16 String"_dump_raw;
    // U"UTF-32 String"_dump_raw;
}

```

```

operator "" _dump(unsigned long long int) : ==>42<==
operator "" _dump(long double) : ==>3.141593<==
operator "" _dump(long double) :
==>31399999999999998506827776.000000<==
operator "" _dump(char) : ==>A<==
operator "" _dump(wchar_t) : ==>66<==
operator "" _dump(char16_t) : ==>67<==

```

```
operator "" _dump(char32_t) : ===>68<===  
operator "" _dump(const char*, size_t): ===>Hello World<===  
operator "" _dump(const wchar_t*, size_t): ===>Wide String<===  
operator "" _dump(const char*, size_t): ===>UTF-8 String<===  
operator "" _dump(const char16_t*, size_t):  
operator "" _dump(const char32_t*, size_t):  
operator "" _dump_raw(const char*) : ===>42<===  
operator "" _dump_raw(const char*) : ===>3.1415926<===  
operator "" _dump_raw(const char*) : ===>3.14e+25<===
```

# Basic Concepts (C++)

---

This section explains concepts that are critical to understanding C++. C programmers will be familiar with many of these concepts, but there are some subtle differences that can cause unexpected program results. The following topics are included:

- [C++ type system](#)
- [Scope](#)
- [Translation units and linkage](#)
- [main function and command-line arguments](#)
- [Program termination](#)
- [Lvalues and rvalues](#)
- [Temporary objects](#)
- [Alignment](#)
- [Trivial, standard-layout and POD types](#)

## See also

[C++ Language Reference](#)

# C++ type system

---

The concept of *type* is very important in C++. Every variable, function argument, and function return value must have a type in order to be compiled. Also, every expression (including literal values) is implicitly given a type by the compiler before it is evaluated. Some examples of types include **int** to store integral values, **double** to store floating-point values (also known as *scalar* data types), or the Standard Library class [std::basic\\_string](#) to store text. You can create your own type by defining a **class** or **struct**. The type specifies the amount of memory that will be allocated for the variable (or expression result), the kinds of values that may be stored in that variable, how those values (as bit patterns) are interpreted, and the operations that can be performed on it. This article contains an informal overview of the major features of the C++ type system.

## Terminology

**Variable:** The symbolic name of a quantity of data so that the name can be used to access the data it refers to throughout the scope of the code where it is defined. In C++, *variable* is generally used to refer to instances of scalar data types, whereas instances of other types are usually called *objects*.

**Object:** For simplicity and consistency, this article uses the term *object* to refer to any instance of a class or structure, and when it is used in the general sense includes all types, even scalar variables.

**POD type** (plain old data): This informal category of data types in C++ refers to types that are scalar (see the Fundamental types section) or are *POD classes*. A POD class has no static data members that aren't also PODs, and has no user-defined constructors, user-defined destructors, or user-defined assignment operators. Also, a POD class has no virtual functions, no base class, and no private or protected non-static data members. POD types are often used for external data interchange, for example with a module written in the C language (which has POD types only).

## Specifying variable and function types

C++ is a *strongly typed* language and it is also *statically-typed*; every object has a type and that type never changes (not to be confused with static data objects). When you declare a variable in your code, you must either specify its type explicitly, or use the **auto** keyword to instruct the compiler to deduce the type from the initializer. When you declare a function in your code, you must specify the type of each argument and its return value, or **void** if no value is returned by the function. The exception is when you are using function templates, which allow for arguments of arbitrary types.

After you first declare a variable, you cannot change its type at some later point. However, you can copy the variable's value or a function's return value into another variable of a different type. Such operations are called *type conversions*, which are sometimes necessary but are also potential sources of data loss or incorrectness.

When you declare a variable of POD type, we strongly recommend you initialize it, which means to give it an initial value. Until you initialize a variable, it has a "garbage" value that consists of whatever bits happened to be in that memory location previously. This is an important aspect of C++ to remember, especially if you are coming from another language that handles initialization for you. When declaring a variable of non-POD class type, the constructor handles initialization.



The following example shows some simple variable declarations with some descriptions for each. The example also shows how the compiler uses type information to allow or disallow certain subsequent operations on the variable.

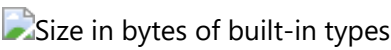
```
int result = 0;           // Declare and initialize an integer.
double coefficient = 10.8; // Declare and initialize a floating
                           // point value.
auto name = "Lady G.";    // Declare a variable and let compiler
                           // deduce the type.
auto address;             // error. Compiler cannot deduce a type
                           // without an initializing value.
age = 12;                 // error. Variable declaration must
                           // specify a type or use auto!
result = "Kenny G.";      // error. Can't assign text to an int.
string result = "zero";   // error. Can't redefine a variable with
                           // new type.
int maxValue;             // Not recommended! maxValue contains
                           // garbage bits until it is initialized.
```

## Fundamental (built-in) types

Unlike some languages, C++ has no universal base type from which all other types are derived. The language includes many *fundamental types*, also known as *built-in types*. This includes numeric types such as **int**, **double**, **long**, **bool**, plus the **char** and **wchar\_t** types for ASCII and UNICODE characters, respectively. Most fundamental types (except **bool**, **double**, **wchar\_t** and related types) all have unsigned versions, which modify the range of values that the variable can store. For example, an **int**, which stores a 32-bit signed integer, can represent a value from -2,147,483,648 to 2,147,483,647. An **unsigned int**, which is also stored as 32-bits, can store a value from 0 to 4,294,967,295. The total number of possible values in each case is the same; only the range is different.

The fundamental types are recognized by the compiler, which has built-in rules that govern what operations you can perform on them, and how they can be converted to other fundamental types. For a complete list of built-in types and their size and numeric limits, see [Built-in types](#).

The following illustration shows the relative sizes of the built-in types:



The following table lists the most frequently used fundamental types:

Type	Size	Comment
int	4 bytes	The default choice for integral values.
double	8 bytes	The default choice for floating point values.
bool	1 byte	Represents values that can be either true or false.

Type	Size	Comment
char	1 byte	Use for ASCII characters in older C-style strings or std::string objects that will never have to be converted to UNICODE.
wchar_t	2 bytes	Represents "wide" character values that may be encoded in UNICODE format (UTF-16 on Windows, other operating systems may differ). This is the character type that is used in strings of type <code>std::wstring</code> .
unsigned char	1 byte	C++ has no built-in <code>byte</code> type. Use unsigned char to represent a byte value.
unsigned int	4 bytes	Default choice for bit flags.
long long	8 bytes	Represents very large integer values.

## The void type

The **void** type is a special type; you cannot declare a variable of type **void**, but you can declare a variable of type **void \*** (pointer to **void**), which is sometimes necessary when allocating raw (un-typed) memory. However, pointers to **void** are not type-safe and generally their use is strongly discouraged in modern C++. In a function declaration, a **void** return value means that the function does not return a value; this is a common and acceptable use of **void**. While the C language required functions that have zero parameters to declare **void** in the parameter list, for example, `fou(void)`, this practice is discouraged in modern C++ and should be declared `fou()`. For more information, see [Type Conversions and Type Safety](#).

## const type qualifier

Any built-in or user-defined type may be qualified by the `const` keyword. Additionally, member functions may be **const**-qualified and even **const**-overloaded. The value of a **const** type cannot be modified after it is initialized.

```
const double PI = 3.1415;
PI = .75 //Error. Cannot modify const variable.
```

The **const** qualifier is used extensively in function and variable declarations and "const correctness" is an important concept in C++; essentially it means to use **const** to guarantee, at compile time, that values are not modified unintentionally. For more information, see [const](#).

A **const** type is distinct from its non-const version; for example, **const int** is a distinct type from **int**. You can use the C++ **const\_cast** operator on those rare occasions when you must remove *const-ness* from a variable. For more information, see [Type Conversions and Type Safety](#).

## String types

Strictly speaking, the C++ language has no built-in string type; **char** and **wchar\_t** store single characters - you must declare an array of these types to approximate a string, adding a terminating null value (for example, ASCII `'\0'`) to the array element one past the last valid character (also called a *C-style string*). C-style strings required much more code to be written or the use of external string utility library functions. But in modern C++, we have the Standard Library types `std::string` (for 8-bit **char**-type character strings) or `std::wstring` (for 16-bit **wchar\_t**-type character strings). These C++ Standard Library containers can be thought of as native string types because they are part of the standard libraries that are included in any compliant C++ build environment. Simply use the `#include <string>` directive to make these types available in your program. (If you are using MFC or ATL, the `CString` class is also available, but is not part of the C++ standard.) The use of null-terminated character arrays (the C-style strings previously mentioned) is strongly discouraged in modern C++.

## User-defined types

When you define a **class**, **struct**, **union**, or **enum**, that construct is used in the rest of your code as if it were a fundamental type. It has a known size in memory, and certain rules about how it can be used apply to it for compile-time checking and, at runtime, for the life of your program. The primary differences between the fundamental built-in types and user-defined types are as follows:

- The compiler has no built-in knowledge of a user-defined type. It learns of the type when it first encounters the definition during the compilation process.
- You specify what operations can be performed on your type, and how it can be converted to other types, by defining (through overloading) the appropriate operators, either as class members or non-member functions. For more information, see [Function Overloading](#)

## Pointer types

Dating back to the earliest versions of the C language, C++ continues to let you declare a variable of a pointer type by using the special declarator `*` (asterisk). A pointer type stores the address of the location in memory where the actual data value is stored. In modern C++, these are referred to as *raw pointers*, and are accessed in your code through special operators `*` (asterisk) or `->` (dash with greater-than). This is called *dereferencing*, and which one that you use depends on whether you are dereferencing a pointer to a scalar or a pointer to a member in an object. Working with pointer types has long been one of the most challenging and confusing aspects of C and C++ program development. This section outlines some facts and practices to help use raw pointers if you want to, but in modern C++ it's no longer required (or recommended) to use raw pointers for object ownership at all, due to the evolution of the [smart pointer](#) (discussed more at the end of this section). It is still useful and safe to use raw pointers for observing objects, but if you must use them for object ownership, you should do so with caution and very careful consideration of how the objects owned by them are created and destroyed.

The first thing that you should know is declaring a raw pointer variable will allocate only the memory that is required to store an address of the memory location that the pointer will be referring to when it is dereferenced. Allocation of the memory for the data value itself (also called *backing store*) is not yet allocated. In other words, by declaring a raw pointer variable, you are creating a memory address variable, not an actual data variable. Dereferencing a pointer variable before making sure that it contains a valid address to a backing store will cause undefined behavior (usually a fatal error) in your program. The following example demonstrates this kind of error:

---

```
int* pNumber;           // Declare a pointer-to-int variable.
*pNumber = 10;          // error. Although this may compile, it is
                        // a serious error. We are dereferencing an
                        // uninitialized pointer variable with no
                        // allocated memory to point to.
```

The example dereferences a pointer type without having any memory allocated to store the actual integer data or a valid memory address assigned to it. The following code corrects these errors:

```
int number = 10;         // Declare and initialize a local integer
                        // variable for data backing store.
int* pNumber = &number; // Declare and initialize a local integer
                        // pointer variable to a valid memory
                        // address to that backing store.
...
*pNumber = 41;           // Dereference and store a new value in
                        // the memory pointed to by
                        // pNumber, the integer variable called
                        // "number". Note "number" was changed, not
                        // "pNumber".
```

The corrected code example uses local stack memory to create the backing store that `pNumber` points to. We use a fundamental type for simplicity. In practice, the backing store for pointers are most often user-defined types that are dynamically-allocated in an area of memory called the *heap* (or *free store*) by using a **new** keyword expression (in C-style programming, the older `malloc()` C runtime library function was used). Once allocated, these variables are usually referred to as objects, especially if they are based on a class definition. Memory that is allocated with **new** must be deleted by a corresponding **delete** statement (or, if you used the `malloc()` function to allocate it, the C runtime function `free()`).

However, it is easy to forget to delete a dynamically-allocated object- especially in complex code, which causes a resource bug called a *memory leak*. For this reason, the use of raw pointers is strongly discouraged in modern C++. It is almost always better to wrap a raw pointer in a [smart pointer](#), which will automatically release the memory when its destructor is invoked (when the code goes out of scope for the smart pointer); by using smart pointers you virtually eliminate a whole class of bugs in your C++ programs. In the following example, assume `MyClass` is a user-defined type that has a public method `DoSomeWork()`;

```
void someFunction() {
    unique_ptr<MyClass> pMc(new MyClass);
    pMc->DoSomeWork();
}
// No memory leak. Out-of-scope automatically calls the destructor
// for the unique_ptr, freeing the resource.
```

For more information about smart pointers, see [Smart Pointers](#).

For more information about pointer conversions, see [Type Conversions and Type Safety](#).

For more information about pointers in general, see [Pointers](#).

## Windows data types

In classic Win32 programming for C and C++, most functions use Windows-specific typedefs and #define macros (defined in `windows.h`) to specify the types of parameters and return values. These Windows data types are mostly just special names (aliases) given to C/C++ built-in types. For a complete list of these typedefs and preprocessor definitions, see [Windows Data Types](#). Some of these typedefs, such as HRESULT and LCID, are useful and descriptive. Others, such as INT, have no special meaning and are just aliases for fundamental C++ types. Other Windows data types have names that are retained from the days of C programming and 16-bit processors, and have no purpose or meaning on modern hardware or operating systems. There are also special data types associated with the Windows Runtime Library, listed as [Windows Runtime base data types](#). In modern C++, the general guideline is to prefer the C++ fundamental types unless the Windows type communicates some additional meaning about how the value is to be interpreted.

## More Information

For more information about the C++ type system, see the following topics.

<a href="#">Value Types</a>	Describes <i>value types</i> along with issues relating to their use.
<a href="#">Type Conversions and Type Safety</a>	Describes common type conversion issues and shows how to avoid them.

## See also

- [Welcome back to C++](#)
- [C++ Language Reference](#)
- [C++ Standard Library](#)

# Scope (C++)

---

When you declare a program element such as a class, function, or variable, its name can only be "seen" and used in certain parts of your program. The context in which a name is visible is called its *scope*. For example, if you declare a variable `x` within a function, `x` is only visible within that function body. It has *local scope*. You may have other variables by the same name in your program; as long as they are in different scopes, they do not violate the One Definition Rule and no error is raised.

For automatic non-static variables, scope also determines when they are created and destroyed in program memory.

There are six kinds of scope:

- **Global scope** A global name is one that is declared outside of any class, function or namespace. However, in C++ even these names exist with an implicit global namespace. The scope of global names extends from the point of declaration to the end of the file in which they are declared. For global names, visibility is also governed by the rules of [linkage](#) which determine whether the name is visible in other files in the program.
- **Namespace scope** A name that is declared within a [namespace](#), outside of any class or enum definition or function block, is visible from its point of declaration to the end of namespace. A namespace may be defined in multiple blocks across different files.
- **Local scope** A name declared within a function or lambda, including the parameter names, have local scope. They are often referred to as "locals". They are only visible from their point of declaration to the end of the function or lambda body. Local scope is a kind of block scope, which is discussed later in this article.
- **Class scope** Names of class members have class scope, which extends throughout the class definition regardless of the point of declaration. Class member accessibility is further controlled by the **public**, **private**, and **protected** keywords. Public or protected members can be accessed only by using the member-selection operators (`.` or `->`) or pointer-to-member operators (`.*` or `->*`).
- **Statement scope** Names declared in a **for**, **if**, **while**, or **switch** statement are visible until the end of the statement block.
- **Function scope** A [label](#) has function scope, which means it is visible throughout a function body even before its point of declaration. Function scope makes it possible to write statements like `goto cleanup` before the `cleanup` label is declared.

## Hiding Names

You can hide a name by declaring it in an enclosed block. In the following figure, `i` is redeclared within the inner block, thereby hiding the variable associated with `i` in the outer block scope.

Block-scope name hiding

Block scope and name hiding

The output from the program shown in the figure is:

```
i = 0
i = 7
j = 9
i = 0
```

[!NOTE] The argument `szWhat` is considered to be in the scope of the function. Therefore, it is treated as if it had been declared in the outermost block of the function.

## Hiding class names

You can hide class names by declaring a function, object or variable, or enumerator in the same scope. However, the class name can still be accessed when prefixed by the keyword **class**.

```
// hiding_class_names.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

// Declare class Account at global scope.
class Account
{
public:
    Account( double InitialBalance )
        { balance = InitialBalance; }
    double GetBalance()
        { return balance; }
private:
    double balance;
};

double Account = 15.37;           // Hides class name Account

int main()
{
    class Account Checking( Account ); // Qualifies Account as
                                        // class name

    cout << "Opening account with balance of: "
          << Checking.GetBalance() << "\n";
}
//Output: Opening account with balance of: 15.37
```

[!NOTE] Any place the class name (`Account`) is called for, the keyword `class` must be used to differentiate it from the global-scoped variable `Account`. This rule does not apply when the class name occurs on the left side of the scope-resolution operator (`::`). Names on the left side of the scope-resolution operator are always considered class names.

The following example demonstrates how to declare a pointer to an object of type `Account` using the **class** keyword:

```
class Account *Checking = new class Account( Account );
```

The `Account` in the initializer (in parentheses) in the preceding statement has global scope; it is of type **double**.

[!NOTE] The reuse of identifier names as shown in this example is considered poor programming style.

For information about declaration and initialization of class objects, see [Classes, Structures, and Unions](#). For information about using the **new** and **delete** free-store operators, see [new and delete operators](#).

## Hiding names with global scope

You can hide names with global scope by explicitly declaring the same name in block scope. However, global-scope names can be accessed using the scope-resolution operator (`::`).

```
#include <iostream>

int i = 7;    // i has global scope, outside all blocks
using namespace std;

int main( int argc, char *argv[] ) {
    int i = 5;    // i has block scope, hides i at global scope
    cout << "Block-scoped i has the value: " << i << "\n";
    cout << "Global-scoped i has the value: " << ::i << "\n";
}
```

```
Block-scoped i has the value: 5
Global-scoped i has the value: 7
```

## See also

[Basic Concepts](#)



# Header files (C++)

---

The names of program elements such as variables, functions, classes, and so on must be declared before they can be used. For example, you can't just write `x = 42` without first declaring 'x'.

```
int x; // declaration
x = 42; // use x
```

The declaration tells the compiler whether the element is an **int**, a **double**, a **function**, a **class** or some other thing. Furthermore, each name must be declared (directly or indirectly) in every .cpp file in which it is used. When you compile a program, each .cpp file is compiled independently into a compilation unit. The compiler has no knowledge of what names are declared in other compilation units. That means that if you define a class or function or global variable, you must provide a declaration of that thing in each additional .cpp file that uses it. Each declaration of that thing must be exactly identical in all files. A slight inconsistency will cause errors, or unintended behavior, when the linker attempts to merge all the compilation units into a single program.

To minimize the potential for errors, C++ has adopted the convention of using *header files* to contain declarations. You make the declarations in a header file, then use the `#include` directive in every .cpp file or other header file that requires that declaration. The `#include` directive inserts a copy of the header file directly into the .cpp file prior to compilation.

[!NOTE] In Visual Studio 2019, the C++20 *modules* feature is introduced as an improvement and eventual replacement for header files. For more information, see [Overview of modules in C++](#).

## Example

The following example shows a common way to declare a class and then use it in a different source file. We'll start with the header file, `my_class.h`. It contains a class definition, but note that the definition is incomplete; the member function `do_something` is not defined:

```
// my_class.h
namespace N
{
    class my_class
    {
    public:
        void do_something();
    };
}
```

Next, create an implementation file (typically with a .cpp or similar extension). We'll call the file `my_class.cpp` and provide a definition for the member declaration. We add an `#include` directive for "my\_class.h" file in order to have the `my_class` declaration inserted at this point in the .cpp file, and we include `<iostream>` to

pull in the declaration for `std::cout`. Note that quotes are used for header files in the same directory as the source file, and angle brackets are used for standard library headers. Also, many standard library headers do not have `.h` or any other file extension.

In the implementation file, we can optionally use a **using** statement to avoid having to qualify every mention of "my\_class" or "cout" with "N::" or "std::". Don't put **using** statements in your header files!

```
// my_class.cpp
#include "my_class.h" // header in local directory
#include <iostream> // header in standard library

using namespace N;
using namespace std;

void my_class::do_something()
{
    cout << "Doing something!" << endl;
}
```

Now we can use `my_class` in another `.cpp` file. We `#include` the header file so that the compiler pulls in the declaration. All the compiler needs to know is that `my_class` is a class that has a public member function called `do_something()`.

```
// my_program.cpp
#include "my_class.h"

using namespace N;

int main()
{
    my_class mc;
    mc.do_something();
    return 0;
}
```

After the compiler finishes compiling each `.cpp` file into `.obj` files, it passes the `.obj` files to the linker. When the linker merges the object files it finds exactly one definition for `my_class`; it is in the `.obj` file produced for `my_class.cpp`, and the build succeeds.

## Include guards

Typically, header files have an *include guard* or a `#pragma once` directive to ensure that they are not inserted multiple times into a single `.cpp` file.

```
// my_class.h
#ifndef MY_CLASS_H // include guard
#define MY_CLASS_H
```

```

namespace N
{
    class my_class
    {
    public:
        void do_something();
    };
}

#endif /* MY_CLASS_H */

```

## What to put in a header file

Because a header file might potentially be included by multiple files, it cannot contain definitions that might produce multiple definitions of the same name. The following are not allowed, or are considered very bad practice:

- built-in type definitions at namespace or global scope
- non-inline function definitions
- non-const variable definitions
- aggregate definitions
- unnamed namespaces
- using directives

Use of the **using** directive will not necessarily cause an error, but can potentially cause a problem because it brings the namespace into scope in every .cpp file that directly or indirectly includes that header.

## Sample header file

The following example shows the various kinds of declarations and definitions that are allowed in a header file:

```

#pragma once
#include <vector> // #include directive
#include <string>

namespace N // namespace declaration
{
    inline namespace P
    {
        //...
    }

    enum class colors : short { red, blue, purple, azure };

    const double PI = 3.14; // const and constexpr definitions
    constexpr int MeaningOfLife{ 42 };
    constexpr int get_meaning()
    {

```

```

        static_assert(MeaningOfLife == 42, "unexpected!"); // static_assert
        return MeaningOfLife;
    }
    using vstr = std::vector<int>; // type alias
    extern double d; // extern variable

#define LOG    // macro definition

#ifdef LOG    // conditional compilation directive
    void print_to_log();
#endif

class my_class    // regular class definition,
{                // but no non-inline function definitions

    friend class other_class;
public:
    void do_something();    // definition in my_class.cpp
    inline void put_value(int i) { vals.push_back(i); } // inline OK

private:
    vstr vals;
    int i;
};

struct RGB
{
    short r{ 0 }; // member initialization
    short g{ 0 };
    short b{ 0 };
};

template <typename T> // template definition
class value_store
{
public:
    value_store<T>() = default;
    void write_value(T val)
    {
        //... function definition OK in template
    }
private:
    std::vector<T> vals;
};

template <typename T> // template declaration
class value_widget;
}

```

# Translation units and linkage

---

In a C++ program, a *symbol*, for example a variable or function name, can be declared any number of times within its scope, but it can only be defined once. This rule is the "One Definition Rule" (ODR). A *declaration* introduces (or re-introduces) a name into the program. A *definition* introduces a name. If the name represents a variable, a definition explicitly initializes it. A *function definition* consists of the signature plus the function body. A class definition consists of the class name followed by a block that lists all the class members. (The bodies of member functions may optionally be defined separately in another file.)

The following example shows some declarations:

```
int i;
int f(int x);
class C;
```

The following example shows some definitions:

```
int i{42};
int f(int x){ return x * i; }
class C {
public:
    void DoSomething();
};
```

A program consists of one or more *translation units*. A translation unit consists of an implementation file and all the headers that it includes directly or indirectly. Implementation files typically have a file extension of *cpp* or *cxx*. Header files typically have an extension of *h* or *hpp*. Each translation unit is compiled independently by the compiler. After the compilation is complete, the linker merges the compiled translation units into a single *program*. Violations of the ODR rule typically show up as linker errors. Linker errors occur when the same name has two different definitions in different translation units.

In general, the best way to make a variable visible across multiple files is to put it in a header file. Then add an `#include` directive in every *cpp* file that requires the declaration. By adding *include guards* around the header contents, you ensure that the names it declares are only defined once.

In C++20, [modules](#) are introduced as an improved alternative to header files.

In some cases it may be necessary to declare a global variable or class in a *cpp* file. In those cases, you need a way to tell the compiler and linker what kind of *linkage* the name has. The type of linkage specifies whether the name of the object applies just to the one file, or to all files. The concept of linkage applies only to global names. The concept of linkage does not apply to names that are declared within a scope. A scope is specified by a set of enclosing braces such as in function or class definitions.

## External vs. internal linkage

A *free function* is a function that is defined at global or namespace scope. Non-const global variables and free functions by default have *external linkage*; they are visible from any translation unit in the program. Therefore, no other global object can have that name. A symbol with *internal linkage* or *no linkage* is visible only within the translation unit in which it is declared. When a name has internal linkage, the same name may exist in another translation unit. Variables declared with class definitions or function bodies have no linkage.

You can force a global name to have internal linkage by explicitly declaring it as **static**. This limits its visibility to the same translation unit in which it is declared. In this context, **static** means something different than when applied to local variables.

The following objects have internal linkage by default:

- const objects
- constexpr objects
- typedefs
- static objects in namespace scope

To give a const object external linkage, declare it as **extern** and assign it a value:

```
extern const int value = 42;
```

See [extern](#) for more information.

## See also

[Basic Concepts](#)

# main function and command-line arguments

---

All C++ programs must have a `main` function. If you try to compile a C++ .exe project without a `main` function, the compiler will raise an error. (Dynamic-link libraries and static libraries don't have a `main` function.) The `main` function is where your source code begins execution, but before a program enters the `main` function, all static class members without explicit initializers are set to zero. In Microsoft C++, global static objects are also initialized before entry to `main`. Several restrictions apply to the `main` function that do not apply to any other C++ functions. The `main` function:

- Cannot be overloaded (see [Function Overloading](#)).
- Cannot be declared as **inline**.
- Cannot be declared as **static**.
- Cannot have its address taken.
- Cannot be called.

The `main` function doesn't have a declaration, because it's built into the language. If it did, the declaration syntax for `main` would look like this:

```
int main();  
int main(int argc, char *argv[], char *envp[]);
```

## Microsoft Specific

If your source files use Unicode wide characters, you can use `wmain`, which is the wide-character version of `main`. The declaration syntax for `wmain` is as follows:

```
int wmain( );  
int wmain(int argc, wchar_t *argv[], wchar_t *envp[]);
```

You can also use `_tmain`, which is defined in `tchar.h`. `_tmain` resolves to `main` unless `_UNICODE` is defined. In that case, `_tmain` resolves to `wmain`.

If no return value is specified, the compiler supplies a return value of zero. Alternatively, the `main` and `wmain` functions can be declared as returning **void** (no return value). If you declare `main` or `wmain` as returning **void**, you cannot return an exit code to the parent process or operating system by using a `return` statement. To return an exit code when `main` or `wmain` is declared as **void**, you must use the `exit` function.

## END Microsoft Specific

## Command line arguments

The arguments for `main` or `wmain` allow convenient command-line parsing of arguments and, optionally, access to environment variables. The types for `argc` and `argv` are defined by the language. The names `argc`, `argv`, and `envp` are traditional, but you can name them whatever you like.

```
int main( int argc, char* argv[], char* envp[]);
int wmain( int argc, wchar_t* argv[], wchar_t* envp[]);
```

The argument definitions are as follows:

#### *argc*

An integer that contains the count of arguments that follow in *argv*. The *argc* parameter is always greater than or equal to 1.

#### *argv*

An array of null-terminated strings representing command-line arguments entered by the user of the program. By convention, *argv[0]* is the command with which the program is invoked, *argv[1]* is the first command-line argument, and so on, until *argv[argc]*, which is always NULL. See [Customizing Command Line Processing](#) for information on suppressing command-line processing.

The first command-line argument is always *argv[1]* and the last one is *argv[argc - 1]*.

[!NOTE] By convention, *argv[0]* is the command with which the program is invoked. However, it is possible to spawn a process using [CreateProcess](#) and if you use both the first and second arguments (*lpApplicationName* and *lpCommandLine*), *argv[0]* may not be the executable name; use [GetModuleFileName](#) to retrieve the executable name, and its fully-qualified path.

### Microsoft Specific

#### *envp*

The *envp* array, which is a common extension in many UNIX systems, is used in Microsoft C++. It is an array of strings representing the variables set in the user's environment. This array is terminated by a NULL entry. It can be declared as an array of pointers to **char** (*char \*envp[]*) or as a pointer to pointers to **char** (*char \*\*envp*). If your program uses *wmain* instead of *main*, use the **wchar\_t** data type instead of **char**. The environment block passed to *main* and *wmain* is a "frozen" copy of the current environment. If you subsequently change the environment via a call to *putenv* or *wputenv*, the current environment (as returned by *getenv* or *wgetenv* and the *\_environ* or *wenviron* variable) will change, but the block pointed to by *envp* will not change. See [Customizing Command Line Processing](#) for information on suppressing environment processing. This argument is ANSI compatible in C, but not in C++.

### END Microsoft Specific

### Example

The following example shows how to use the *argc*, *argv*, and *envp* arguments to *main*:

```
// argument_definitions.cpp
// compile with: /EHsc
#include <iostream>
#include <string.h>

using namespace std;
int main( int argc, char *argv[], char *envp[] ) {
    int iNumberLines = 0;    // Default is no line numbers.
```



```

// If /n is passed to the .exe, display numbered listing
// of environment variables.

if ( (argc == 2) && _stricmp( argv[1], "/n" ) == 0 )
    iNumberLines = 1;

// Walk through list of strings until a NULL is encountered.
for( int i = 0; envp[i] != NULL; ++i ) {
    if( iNumberLines )
        cout << i << ": " << envp[i] << "\n";
    }
}

```

## Parsing C++ command-Line arguments

### Microsoft Specific

Microsoft C/C++ startup code uses the following rules when interpreting arguments given on the operating system command line:

- Arguments are delimited by white space, which is either a space or a tab.
- The caret character (^) is not recognized as an escape character or delimiter. The character is handled completely by the command-line parser in the operating system before being passed to the `argv` array in the program.
- A string surrounded by double quotation marks ("*string*") is interpreted as a single argument, regardless of white space contained within. A quoted string can be embedded in an argument.
- A double quotation mark preceded by a backslash (\") is interpreted as a literal double quotation mark character (").
- Backslashes are interpreted literally, unless they immediately precede a double quotation mark.
- If an even number of backslashes is followed by a double quotation mark, one backslash is placed in the `argv` array for every pair of backslashes, and the double quotation mark is interpreted as a string delimiter.
- If an odd number of backslashes is followed by a double quotation mark, one backslash is placed in the `argv` array for every pair of backslashes, and the double quotation mark is "escaped" by the remaining backslash, causing a literal double quotation mark (") to be placed in `argv`.

### Example

The following program demonstrates how command-line arguments are passed:

```

// command_line_arguments.cpp
// compile with: /EHsc
#include <iostream>

```

```
using namespace std;
int main( int argc,      // Number of strings in array argv
         char *argv[],   // Array of command-line argument strings
         char *envp[] )  // Array of environment variable strings
{
    int count;

    // Display each command-line argument.
    cout << "\nCommand-line arguments:\n";
    for( count = 0; count < argc; count++ )
        cout << "  argv[" << count << "]  "
              << argv[count] << "\n";
}
```

The following table shows example input and expected output, demonstrating the rules in the preceding list.

Results of parsing command lines

Command-Line Input	argv[1]	argv[2]	argv[3]
"abc" d e	abc	d	e
a\\b d"e f"g h	a\\b	de fg	h
a\\\\"b c d	a\\"b	c	d
a\\\\"b c" d e	a\\b c	d	e

END Microsoft Specific

Wildcard expansion

Microsoft Specific

You can use wildcards — the question mark (?) and asterisk (\*) — to specify filename and path arguments on the command-line.

Command-line arguments are handled by a routine called `_setargv` (or `_wsetargv` in the wide-character environment), which by default does not expand wildcards into separate strings in the `argv` string array. For more information on enabling wildcard expansion, refer to [Expanding Wildcard Arguments](#).

END Microsoft Specific

Customizing C++ command-line processing

Microsoft Specific

If your program does not take command-line arguments, you can save a small amount of space by suppressing use of the library routine that performs command-line processing. This routine is called `_setargv` and is described in [Wildcard Expansion](#). To suppress its use, define a routine that does nothing in the file containing the `main` function, and name it `_setargv`. The call to `_setargv` is then satisfied by your definition of `_setargv`, and the library version is not loaded.

Similarly, if you never access the environment table through the `envp` argument, you can provide your own empty routine to be used in place of `_setenvp`, the environment-processing routine. Just as with the `_setargv` function, `_setenvp` must be declared as **extern "C"**.

Your program might make calls to the `spawn` or `exec` family of routines in the C run-time library. If it does, you shouldn't suppress the environment-processing routine, since this routine is used to pass an environment from the parent process to the child process.

#### **END Microsoft Specific**

## See also

[Basic Concepts](#)

# C++ program termination

---

In C++, you can exit a program in these ways:

- Call the `exit` function.
- Call the `abort` function.
- Execute a `return` statement from `main`.

## exit function

The `exit` function, declared in `<stdlib.h>`, terminates a C++ program. The value supplied as an argument to `exit` is returned to the operating system as the program's return code or exit code. By convention, a return code of zero means that the program completed successfully. You can use the constants `EXIT_FAILURE` and `EXIT_SUCCESS`, also defined in `<stdlib.h>`, to indicate success or failure of your program.

Issuing a **`return`** statement from the `main` function is equivalent to calling the `exit` function with the return value as its argument.

## abort function

The `abort` function, also declared in the standard include file `<stdlib.h>`, terminates a C++ program. The difference between `exit` and `abort` is that `exit` allows the C++ run-time termination processing to take place (global object destructors will be called), whereas `abort` terminates the program immediately. The `abort` function bypasses the normal destruction process for initialized global static objects. It also bypasses any special processing that was specified using the `atexit` function.

## atexit function

Use the `atexit` function to specify actions that execute prior to program termination. No global static objects initialized prior to the call to **`atexit`** are destroyed prior to execution of the exit-processing function.

## return statement in main

Issuing a `return` statement from `main` is functionally equivalent to calling the `exit` function. Consider the following example:

```
// return_statement.cpp
#include <stdlib.h>
int main()
{
    exit( 3 );
    return 3;
}
```

The `exit` and **`return`** statements in the preceding example are functionally identical. However, C++ requires that functions that have return types other than **`void`** return a value. The **`return`** statement allows you to return a value from `main`.

## Destruction of static objects

When you call `exit` or execute a `return` statement from `main`, static objects are destroyed in the reverse order of their initialization (after the call to `atexit` if one exists). The following example shows how such initialization and cleanup works.

### Example

In the following example, the static objects `sd1` and `sd2` are created and initialized before entry to `main`. After this program terminates using the `return` statement, first `sd2` is destroyed and then `sd1`. The destructor for the `ShowData` class closes the files associated with these static objects.

```
// using_exit_or_return1.cpp
#include <stdio.h>
class ShowData {
public:
    // Constructor opens a file.
    ShowData( const char *szDev ) {
        errno_t err;
        err = fopen_s(&OutputDev, szDev, "w" );
    }

    // Destructor closes the file.
    ~ShowData() { fclose( OutputDev ); }

    // Disp function shows a string on the output device.
    void Disp( char *szData ) {
        fputs( szData, OutputDev );
    }
private:
    FILE *OutputDev;
};

// Define a static object of type ShowData. The output device
// selected is "CON" -- the standard output device.
ShowData sd1 = "CON";

// Define another static object of type ShowData. The output
// is directed to a file called "HELLO.DAT"
ShowData sd2 = "hello.dat";

int main() {
    sd1.Disp( "hello to default device\n" );
    sd2.Disp( "hello to file hello.dat\n" );
}
```

Another way to write this code is to declare the `ShowData` objects with block scope, allowing them to be destroyed when they go out of scope:

```
int main() {  
    ShowData sd1, sd2( "hello.dat" );  
  
    sd1.Disp( "hello to default device\n" );  
    sd2.Disp( "hello to file hello.dat\n" );  
}
```

## See also

[main function and command-line arguments](#)

# Lvalues and Rvalues (C++)

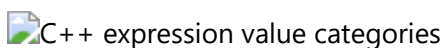
---

Every C++ expression has a type, and belongs to a *value category*. The value categories are the basis for rules that compilers must follow when creating, copying, and moving temporary objects during expression evaluation.

The C++17 standard defines expression value categories as follows:

- A *glvalue* is an expression whose evaluation determines the identity of an object, bit-field, or function.
- A *prvalue* is an expression whose evaluation initializes an object or a bit-field, or computes the value of the operand of an operator, as specified by the context in which it appears.
- An *xvalue* is a glvalue that denotes an object or bit-field whose resources can be reused (usually because it is near the end of its lifetime). Example: Certain kinds of expressions involving rvalue references (8.3.2) yield xvalues, such as a call to a function whose return type is an rvalue reference or a cast to an rvalue reference type.
- An *lvalue* is a glvalue that is not an xvalue.
- An *rvalue* is a prvalue or an xvalue.

The following diagram illustrates the relationships between the categories:



An lvalue has an address that your program can access. Examples of lvalue expressions include variable names, including **const** variables, array elements, function calls that return an lvalue reference, bit-fields, unions, and class members.

A prvalue expression has no address that is accessible by your program. Examples of prvalue expressions include literals, function calls that return a non-reference type, and temporary objects that are created during expression evaluation but accessible only by the compiler.

An xvalue expression has an address that is no longer accessible by your program but can be used to initialize an rvalue reference, which provides access to the expression. Examples include function calls that return an rvalue reference, and the array subscript, member and pointer to member expressions where the array or object is an rvalue reference.

## Example

The following example demonstrates several correct and incorrect usages of lvalues and rvalues:

```
// lvalues_and_rvalues2.cpp
int main()
{
    int i, j, *p;

    // Correct usage: the variable i is an lvalue and the literal 7 is a prvalue.
    i = 7;

    // Incorrect usage: The left operand must be an lvalue (C2106). `j * 4` is a
```

```

prvalue.
    7 = i; // C2106
    j * 4 = 7; // C2106

    // Correct usage: the dereferenced pointer is an lvalue.
    *p = i;

    // Correct usage: the conditional operator returns an lvalue.
    ((i < 3) ? i : j) = 7;

    // Incorrect usage: the constant ci is a non-modifiable lvalue (C3892).
    const int ci = 7;
    ci = 9; // C3892
}

```

[!NOTE] The examples in this topic illustrate correct and incorrect usage when operators are not overloaded. By overloading operators, you can make an expression such as `j * 4` an lvalue.

The terms *lvalue* and *rvalue* are often used when you refer to object references. For more information about references, see [Lvalue Reference Declarator: &](#) and [Rvalue Reference Declarator: &&](#).

## See also

[Basic Concepts](#)

[Lvalue Reference Declarator: &](#)

[Rvalue Reference Declarator: &&](#)



# Temporary Objects

---

In some cases, it is necessary for the compiler to create temporary objects. These temporary objects can be created for the following reasons:

- To initialize a **const** reference with an initializer of a type different from that of the underlying type of the reference being initialized.
- To store the return value of a function that returns a user-defined type. These temporaries are created only if your program does not copy the return value to an object. For example:

```
UDT Func1();    // Declare a function that returns a user-defined
                // type.

...

Func1();        // Call Func1, but discard return value.
                // A temporary object is created to store the return
                // value.
```

Because the return value is not copied to another object, a temporary object is created. A more common case where temporaries are created is during the evaluation of an expression where overloaded operator functions must be called. These overloaded operator functions return a user-defined type that often is not copied to another object.

Consider the expression `ComplexResult = Complex1 + Complex2 + Complex3`. The expression `Complex1 + Complex2` is evaluated, and the result is stored in a temporary object. Next, the expression `temporary + Complex3` is evaluated, and the result is copied to `ComplexResult` (assuming the assignment operator is not overloaded).

- To store the result of a cast to a user-defined type. When an object of a given type is explicitly converted to a user-defined type, that new object is constructed as a temporary object.

Temporary objects have a lifetime that is defined by their point of creation and the point at which they are destroyed. Any expression that creates more than one temporary object eventually destroys them in the reverse order in which they were created. The points at which destruction occurs are shown in the following table.

## Destruction Points for Temporary Objects

Reason Temporary Created	Destruction Point
Result of expression evaluation	All temporaries created as a result of expression evaluation are destroyed at the end of the expression statement (that is, at the semicolon), or at the end of the controlling expressions for <b>for</b> , <b>if</b> , <b>while</b> , <b>do</b> , and <b>switch</b> statements.

**Reason****Temporary****Destruction Point****Created**

---

Initializing

**const**

references

If an initializer is not an l-value of the same type as the reference being initialized, a temporary of the underlying object type is created and initialized with the initialization expression. This temporary object is destroyed immediately after the reference object to which it is bound is destroyed.

# Alignment

---

One of the low-level features of C++ is the ability to specify the precise alignment of objects in memory to take maximum advantage of a specific hardware architecture. By default, the compiler aligns class and struct members on their size value: `bool` and `char` on 1-byte boundaries, `short` on 2-byte boundaries, `int`, `long`, and `float` on 4-byte boundaries, and `long long`, `double`, and `long double` on 8-byte boundaries.

In most scenarios, you never have to be concerned with alignment because the default alignment is already optimal. In some cases, however, you can achieve significant performance improvements, or memory savings, by specifying a custom alignment for your data structures. Before Visual Studio 2015 you could use the Microsoft-specific keywords `__declspec(alignof)` and `__declspec(aligned)` to specify an alignment greater than the default. Starting in Visual Studio 2015 you should use the C++11 standard keywords `alignof` and `alignas` for maximum code portability. The new keywords behave in the same way under the hood as the Microsoft-specific extensions. The documentation for those extensions also applies to the new keywords. For more information, see [\\_\\_alignof Operator](#) and [align](#). The C++ standard doesn't specify packing behavior for alignment on boundaries smaller than the compiler default for the target platform, so you still need to use the Microsoft `#pragma pack` in that case.

Use the [aligned\\_storage class](#) for memory allocation of data structures with custom alignments. The [aligned\\_union class](#) is for specifying alignment for unions with non-trivial constructors or destructors.

## Alignment and memory addresses

Alignment is a property of a memory address, expressed as the numeric address modulo a power of 2. For example, the address 0x0001103F modulo 4 is 3. That address is said to be aligned to  $4n+3$ , where 4 indicates the chosen power of 2. The alignment of an address depends on the chosen power of 2. The same address modulo 8 is 7. An address is said to be aligned to X if its alignment is  $Xn+0$ .

CPUs execute instructions that operate on data stored in memory. The data are identified by their addresses in memory. A single datum also has a size. We call a datum *naturally aligned* if its address is aligned to its size. It's called *misaligned* otherwise. For example, an 8-byte floating-point datum is naturally aligned if the address used to identify it has an 8-byte alignment.

## Compiler handling of data alignment

Compilers attempt to make data allocations in a way that prevents data misalignment.

For simple data types, the compiler assigns addresses that are multiples of the size in bytes of the data type. For example, the compiler assigns addresses to variables of type `long` that are multiples of 4, setting the bottom 2 bits of the address to zero.

The compiler also pads structures in a way that naturally aligns each element of the structure. Consider the structure `struct x_` in the following code example:

```
struct x_  
{  
    char a;    // 1 byte
```

```

int b;      // 4 bytes
short c;    // 2 bytes
char d;     // 1 byte
} bar[3];

```

The compiler pads this structure to enforce alignment naturally.

The following code example shows how the compiler places the padded structure in memory:

```

// Shows the actual memory layout
struct x_
{
    char a;           // 1 byte
    char _pad0[3];    // padding to put 'b' on 4-byte boundary
    int b;            // 4 bytes
    short c;          // 2 bytes
    char d;           // 1 byte
    char _pad1[1];    // padding to make sizeof(x_) multiple of 4
} bar[3];

```

Both declarations return `sizeof(struct x_)` as 12 bytes.

The second declaration includes two padding elements:

1. `char _pad0[3]` to align the `int b` member on a 4-byte boundary.
2. `char _pad1[1]` to align the array elements of the structure `struct x_ bar[3]` on a four-byte boundary.

The padding aligns the elements of `bar[3]` in a way that allows natural access.

The following code example shows the `bar[3]` array layout:

adr	offset	element
-----	-----	
0x0000		char a; // bar[0]
0x0001		char pad0[3];
0x0004		int b;
0x0008		short c;
0x000a		char d;
0x000b		char _pad1[1];
0x000c		char a; // bar[1]
0x000d		char _pad0[3];
0x0010		int b;
0x0014		short c;
0x0016		char d;
0x0017		char _pad1[1];
0x0018		char a; // bar[2]

```
0x0019  char _pad0[3];
0x001c  int b;
0x0020  short c;
0x0022  char d;
0x0023  char _pad1[1];
```

## alignof and alignas

The **alignas** type specifier is a portable, C++ standard way to specify custom alignment of variables and user defined types. The **alignof** operator is likewise a standard, portable way to obtain the alignment of a specified type or variable.

## Example

You can use **alignas** on a class, struct or union, or on individual members. When multiple **alignas** specifiers are encountered, the compiler will choose the strictest one, (the one with the largest value).

```
// alignas_alignof.cpp
// compile with: cl /EHsc alignas_alignof.cpp
#include <iostream>

struct alignas(16) Bar
{
    int i;        // 4 bytes
    int n;        // 4 bytes
    alignas(4) char arr[3];
    short s;      // 2 bytes
};

int main()
{
    std::cout << alignof(Bar) << std::endl; // output: 16
}
```

## See also

[Data structure alignment](#)

# Trivial, standard-layout, POD, and literal types

---

The term *layout* refers to how the members of an object of class, struct or union type are arranged in memory. In some cases, the layout is well-defined by the language specification. But when a class or struct contains certain C++ language features such as virtual base classes, virtual functions, members with different access control, then the compiler is free to choose a layout. That layout may vary depending on what optimizations are being performed and in many cases the object might not even occupy a contiguous area of memory. For example, if a class has virtual functions, all the instances of that class might share a single virtual function table. Such types are very useful, but they also have limitations. Because the layout is undefined they cannot be passed to programs written in other languages, such as C, and because they might be non-contiguous they cannot be reliably copied with fast low-level functions such as `memcpy`, or serialized over a network.

To enable compilers as well as C++ programs and metaprograms to reason about the suitability of any given type for operations that depend on a particular memory layout, C++14 introduced three categories of simple classes and structs: *trivial*, *standard-layout*, and *POD* or Plain Old Data. The Standard Library has the function templates `is_trivial<T>`, `is_standard_layout<T>` and `is_pod<T>` that determine whether a given type belongs to a given category.

## Trivial types

When a class or struct in C++ has compiler-provided or explicitly defaulted special member functions, then it is a trivial type. It occupies a contiguous memory area. It can have members with different access specifiers. In C++, the compiler is free to choose how to order members in this situation. Therefore, you can `memcpy` such objects but you cannot reliably consume them from a C program. A trivial type `T` can be copied into an array of `char` or `unsigned char`, and safely copied back into a `T` variable. Note that because of alignment requirements, there might be padding bytes between type members.

Trivial types have a trivial default constructor, trivial copy constructor, trivial copy assignment operator and trivial destructor. In each case, *trivial* means the constructor/operator/destructor is not user-provided and belongs to a class that has

- no virtual functions or virtual base classes,
- no base classes with a corresponding non-trivial constructor/operator/destructor
- no data members of class type with a corresponding non-trivial constructor/operator/destructor

The following examples show trivial types. In `Trivial2`, the presence of the `Trivial2(int a, int b)` constructor requires that you provide a default constructor. For the type to qualify as trivial, you must explicitly default that constructor.

```
struct Trivial
{
    int i;
private:
    int j;
};
```

```

struct Trivial2
{
    int i;
    Trivial2(int a, int b) : i(a), j(b) {}
    Trivial2() = default;
private:
    int j;    // Different access control
};

```

## Standard layout types

When a class or struct does not contain certain C++ language features such as virtual functions which are not found in the C language, and all members have the same access control, it is a standard-layout type. It is memcpy-able and the layout is sufficiently defined that it can be consumed by C programs. Standard-layout types can have user-defined special member functions. In addition, standard layout types have these characteristics:

- no virtual functions or virtual base classes
- all non-static data members have the same access control
- all non-static members of class type are standard-layout
- any base classes are standard-layout
- has no base classes of the same type as the first non-static data member.
- meets one of these conditions:
  - no non-static data member in the most-derived class and no more than one base class with non-static data members, or
  - has no base classes with non-static data members

The following code shows one example of a standard-layout type:

```

struct SL
{
    // All members have same access:
    int i;
    int j;
    SL(int a, int b) : i(a), j(b) {} // User-defined constructor OK
};

```

The last two requirements can perhaps be better illustrated with code. In the next example, even though `Base` is standard-layout, `Derived` is not standard layout because both it (the most derived class) and `Base` have non-static data members:

```

struct Base
{
    int i;
    int j;
};

// std::is_standard_layout<<Derived> == false!
struct Derived : public Base
{
    int x;
    int y;
};

```

In this example `Derived` is standard-layout because `Base` has no non-static data members:

```

struct Base
{
    void Foo() {}
};

// std::is_standard_layout<<Derived> == true
struct Derived : public Base
{
    int x;
    int y;
};

```

Derived would also be standard-layout if `Base` had the data members and `Derived` had only member functions.

## POD types

When a class or struct is both trivial and standard-layout, it is a POD (Plain Old Data) type. The memory layout of POD types is therefore contiguous and each member has a higher address than the member that was declared before it, so that byte for byte copies and binary I/O can be performed on these types. Scalar types such as `int` are also POD types. POD types that are classes can have only POD types as non-static data members.

## Example

The following example shows the distinctions between trivial, standard-layout, and POD types:

```

#include <type_traits>
#include <iostream>

using namespace std;

```



```

struct B
{
protected:
    virtual void Foo() {}
};

// Neither trivial nor standard-layout
struct A : B
{
    int a;
    int b;
    void Foo() override {} // Virtual function
};

// Trivial but not standard-layout
struct C
{
    int a;
private:
    int b;    // Different access control
};

// Standard-layout but not trivial
struct D
{
    int a;
    int b;
    D() {} //User-defined constructor
};

struct POD
{
    int a;
    int b;
};

int main()
{
    cout << boolalpha;
    cout << "A is trivial is " << is_trivial<A>() << endl; // false
    cout << "A is standard-layout is " << is_standard_layout<A>() << endl; //
false

    cout << "C is trivial is " << is_trivial<C>() << endl; // true
    cout << "C is standard-layout is " << is_standard_layout<C>() << endl; //
false

    cout << "D is trivial is " << is_trivial<D>() << endl; // false
    cout << "D is standard-layout is " << is_standard_layout<D>() << endl; // true

    cout << "POD is trivial is " << is_trivial<POD>() << endl; // true
    cout << "POD is standard-layout is " << is_standard_layout<POD>() << endl; //
true

```

```
return 0;  
}
```

## Literal types

A literal type is one whose layout can be determined at compile time. The following are the literal types:

- void
- scalar types
- references
- Arrays of void, scalar types or references
- A class that has a trivial destructor, and one or more constexpr constructors that are not move or copy constructors. Additionally, all its non-static data members and base classes must be literal types and not volatile.

## See also

[Basic Concepts](#)

# C++ classes as value types

---

C++ classes are by default value types. They can be specified as reference types, which enable polymorphic behavior to support object-oriented programming. Value types are sometimes viewed from the perspective of memory and layout control, whereas reference types are about base classes and virtual functions for polymorphic purposes. By default, value types are copyable, which means there is always a copy constructor and a copy assignment operator. For reference types, you make the class non-copyable (disable the copy constructor and copy assignment operator) and use a virtual destructor, which supports their intended polymorphism. Value types are also about the contents, which, when they are copied, always give you two independent values that can be modified separately. Reference types are about identity - what kind of object is it? For this reason, "reference types" are also referred to as "polymorphic types".

If you really want a reference-like type (base class, virtual functions), you need to explicitly disable copying, as shown in the `MyRefType` class in the following code.

```
// cl /EHsc /nologo /W4

class MyRefType {
private:
    MyRefType & operator=(const MyRefType &);
    MyRefType(const MyRefType &);
public:
    MyRefType () {}
};

int main()
{
    MyRefType Data1, Data2;
    // ...
    Data1 = Data2;
}
```

Compiling the above code will result in the following error:

```
test.cpp(15) : error C2248: 'MyRefType::operator =' : cannot access private member
declared in class 'MyRefType'
    meow.cpp(5) : see declaration of 'MyRefType::operator ='
    meow.cpp(3) : see declaration of 'MyRefType'
```

## Value types and move efficiency

Copy allocation overhead is avoided due to new copy optimizations. For example, when you insert a string in the middle of a vector of strings, there will be no copy re-allocation overhead, only a move- even if it results in a grow of the vector itself. This also applies to other operations, for instance performing an add operation on two very large objects. How do you enable these value operation optimizations? In some C++ compilers,

the compiler will enable this for you implicitly, much like copy constructors can be automatically generated by the compiler. However, in C++, your class will need to "opt-in" to move assignment and constructors by declaring it in your class definition. This is accomplished by using the double ampersand (&&) rvalue reference in the appropriate member function declarations and defining move constructor and move assignment methods. You also need to insert the correct code to "steal the guts" out of the source object.

How do you decide if you need move enabled? If you already know you need copy construction enabled, you probably want move enabled if it can be cheaper than a deep copy. However, if you know you need move support, it doesn't necessarily mean you want copy enabled. This latter case would be called a "move-only type". An example already in the standard library is `unique_ptr`. As a side note, the old `auto_ptr` is deprecated, and was replaced by `unique_ptr` precisely due to the lack of move semantics support in the previous version of C++.

By using move semantics you can return-by-value or insert-in-middle. Move is an optimization of copy. There is need for heap allocation as a workaround. Consider the following pseudocode:

```
#include <set>
#include <vector>
#include <string>
using namespace std;

//...
set<widget> LoadHugeData() {
    set<widget> ret;
    // ... load data from disk and populate ret
    return ret;
}
//...
widgets = LoadHugeData();    // efficient, no deep copy

vector<string> v = IfIHadAMillionStrings();
v.insert( begin(v)+v.size()/2, "scott" );    // efficient, no deep copy-shuffle
v.insert( begin(v)+v.size()/2, "Andrei" );    // (just 1M ptr/len assignments)
//...
HugeMatrix operator+(const HugeMatrix& , const HugeMatrix& );
HugeMatrix operator+(const HugeMatrix& , HugeMatrix&&);
HugeMatrix operator+(HugeMatrix&&, const HugeMatrix& );
HugeMatrix operator+(HugeMatrix&&, HugeMatrix&&);
//...
hm5 = hm1+hm2+hm3+hm4+hm5;    // efficient, no extra copies
```

## Enabling move for appropriate value types

For a value-like class where move can be cheaper than a deep copy, enable move construction and move assignment for efficiency. Consider the following pseudocode:

```
#include <memory>
#include <stdexcept>
using namespace std;
```

```
// ...
class my_class {
    unique_ptr<BigHugeData> data;
public:
    my_class( my_class&& other )    // move construction
        : data( move( other.data ) ) { }
    my_class& operator=( my_class&& other )    // move assignment
    { data = move( other.data ); return *this; }
    // ...
    void method() {    // check (if appropriate)
        if( !data )
            throw std::runtime_error("RUNTIME ERROR: Insufficient resources!");
    }
};
```

If you enable copy construction/assignment, also enable move construction/assignment if it can be cheaper than a deep copy.

Some *non-value* types are move-only, such as when you can't clone a resource, only transfer ownership.

Example: `unique_ptr`.

## See also

[C++ type system](#)

[Welcome back to C++](#)

[C++ Language Reference](#)

[C++ Standard Library](#)

# Type conversions and type safety

---

This document identifies common type conversion problems and describes how you can avoid them in your C++ code.

When you write a C++ program, it's important to ensure that it's type-safe. This means that every variable, function argument, and function return value is storing an acceptable kind of data, and that operations that involve values of different types "make sense" and don't cause data loss, incorrect interpretation of bit patterns, or memory corruption. A program that never explicitly or implicitly converts values from one type to another is type-safe by definition. However, type conversions, even unsafe conversions, are sometimes required. For example, you might have to store the result of a floating point operation in a variable of type **int**, or you might have to pass the value in an unsigned **int** to a function that takes a signed **int**. Both examples illustrate unsafe conversions because they may cause data loss or re-interpretation of a value.

When the compiler detects an unsafe conversion, it issues either an error or a warning. An error stops compilation; a warning allows compilation to continue but indicates a possible error in the code. However, even if your program compiles without warnings, it still may contain code that leads to implicit type conversions that produce incorrect results. Type errors can also be introduced by explicit conversions, or casts, in the code.

## Implicit type conversions

When an expression contains operands of different built-in types, and no explicit casts are present, the compiler uses built-in *standard conversions* to convert one of the operands so that the types match. The compiler tries the conversions in a well-defined sequence until one succeeds. If the selected conversion is a promotion, the compiler does not issue a warning. If the conversion is a narrowing, the compiler issues a warning about possible data loss. Whether actual data loss occurs depends on the actual values involved, but we recommend that you treat this warning as an error. If a user-defined type is involved, then the compiler tries to use the conversions that you have specified in the class definition. If it can't find an acceptable conversion, the compiler issues an error and does not compile the program. For more information about the rules that govern the standard conversions, see [Standard Conversions](#). For more information about user-defined conversions, see [User-Defined Conversions \(C++/CLI\)](#).

### Widening conversions (promotion)

In a widening conversion, a value in a smaller variable is assigned to a larger variable with no loss of data. Because widening conversions are always safe, the compiler performs them silently and does not issue warnings. The following conversions are widening conversions.

From	To
Any signed or unsigned integral type except <b>long long</b> or <b>__int64</b>	<b>double</b>
<b>bool</b> or <b>char</b>	Any other built-in type
<b>short</b> or <b>wchar_t</b>	<b>int</b> , <b>long</b> , <b>long long</b>
<b>int</b> , <b>long</b>	<b>long long</b>

From	To
float	double

## Narrowing conversions (coercion)

The compiler performs narrowing conversions implicitly, but it warns you about potential data loss. Take these warnings very seriously. If you are certain that no data loss will occur because the values in the larger variable will always fit in the smaller variable, then add an explicit cast so that the compiler will no longer issue a warning. If you are not sure that the conversion is safe, add to your code some kind of runtime check to handle possible data loss so that it does not cause your program to produce incorrect results.

Any conversion from a floating point type to an integral type is a narrowing conversion because the fractional portion of the floating point value is discarded and lost.

The following code example shows some implicit narrowing conversions, and the warnings that the compiler issues for them.

```
int i = INT_MAX + 1; //warning C4307: '+' : integral constant overflow
wchar_t wch = 'A'; //OK
char c = wch; // warning C4244: 'initializing': conversion from 'wchar_t'
              // to 'char', possible loss of data
unsigned char c2 = 0xfffe; //warning C4305: 'initializing': truncation from
                          // 'int' to 'unsigned char'
int j = 1.9f; // warning C4244: 'initializing': conversion from 'float' to
              // 'int', possible loss of data
int k = 7.7; // warning C4244: 'initializing': conversion from 'double' to
              // 'int', possible loss of data
```

## Signed - unsigned conversions

A signed integral type and its unsigned counterpart are always the same size, but they differ in how the bit pattern is interpreted for value transformation. The following code example demonstrates what happens when the same bit pattern is interpreted as a signed value and as an unsigned value. The bit pattern stored in both `num` and `num2` never changes from what is shown in the earlier illustration.

```
using namespace std;
unsigned short num = numeric_limits<unsigned short>::max(); // #include <limits>
short num2 = num;
cout << "unsigned val = " << num << " signed val = " << num2 << endl;
// Prints: unsigned val = 65535 signed val = -1

// Go the other way.
num2 = -1;
num = num2;
cout << "unsigned val = " << num << " signed val = " << num2 << endl;
// Prints: unsigned val = 65535 signed val = -1
```

Notice that values are reinterpreted in both directions. If your program produces odd results in which the sign of the value seems inverted from what you expect, look for implicit conversions between signed and unsigned integral types. In the following example, the result of the expression `(0 - 1)` is implicitly converted from **int** to **unsigned int** when it's stored in `num`. This causes the bit pattern to be reinterpreted.

```
unsigned int u3 = 0 - 1;
cout << u3 << endl; // prints 4294967295
```

The compiler does not warn about implicit conversions between signed and unsigned integral types. Therefore, we recommend that you avoid signed-to-unsigned conversions altogether. If you can't avoid them, then add to your code a runtime check to detect whether the value being converted is greater than or equal to zero and less than or equal to the maximum value of the signed type. Values in this range will transfer from signed to unsigned or from unsigned to signed without being reinterpreted.

## Pointer conversions

In many expressions, a C-style array is implicitly converted to a pointer to the first element in the array, and constant conversions can happen silently. Although this is convenient, it's also potentially error-prone. For example, the following badly designed code example seems nonsensical, and yet it will compile and produces a result of 'p'. First, the "Help" string constant literal is converted to a `char*` that points to the first element of the array; that pointer is then incremented by three elements so that it now points to the last element 'p'.

```
char* s = "Help" + 3;
```

## Explicit conversions (casts)

By using a cast operation, you can instruct the compiler to convert a value of one type to another type. The compiler will raise an error in some cases if the two types are completely unrelated, but in other cases it will not raise an error even if the operation is not type-safe. Use casts sparingly because any conversion from one type to another is a potential source of program error. However, casts are sometimes required, and not all casts are equally dangerous. One effective use of a cast is when your code performs a narrowing conversion and you know that the conversion is not causing your program to produce incorrect results. In effect, this tells the compiler that you know what you are doing and to stop bothering you with warnings about it. Another use is to cast from a pointer-to-derived class to a pointer-to-base class. Another use is to cast away the **const**-ness of a variable to pass it to a function that requires a non-**const** argument. Most of these cast operations involve some risk.

In C-style programming, the same C-style cast operator is used for all kinds of casts.

```
(int) x; // old-style cast, old-style syntax
int(x); // old-style cast, functional syntax
```

The C-style cast operator is identical to the call operator `()` and is therefore inconspicuous in code and easy to overlook. Both are bad because they're difficult to recognize at a glance or search for, and they're disparate



enough to invoke any combination of **static**, **const**, and **reinterpret\_cast**. Figuring out what an old-style cast actually does can be difficult and error-prone. For all these reasons, when a cast is required, we recommend that you use one of the following C++ cast operators, which in some cases are significantly more type-safe, and which express much more explicitly the programming intent:

- **static\_cast**, for casts that are checked at compile time only. **static\_cast** returns an error if the compiler detects that you are trying to cast between types that are completely incompatible. You can also use it to cast between pointer-to-base and pointer-to-derived, but the compiler can't always tell whether such conversions will be safe at runtime.

```
double d = 1.58947;
int i = d; // warning C4244 possible loss of data
int j = static_cast<int>(d); // No warning.
string s = static_cast<string>(d); // Error C2440:cannot convert from
                                   // double to std:string

// No error but not necessarily safe.
Base* b = new Base();
Derived* d2 = static_cast<Derived*>(b);
```

For more information, see [static\\_cast](#).

- **dynamic\_cast**, for safe, runtime-checked casts of pointer-to-base to pointer-to-derived. A **dynamic\_cast** is safer than a **static\_cast** for downcasts, but the runtime check incurs some overhead.

```
Base* b = new Base();

// Run-time check to determine whether b is actually a Derived*
Derived* d3 = dynamic_cast<Derived*>(b);

// If b was originally a Derived*, then d3 is a valid pointer.
if(d3)
{
    // Safe to call Derived method.
    cout << d3->DoSomethingMore() << endl;
}
else
{
    // Run-time check failed.
    cout << "d3 is null" << endl;
}

//Output: d3 is null;
```

For more information, see [dynamic\\_cast](#).

- **const\_cast**, for casting away the **const**-ness of a variable, or converting a non-**const** variable to be **const**. Casting away **const**-ness by using this operator is just as error-prone as is using a C-style cast,

except that with **const-cast** you are less likely to perform the cast accidentally. Sometimes you have to cast away the **const**-ness of a variable, for example, to pass a **const** variable to a function that takes a non-**const** parameter. The following example shows how to do this.

```
void Func(double& d) { ... }
void ConstCast()
{
    const double pi = 3.14;
    Func(const_cast<double&>(pi)); //No error.
}
```

For more information, see [const\\_cast](#).

- **reinterpret\_cast**, for casts between unrelated types such as **pointer** to **int**.

[!NOTE] This cast operator is not used as often as the others, and it's not guaranteed to be portable to other compilers.

The following example illustrates how **reinterpret\_cast** differs from **static\_cast**.

```
const char* str = "hello";
int i = static_cast<int>(str); //error C2440: 'static_cast' : cannot
                               // convert from 'const char *' to 'int'
int j = (int)str; // C-style cast. Did the programmer really intend
                 // to do this?
int k = reinterpret_cast<int>(str); // Programming intent is clear.
                                     // However, it is not 64-bit safe.
```

For more information, see [reinterpret\\_cast Operator](#).

## See also

[C++ type system](#)

[Welcome back to C++](#)

[C++ Language Reference](#)

[C++ Standard Library](#)

# Standard conversions

---

The C++ language defines conversions between its fundamental types. It also defines conversions for pointer, reference, and pointer-to-member derived types. These conversions are called *standard conversions*.

This section discusses the following standard conversions:

- Integral promotions
- Integral conversions
- Floating conversions
- Floating and integral conversions
- Arithmetic conversions
- Pointer conversions
- Reference conversions
- Pointer-to-member conversions

[!NOTE] User-defined types can specify their own conversions. Conversion of user-defined types is covered in [Constructors](#) and [Conversions](#).

The following code causes conversions (in this example, integral promotions):

```
long  long_num1, long_num2;
int   int_num;

// int_num promoted to type long prior to assignment.
long_num1 = int_num;

// int_num promoted to type long prior to multiplication.
long_num2 = int_num * long_num2;
```

The result of a conversion is an l-value only if it produces a reference type. For example, a user-defined conversion declared as `operator int&()` returns a reference and is an l-value. However, a conversion declared as `operator int()` returns an object and isn't an l-value.

## Integral promotions

Objects of an integral type can be converted to another wider integral type, that is, a type that can represent a larger set of values. This widening type of conversion is called *integral promotion*. With integral promotion, you can use the following types in an expression wherever another integral type can be used:

- Objects, literals, and constants of type **char** and **short int**

- Enumeration types
- **int** bit fields
- Enumerators

C++ promotions are "value-preserving," as the value after the promotion is guaranteed to be the same as the value before the promotion. In value-preserving promotions, objects of shorter integral types (such as bit fields or objects of type **char**) are promoted to type **int** if **int** can represent the full range of the original type. If **int** can't represent the full range of values, then the object is promoted to type **unsigned int**. Although this strategy is the same as the one used by Standard C, value-preserving conversions don't preserve the "signedness" of the object.

Value-preserving promotions and promotions that preserve signedness normally produce the same results. However, they can produce different results if the promoted object appears as:

- An operand of `/`, `%`, `/=`, `%=`, `<`, `<=`, `>`, or `>=`

These operators rely on sign for determining the result. Value-preserving and sign-preserving promotions produce different results when applied to these operands.

- The left operand of `>>` or `>>=`

These operators treat signed and unsigned quantities differently in a shift operation. For signed quantities, a right shift operation propagates the sign bit into the vacated bit positions, while the vacated bit positions are zero-filled in unsigned quantities.

- An argument to an overloaded function, or the operand of an overloaded operator, that depends on the signedness of the operand type for argument matching. For more information about defining overloaded operators, see [Overloaded operators](#).

## Integral conversions

*Integral conversions* are conversions between integral types. The integral types are **char**, **short** (or **short int**), **int**, **long**, and **long long**. These types may be qualified with **signed** or **unsigned**, and **unsigned** can be used as shorthand for **unsigned int**.

### Signed to unsigned

Objects of signed integral types can be converted to corresponding unsigned types. When these conversions occur, the actual bit pattern doesn't change. However, the interpretation of the data changes. Consider this code:

```
#include <iostream>

using namespace std;
int main()
{
    short i = -3;
    unsigned short u;
```

```
    cout << (u = i) << "\n";  
}  
// Output: 65533
```

In the preceding example, a **signed short**, `i`, is defined and initialized to a negative number. The expression `(u = i)` causes `i` to be converted to an **unsigned short** before the assignment to `u`.

## Unsigned to signed

Objects of unsigned integral types can be converted to corresponding signed types. However, if the unsigned value is outside the representable range of the signed type, the result won't have the correct value, as demonstrated in the following example:

```
#include <iostream>  
  
using namespace std;  
int main()  
{  
    short i;  
    unsigned short u = 65533;  
  
    cout << (i = u) << "\n";  
}  
//Output: -3
```

In the preceding example, `u` is an **unsigned short** integral object that must be converted to a signed quantity to evaluate the expression `(i = u)`. Because its value can't be properly represented in a **signed short**, the data is misinterpreted as shown.

## Floating point conversions

An object of a floating type can be safely converted to a more precise floating type — that is, the conversion causes no loss of significance. For example, conversions from **float** to **double** or from **double** to **long double** are safe, and the value is unchanged.

An object of a floating type can also be converted to a less precise type, if it's in a range representable by that type. (See [Floating Limits](#) for the ranges of floating types.) If the original value isn't representable precisely, it can be converted to either the next higher or the next lower representable value. The result is undefined if no such value exists. Consider the following example:

```
cout << (float)1E300 << endl;
```

The maximum value representable by type **float** is 3.402823466E38 — a much smaller number than 1E300. Therefore, the number is converted to infinity, and the result is "inf".

## Conversions between integral and floating point types

Certain expressions can cause objects of floating type to be converted to integral types, or vice versa. When an object of integral type is converted to a floating type, and the original value isn't representable exactly, the result is either the next higher or the next lower representable value.

When an object of floating type is converted to an integral type, the fractional part is *truncated*, or rounded toward zero. A number like 1.3 is converted to 1, and -1.3 is converted to -1. If the truncated value is higher than the highest representable value, or lower than the lowest representable value, the result is undefined.

## Arithmetic conversions

Many binary operators (discussed in [Expressions with binary operators](#)) cause conversions of operands, and yield results the same way. The conversions these operators cause are called *usual arithmetic conversions*. Arithmetic conversions of operands that have different native types are done as shown in the following table. Typedef types behave according to their underlying native types.

### Conditions for type conversion

Conditions Met	Conversion
Either operand is of type <b>long double</b> .	Other operand is converted to type <b>long double</b> .
Preceding condition not met and either operand is of type <b>double</b> .	Other operand is converted to type <b>double</b> .
Preceding conditions not met and either operand is of type <b>float</b> .	Other operand is converted to type <b>float</b> .
Preceding conditions not met (none of the operands are of floating types).	<p>Operands get integral promotions as follows:</p> <ul style="list-style-type: none"><li>- If either operand is of type <b>unsigned long</b>, the other operand is converted to type <b>unsigned long</b>.</li><li>- If preceding condition not met, and if either operand is of type <b>long</b> and the other of type <b>unsigned int</b>, both operands are converted to type <b>unsigned long</b>.</li><li>- If the preceding two conditions aren't met, and if either operand is of type <b>long</b>, the other operand is converted to type <b>long</b>.</li><li>- If the preceding three conditions aren't met, and if either operand is of type <b>unsigned int</b>, the other operand is converted to type <b>unsigned int</b>.</li><li>- If none of the preceding conditions are met, both operands are converted to type <b>int</b>.</li></ul>

The following code illustrates the conversion rules described in the table:

```
double dVal;  
float fVal;
```

```

int iVal;
unsigned long ulVal;

int main() {
    // iVal converted to unsigned long
    // result of multiplication converted to double
    dVal = iVal * ulVal;

    // ulVal converted to float
    // result of addition converted to double
    dVal = ulVal + fVal;
}

```

The first statement in the preceding example shows multiplication of two integral types, `iVal` and `ulVal`. The condition met is that neither operand is of floating type, and one operand is of type **unsigned int**. So, the other operand, `iVal`, is converted to type **unsigned int**. The result is then assigned to `dVal`. The condition met here is that one operand is of type **double**, so the **unsigned int** result of the multiplication is converted to type **double**.

The second statement in the preceding example shows addition of a **float** and an integral type: `fVal` and `ulVal`. The `ulVal` variable is converted to type **float** (third condition in the table). The result of the addition is converted to type **double** (second condition in the table) and assigned to `dVal`.

## Pointer conversions


Pointers can be converted during assignment, initialization, comparison, and other expressions.

### Pointer to classes

There are two cases in which a pointer to a class can be converted to a pointer to a base class.

The first case is when the specified base class is accessible and the conversion is unambiguous. For more information about ambiguous base-class references, see [Multiple base classes](#).

Whether a base class is accessible depends on the kind of inheritance used in derivation. Consider the inheritance illustrated in the following figure.

 Inheritance graph showing base-class accessibility  
Inheritance Graph for Illustration of Base-Class Accessibility

The following table shows the base-class accessibility for the situation illustrated in the figure.

Type of Function	Derivation	Conversion from
		B* to A* Legal?
External (not class-scoped) function	Private	No
	Protected	No
	Public	Yes

Type of Function	Derivation	Conversion from
		B* to A* Legal?
B member function (in B scope)	Private	Yes
	Protected	Yes
	Public	Yes
C member function (in C scope)	Private	No
	Protected	Yes
	Public	Yes

The second case in which a pointer to a class can be converted to a pointer to a base class is when you use an explicit type conversion. For more information about explicit type conversions, see [Explicit type conversion operator](#).

The result of such a conversion is a pointer to the *subobject*, the portion of the object that is completely described by the base class.

The following code defines two classes, **A** and **B**, where **B** is derived from **A**. (For more information on inheritance, see [Derived Classes](#).) It then defines **bObject**, an object of type **B**, and two pointers (**pA** and **pB**) that point to the object.

```
// C2039 expected
class A
{
public:
    int AComponent;
    int AMemberFunc();
};

class B : public A
{
public:
    int BComponent;
    int BMemberFunc();
};

int main()
{
    B bObject;
    A *pA = &bObject;
    B *pB = &bObject;

    pA->AMemberFunc(); // OK in class A
    pB->AMemberFunc(); // OK: inherited from class A
    pA->BMemberFunc(); // Error: not in class A
}
```



The pointer `pA` is of type `A *`, which can be interpreted as meaning "pointer to an object of type `A`." Members of `bObject` (such as `BComponent` and `BMemberFunc`) are unique to type `B` and are therefore inaccessible through `pA`. The `pA` pointer allows access only to those characteristics (member functions and data) of the object that are defined in class `A`.

## Pointer to function

A pointer to a function can be converted to type `void *`, if type `void *` is large enough to hold that pointer.

## Pointer to void

Pointers to type `void` can be converted to pointers to any other type, but only with an explicit type cast (unlike in C). A pointer to any type can be converted implicitly to a pointer to type `void`. A pointer to an incomplete object of a type can be converted to a pointer to `void` (implicitly) and back (explicitly). The result of such a conversion is equal to the value of the original pointer. An object is considered incomplete if it's declared, but there's insufficient information available to determine its size or base class.

A pointer to any object that is not `const` or `volatile` can be implicitly converted to a pointer of type `void *`.

## const and volatile pointers

C++ doesn't supply a standard conversion from a `const` or `volatile` type to a type that's not `const` or `volatile`. However, any sort of conversion can be specified using explicit type casts (including conversions that are unsafe).

[!NOTE] C++ pointers to members, except pointers to static members, are different from normal pointers and don't have the same standard conversions. Pointers to static members are normal pointers and have the same conversions as normal pointers.

## null pointer conversions

An integral constant expression that evaluates to zero, or such an expression cast to a pointer type, is converted to a pointer called the *null pointer*. This pointer always compares unequal to a pointer to any valid object or function. An exception is pointers to base objects, which can have the same offset and still point to different objects.

In C++11, the `nullptr` type should be preferred to the C-style null pointer.

## Pointer expression conversions

Any expression with an array type can be converted to a pointer of the same type. The result of the conversion is a pointer to the first array element. The following example demonstrates such a conversion:

```
char szPath[_MAX_PATH]; // Array of type char.
char *pszPath = szPath; // Equals &szPath[0].
```

An expression that results in a function returning a particular type is converted to a pointer to a function returning that type, except when:

- The expression is used as an operand to the address-of operator (&).
- The expression is used as an operand to the function-call operator.

## Reference conversions

A reference to a class can be converted to a reference to a base class in these cases:

- The specified base class is accessible.
- The conversion is unambiguous. (For more information about ambiguous base-class references, see [Multiple base classes](#).)

The result of the conversion is a pointer to the subobject that represents the base class.

## Pointer to member

Pointers to class members can be converted during assignment, initialization, comparison, and other expressions. This section describes the following pointer-to-member conversions:

### Pointer to base class member

A pointer to a member of a base class can be converted to a pointer to a member of a class derived from it, when the following conditions are met:

- The inverse conversion, from pointer to derived class to base-class pointer, is accessible.
- The derived class does not inherit virtually from the base class.

When the left operand is a pointer to member, the right operand must be of pointer-to-member type or be a constant expression that evaluates to 0. This assignment is valid only in the following cases:

- The right operand is a pointer to a member of the same class as the left operand.
- The left operand is a pointer to a member of a class derived publicly and unambiguously from the class of the right operand.

### null pointer to member conversions

An integral constant expression that evaluates to zero is converted to a null pointer. This pointer always compares unequal to a pointer to any valid object or function. An exception is pointers to based objects, which can have the same offset and still point to different objects.

The following code illustrates the definition of a pointer to member `i` in class `A`. The pointer, `pai`, is initialized to 0, which is the null pointer.

```
class A
{
public:
    int i;
};
```

```
int A::*pai = 0;
```

```
int main()  
{  
}
```

## See also

[C++ language reference](#)

# Built-in types (C++)

---

Built-in types (also called *fundamental types*) are specified by the C++ language standard and are built into the compiler. Built-in types are not defined in any header file. Built-in types are divided into three categories: integral, floating point, and void. Integral types are capable of handling whole numbers. Floating point types are capable of specifying values that may have fractional parts.

The `void` type describes an empty set of values. No variable of type `void` can be specified — it is used primarily to declare functions that return no values or to declare generic pointers to untyped or arbitrarily typed data. Any expression can be explicitly converted or cast to type `void`. However, such expressions are restricted to the following uses:

- An expression statement. (For more information, see [Expressions](#).)
- The left operand of the comma operator. (For more information, see [Comma Operator](#).)
- The second or third operand of the conditional operator (`? :`). (For more information, see [Expressions with the Conditional Operator](#).)

The following table explains the restrictions on type sizes in relation to each other. These restrictions are mandated by the C++ standard and are independent of the Microsoft implementation. The absolute size of certain built-in types is not specified in the standard.

## Built-in type size restrictions

Category	Type	Contents
Integral		Type <b>char</b> is an integral type that usually contains members of the basic execution character set — By default, this is ASCII in Microsoft C++.
	<b>char</b>	The C++ compiler treats variables of type <b>char</b> , <b>signed char</b> , and <b>unsigned char</b> as having different types. Variables of type <b>char</b> are promoted to <b>int</b> as if they are type <b>signed char</b> by default, unless the <code>/J</code> compilation option is used. In this case, they are treated as type <b>unsigned char</b> and are promoted to <b>int</b> without sign extension.
	<b>bool</b>	Type <b>bool</b> is an integral type that can have one of the two values <b>true</b> or <b>false</b> . Its size is unspecified.
	<b>short</b>	<p>Type <b>short int</b> (or simply <b>short</b>) is an integral type that is larger than or equal to the size of type <b>char</b>, and shorter than or equal to the size of type <b>int</b>.</p> <p>Objects of type <b>short</b> can be declared as <b>signed short</b> or <b>unsigned short</b>. <b>Signed short</b> is a synonym for <b>short</b>.</p>

Category	Type	Contents
	<b>int</b>	<p>Type <b>int</b> is an integral type that is larger than or equal to the size of type <b>short int</b>, and shorter than or equal to the size of type <b>long</b>.</p> <p>Objects of type <b>int</b> can be declared as <b>signed int</b> or <b>unsigned int</b>. <b>Signed int</b> is a synonym for <b>int</b>.</p>
	<b>__int8,</b> <b>__int16,</b> <b>__int32,</b> <b>__int64</b>	Sized integer <b>__int</b> <i>n</i> , where <i>n</i> is the size, in bits, of the integer variable. <b>__int8</b> , <b>__int16</b> , <b>__int32</b> and <b>__int64</b> are Microsoft-specific keywords. Not all types are available on all architectures. ( <b>__int128</b> is not supported.)
	<b>long</b>	<p>Type <b>long</b> (or <b>long int</b>) is an integral type that is larger than or equal to the size of type <b>int</b>. (On Windows <b>long</b> is the same size as <b>int</b>.)</p> <p>Objects of type <b>long</b> can be declared as <b>signed long</b> or <b>unsigned long</b>. <b>Signed long</b> is a synonym for <b>long</b>.</p>
	<b>long long</b>	<p>Larger than an unsigned <b>long</b>.</p> <p>Objects of type <b>long long</b> can be declared as <b>signed long long</b> or <b>unsigned long long</b>. <b>signed long long</b> is a synonym for <b>long long</b>.</p>
	<b>wchar_t,</b> <b>__wchar_t</b>	<p>A variable of type <b>wchar_t</b> designates a wide-character or multibyte character type. By default, <b>wchar_t</b> is a native type, but you can use <a href="#">/Zc:wchar_t-</a> to make <b>wchar_t</b> a typedef for <b>unsigned short</b>. The <b>__wchar_t</b> type is a Microsoft-specific synonym for the native <b>wchar_t</b> type.</p> <p>Use the L prefix before a character or string literal to designate the wide-character type.</p>
Floating point	<b>float</b>	Type <b>float</b> is the smallest floating point type.
	<b>double</b>	<p>Type <b>double</b> is a floating point type that is larger than or equal to type <b>float</b>, but shorter than or equal to the size of type <b>long double</b>.</p> <p>Microsoft-specific: The representation of <b>long double</b> and <b>double</b> is identical. However, <b>long double</b> and <b>double</b> are separate types.</p>
	<b>long double</b>	Type <b>long double</b> is a floating point type that is larger than or equal to type <b>double</b> .

## Microsoft Specific

The following table lists the amount of storage required for built-in types in Microsoft C++. In particular, note that **long** is 4 bytes even on 64-bit operating systems.

## Sizes of built-in types

Type	Size
<b>bool, char, unsigned char, signed char, __int8</b>	1 byte
<b>__int16, short, unsigned short, wchar_t, __wchar_t</b>	2 bytes
<b>float, __int32, int, unsigned int, long, unsigned long</b>	4 bytes
<b>double, __int64, long double, long long</b>	8 bytes

#### END Microsoft Specific

See [Data Type Ranges](#) for a summary of the range of values of each type.

For more information about type conversion, see [Standard Conversions](#).

## See also

[Data Type Ranges](#)

# Data Type Ranges

The Microsoft C++ 32-bit and 64-bit compilers recognize the types in the table later in this article.

- `int` (`unsigned int`)
- `__int8` (`unsigned __int8`)
- `__int16` (`unsigned __int16`)
- `__int32` (`unsigned __int32`)
- `__int64` (`unsigned __int64`)
- `short` (`unsigned short`)
- `long` (`unsigned long`)
- `long long` (`unsigned long long`)

If its name begins with two underscores (`__`), a data type is non-standard.

The ranges that are specified in the following table are inclusive-inclusive.

Type Name	Bytes	Other Names	Range of Values
<code>int</code>	4	<code>signed</code>	-2,147,483,648 to 2,147,483,647
<code>unsigned int</code>	4	<code>unsigned</code>	0 to 4,294,967,295
<code>__int8</code>	1	<code>char</code>	-128 to 127
<code>unsigned __int8</code>	1	<code>unsigned char</code>	0 to 255
<code>__int16</code>	2	<code>short</code> , <code>short int</code> , <code>signed short int</code>	-32,768 to 32,767
<code>unsigned __int16</code>	2	<code>unsigned short</code> , <code>unsigned short int</code>	0 to 65,535
<code>__int32</code>	4	<code>signed</code> , <code>signed int</code> , <code>int</code>	-2,147,483,648 to 2,147,483,647
<code>unsigned __int32</code>	4	<code>unsigned</code> , <code>unsigned int</code>	0 to 4,294,967,295
<code>__int64</code>	8	<code>long long</code> , <code>signed long long</code>	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
<code>unsigned __int64</code>	8	<code>unsigned long long</code>	0 to 18,446,744,073,709,551,615
<code>bool</code>	1	none	<code>false</code> or <code>true</code>

Type Name	Bytes	Other Names	Range of Values
<b>char</b>	1	none	-128 to 127 by default 0 to 255 when compiled by using <a href="#">/J</a>
<b>signed char</b>	1	none	-128 to 127
<b>unsigned char</b>	1	none	0 to 255
<b>short</b>	2	<b>short int, signed short int</b>	-32,768 to 32,767
<b>unsigned short</b>	2	<b>unsigned short int</b>	0 to 65,535
<b>long</b>	4	<b>long int, signed long int</b>	-2,147,483,648 to 2,147,483,647
<b>unsigned long</b>	4	<b>unsigned long int</b>	0 to 4,294,967,295
<b>long long</b>	8	none (but equivalent to <b>__int64</b> )	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
<b>unsigned long long</b>	8	none (but equivalent to <b>unsigned __int64</b> )	0 to 18,446,744,073,709,551,615
<b>enum</b>	varies	none	
<b>float</b>	4	none	3.4E +/- 38 (7 digits)
<b>double</b>	8	none	1.7E +/- 308 (15 digits)
<b>long double</b>	same as <b>double</b>	none	Same as <b>double</b>
<b>wchar_t</b>	2	<b>__wchar_t</b>	0 to 65,535

Depending on how it's used, a variable of **\_\_wchar\_t** designates either a wide-character type or multibyte-character type. Use the **L** prefix before a character or string constant to designate the wide-character-type constant.

**signed** and **unsigned** are modifiers that you can use with any integral type except **bool**. Note that **char**, **signed char**, and **unsigned char** are three distinct types for the purposes of mechanisms like overloading and templates.

The **int** and **unsigned int** types have a size of four bytes. However, portable code should not depend on the size of **int** because the language standard allows this to be implementation-specific.

C/C++ in Visual Studio also supports sized integer types. For more information, see [\\_\\_int8](#), [\\_\\_int16](#), [\\_\\_int32](#), [\\_\\_int64](#) and [Integer Limits](#).

For more information about the restrictions of the sizes of each type, see [Built-in types](#).



The range of enumerated types varies depending on the language context and specified compiler flags. For more information, see [C Enumeration Declarations](#) and [Enumerations](#).

## See also

[Keywords](#)

[Built-in types](#)

# nullptr

---

Designates a null pointer constant of type `std::nullptr_t`, which is convertible to any raw pointer type. Although you can use the keyword **nullptr** without including any headers, if your code uses the type `std::nullptr_t`, then you must define it by including the header `<cstdlib>`.

[!NOTE] The **nullptr** keyword is also defined in C++/CLI for managed code applications and is not interchangeable with the ISO Standard C++ keyword. If your code might be compiled by using the `/clr` compiler option, which targets managed code, then use `__nullptr` in any line of code where you must guarantee that the compiler uses the native C++ interpretation. For more information, see [nullptr](#).

## Remarks

Avoid using `NULL` or zero (`0`) as a null pointer constant; **nullptr** is less vulnerable to misuse and works better in most situations. For example, given `func(std::pair<const char *, double>)`, then calling `func(std::make_pair(NULL, 3.14))` causes a compiler error. The macro `NULL` expands to `0`, so that the call `std::make_pair(0, 3.14)` returns `std::pair<int, double>`, which is not convertible to `func()`'s `std::pair<const char *, double>` parameter type. Calling `func(std::make_pair(nullptr, 3.14))` successfully compiles because `std::make_pair(nullptr, 3.14)` returns `std::pair<std::nullptr_t, double>`, which is convertible to `std::pair<const char *, double>`.

## See also

[Keywords](#)

[nullptr\(C++/CLI\)](#)

# void (C++)

---

When used as a function return type, the **void** keyword specifies that the function does not return a value. When used for a function's parameter list, **void** specifies that the function takes no parameters. When used in the declaration of a pointer, **void** specifies that the pointer is "universal."

If a pointer's type is **void\***, the pointer can point to any variable that is not declared with the **const** or **volatile** keyword. A **void\*** pointer cannot be dereferenced unless it is cast to another type. A **void\*** pointer can be converted into any other type of data pointer.

A **void** pointer can point to a function, but not to a class member in C++.

You cannot declare a variable of type **void**.

## Example

```
// void.cpp
void vobject;    // C2182
void *pv;        // okay
int *pint; int i;
int main() {
    pv = &i;
    // Cast optional in C required in C++
    pint = (int *)pv;
}
```

## See also

[Keywords](#)

[Built-in types](#)

# bool (C++)

---

This keyword is a built-in type. A variable of this type can have values [true](#) and [false](#). Conditional expressions have the type **bool** and so have values of type **bool**. For example, `i!=0` now has TRUE or FALSE depending on the value of `i`.

**Visual Studio 2017 version 15.3 and later** (available with [/std:c++17](#)): The operand of a postfix or prefix increment or decrement operator may not be of type **bool**. In other words, given a variable `b` of type **bool**, these expressions are no longer allowed:

```
b++;  
++b;  
b--;  
--b;
```

The values TRUE and FALSE have the following relationship:

```
!false == true  
!true == false
```

In the following statement:

```
if (condexpr1) statement1;
```

If `condexpr1` is TRUE, `statement1` is always executed; if `condexpr1` is FALSE, `statement1` is never executed.

When a postfix or prefix `++` operator is applied to a variable of type **bool**, the variable is set to TRUE. **Visual Studio 2017 version 15.3 and later**: operator`++` for **bool** was removed from the language and is no longer supported.

The postfix or prefix `--` operator cannot be applied to a variable of this type.

The **bool** type participates in integral promotions. An r-value of type **bool** can be converted to an r-value of type **int**, with FALSE becoming zero and TRUE becoming one. As a distinct type, **bool** participates in overload resolution.

## See also

[Keywords](#)

[Built-in types](#)

# false (C++)

---

The keyword is one of the two values for a variable of type `bool` or a conditional expression (a conditional expression is now a **true** Boolean expression). For example, if `i` is a variable of type **bool**, the `i = false;` statement assigns **false** to `i`.

## Example

```
// bool_false.cpp
#include <stdio.h>

int main()
{
    bool bb = true;
    printf_s("%d\n", bb);
    bb = false;
    printf_s("%d\n", bb);
}
```

```
1
0
```

## See also

[Keywords](#)

# true (C++)

---

## Syntax

```
bool-identifier = true ;  
bool-expression logical-operator true ;
```

## Remarks

This keyword is one of the two values for a variable of type `bool` or a conditional expression (a conditional expression is now a true boolean expression). If `i` is of type `bool`, then the statement `i = true;` assigns **true** to `i`.

## Example

```
// bool_true.cpp  
#include <stdio.h>  
int main()  
{  
    bool bb = true;  
    printf_s("%d\n", bb);  
    bb = false;  
    printf_s("%d\n", bb);  
}
```

```
1  
0
```

## See also

[Keywords](#)

# char, wchar\_t, char16\_t, char32\_t

---

The types **char**, **wchar\_t**, **char16\_t** and **char32\_t** are built-in types that represent alphanumeric characters as well as non-alphanumeric glyphs and non-printing characters.

## Syntax

```
char      ch1{ 'a' }; // or { u8'a' }
wchar_t   ch2{ L'a' };
char16_t  ch3{ u'a' };
char32_t  ch4{ U'a' };
```

## Remarks

The **char** type was the original character type in C and C++. The type **unsigned char** is often used to represent a *byte*, which is not a built-in type in C++. The **char** type can be used to store characters from the ASCII character set or any of the ISO-8859 character sets, and individual bytes of multi-byte characters such as Shift-JIS or the UTF-8 encoding of the Unicode character set. Strings of **char** type are referred to as *narrow* strings, even when used to encode multi-byte characters. In the Microsoft compiler, **char** is an 8-bit type.

The **wchar\_t** type is an implementation-defined wide character type. In the Microsoft compiler, it represents a 16-bit wide character used to store Unicode encoded as UTF-16LE, the native character type on Windows operating systems. The wide character versions of the Universal C Runtime (UCRT) library functions use **wchar\_t** and its pointer and array types as parameters and return values, as do the wide character versions of the native Windows API.

The **char16\_t** and **char32\_t** types represent 16-bit and 32-bit wide characters, respectively. Unicode encoded as UTF-16 can be stored in the **char16\_t** type, and Unicode encoded as UTF-32 can be stored in the **char32\_t** type. Strings of these types and **wchar\_t** are all referred to as *wide* strings, though the term often refers specifically to strings of **wchar\_t** type.

In the C++ standard library, the **basic\_string** type is specialized for both narrow and wide strings. Use **std::string** when the characters are of type **char**, **std::u16string** when the characters are of type **char16\_t**, **std::u32string** when the characters are of type **char32\_t**, and **std::wstring** when the characters are of type **wchar\_t**. Other types that represent text, including **std::stringstream** and **std::cout** have specializations for narrow and wide strings.

# \_\_int8, \_\_int16, \_\_int32, \_\_int64

---

## Microsoft Specific

Microsoft C/C++ features support for sized integer types. You can declare 8-, 16-, 32-, or 64-bit integer variables by using the **\_\_int $n$**  type specifier, where  $n$  is 8, 16, 32, or 64.

The following example declares one variable for each of these types of sized integers:

```
__int8 nSmall;      // Declares 8-bit integer
__int16 nMedium;    // Declares 16-bit integer
__int32 nLarge;     // Declares 32-bit integer
__int64 nHuge;      // Declares 64-bit integer
```

The types **\_\_int8**, **\_\_int16**, and **\_\_int32** are synonyms for the ANSI types that have the same size, and are useful for writing portable code that behaves identically across multiple platforms. The **\_\_int8** data type is synonymous with type **char**, **\_\_int16** is synonymous with type **short**, and **\_\_int32** is synonymous with type **int**. The **\_\_int64** type is synonymous with type **long long**.

For compatibility with previous versions, **\_int8**, **\_int16**, **\_int32**, and **\_int64** are synonyms for **\_\_int8**, **\_\_int16**, **\_\_int32**, and **\_\_int64** unless compiler option [/Za \(Disable language extensions\)](#) is specified.

## Example

The following sample shows that an **\_\_intxx** parameter will be promoted to **int**:

```
// sized_int_types.cpp

#include <stdio.h>

void func(int i) {
    printf_s("%s\n", __FUNCTION__);
}

int main()
{
    __int8 i8 = 100;
    func(i8);    // no void func(__int8 i8) function
                // __int8 will be promoted to int
}
```

func

## END Microsoft Specific



# See also

- [Keywords](#)
- [Built-in types](#)
- [Data Type Ranges](#)

# \_\_m64

---

## Microsoft Specific

The **\_\_m64** data type is for use with the MMX and 3DNow! intrinsics, and is defined in <xmmmintrin.h>.

```
// data_types__m64.cpp
#include <xmmmintrin.h>
int main()
{
    __m64 x;
}
```

## Remarks

You should not access the **\_\_m64** fields directly. You can, however, see these types in the debugger. A variable of type **\_\_m64** maps to the MM[0-7] registers.

Variables of type **\_\_m64** are automatically aligned on 8-byte boundaries.

The **\_\_m64** data type is not supported on x64 processors. Applications that use **\_\_m64** as part of MMX intrinsics must be rewritten to use equivalent SSE and SSE2 intrinsics.

## END Microsoft Specific

## See also

[Keywords](#)

[Built-in types](#)

[Data Type Ranges](#)

# \_\_m128

---

## Microsoft Specific

The **\_\_m128 data** type, for use with the Streaming SIMD Extensions and Streaming SIMD Extensions 2 instructions intrinsics, is defined in <xmmintrin.h>.

```
// data_types__m128.cpp
#include <xmmintrin.h>
int main() {
    __m128 x;
}
```

## Remarks

You should not access the **\_\_m128** fields directly. You can, however, see these types in the debugger. A variable of type **\_\_m128** maps to the XMM[0-7] registers.

Variables of type **\_\_m128** are automatically aligned on 16-byte boundaries.

The **\_\_m128** data type is not supported on ARM processors.

## END Microsoft Specific

## See also

[Keywords](#)

[Built-in types](#)

[Data Type Ranges](#)

# \_\_m128d

---

## Microsoft Specific

The **\_\_m128d** data type, for use with the Streaming SIMD Extensions 2 instructions intrinsics, is defined in `<emmintrin.h>`.

```
// data_types__m128d.cpp
#include <emmintrin.h>
int main() {
    __m128d x;
}
```

## Remarks

You should not access the **\_\_m128d** fields directly. You can, however, see these types in the debugger. A variable of type **\_\_m128** maps to the XMM[0-7] registers.

Variables of type **\_\_m128d** are automatically aligned on 16-byte boundaries.

The **\_\_m128d** data type is not supported on ARM processors.

## END Microsoft Specific

## See also

[Keywords](#)

[Built-in types](#)

[Data Type Ranges](#)

# \_\_m128i

---

## Microsoft Specific

The **\_\_m128i** data type, for use with the Streaming SIMD Extensions 2 (SSE2) instructions intrinsics, is defined in `<emmintrin.h>`.

```
// data_types__m128i.cpp
#include <emmintrin.h>
int main() {
    __m128i x;
}
```

## Remarks

You should not access the **\_\_m128i** fields directly. You can, however, see these types in the debugger. A variable of type **\_\_m128i** maps to the XMM[0-7] registers.

Variables of type **\_\_m128i** are automatically aligned on 16-byte boundaries.

[!NOTE] Using variables of type **\_\_m128i** will cause the compiler to generate the SSE2 `movdqa` instruction. This instruction does not cause a fault on Pentium III processors but will result in silent failure, with possible side effects caused by whatever instructions `movdqa` translates into on Pentium III processors.

The **\_\_m128i** data type is not supported on ARM processors.

## END Microsoft Specific

## See also

[Keywords](#)

[Built-in types](#)

[Data Type Ranges](#)

# \_\_ptr32, \_\_ptr64

---

## Microsoft Specific

**\_\_ptr32** represents a native pointer on a 32-bit system, while **\_\_ptr64** represents a native pointer on a 64-bit system.

The following example shows how to declare each of these pointer types:

```
int * __ptr32 p32;  
int * __ptr64 p64;
```

On a 32-bit system, a pointer declared with **\_\_ptr64** is truncated to a 32-bit pointer. On a 64-bit system, a pointer declared with **\_\_ptr32** is coerced to a 64-bit pointer.

[!NOTE] You cannot use **\_\_ptr32** or **\_\_ptr64** when compiling with **/clr:pure**. Otherwise, Compiler Error C2472 will be generated. The **/clr:pure** and **/clr:safe** compiler options are deprecated in Visual Studio 2015 and unsupported in Visual Studio 2017.

For compatibility with previous versions, **\_ptr32** and **\_ptr64** are synonyms for **\_\_ptr32** and **\_\_ptr64** unless compiler option **/Za** ([Disable language extensions](#)) is specified.

## Example

The following example shows how to declare and allocate pointers with the **\_\_ptr32** and **\_\_ptr64** keywords.

```
#include <cstdlib>  
#include <iostream>  
  
int main()  
{  
    using namespace std;  
  
    int * __ptr32 p32;  
    int * __ptr64 p64;  
  
    p32 = (int * __ptr32)malloc(4);  
    *p32 = 32;  
    cout << *p32 << endl;  
  
    p64 = (int * __ptr64)malloc(4);  
    *p64 = 64;  
    cout << *p64 << endl;  
}
```

32  
64

**END Microsoft Specific**

See also

[Built-in types](#)

# Numerical Limits (C++)

---

The two standard include files, `<limits.h>` and `<float.h>`, define the numerical limits, or minimum and maximum values that a variable of a given type can hold. These minimums and maximums are guaranteed to be portable to any C++ compiler that uses the same data representation as ANSI C. The `<limits.h>` include file defines the [numerical limits for integral types](#), and `<float.h>` defines the [numerical limits for floating types](#).

## See also

[Basic Concepts](#)



# Integer Limits

## Microsoft Specific

The limits for integer types are listed in the following table. These limits are also defined in the standard header file <limits.h>.

### Limits on Integer Constants

Constant	Meaning	Value
CHAR_BIT	Number of bits in the smallest variable that is not a bit field.	8
SCHAR_MIN	Minimum value for a variable of type <b>signed char</b> .	-128
SCHAR_MAX	Maximum value for a variable of type <b>signed char</b> .	127
UCHAR_MAX	Maximum value for a variable of type <b>unsigned char</b> .	255 (0xff)
CHAR_MIN	Minimum value for a variable of type <b>char</b> .	-128; 0 if /J option used
CHAR_MAX	Maximum value for a variable of type <b>char</b> .	127; 255 if /J option used
MB_LEN_MAX	Maximum number of bytes in a multicharacter constant.	5
SHRT_MIN	Minimum value for a variable of type <b>short</b> .	-32768
SHRT_MAX	Maximum value for a variable of type <b>short</b> .	32767
USHRT_MAX	Maximum value for a variable of type <b>unsigned short</b> .	65535 (0xffff)
INT_MIN	Minimum value for a variable of type <b>int</b> .	-2147483648
INT_MAX	Maximum value for a variable of type <b>int</b> .	2147483647
UINT_MAX	Maximum value for a variable of type <b>unsigned int</b> .	4294967295 (0xffffffff)
LONG_MIN	Minimum value for a variable of type <b>long</b> .	-2147483648
LONG_MAX	Maximum value for a variable of type <b>long</b> .	2147483647
ULONG_MAX	Maximum value for a variable of type <b>unsigned long</b> .	4294967295 (0xffffffff)
LLONG_MIN	Minimum value for a variable of type <b>long long</b>	-9223372036854775808
LLONG_MAX	Maximum value for a variable of type <b>long long</b>	9223372036854775807

Constant	Meaning	Value
ULLONG_MAX	Maximum value for a variable of type <b>unsigned long long</b>	18446744073709551615 (0xffffffffffffffff)

If a value exceeds the largest integer representation, the Microsoft compiler generates an error.

**END Microsoft Specific**

See also

[Floating Limits](#)

# Floating Limits

## Microsoft Specific

The following table lists the limits on the values of floating-point constants. These limits are also defined in the standard header file <float.h>.

### Limits on Floating-Point Constants

Constant	Meaning	Value
FLT_DIG	Number of digits, q, such that a floating-point number with q decimal digits can be rounded into a floating-point representation and back without loss of precision.	6
DBL_DIG		15
LDBL_DIG		15
FLT_EPSILON	Smallest positive number x, such that x + 1.0 is not equal to 1.0.	1.192092896e-07F
DBL_EPSILON		2.2204460492503131e-016
LDBL_EPSILON		2.2204460492503131e-016
FLT_GUARD		0
FLT_MANT_DIG	Number of digits in the radix specified by FLT_RADIX in the floating-point significand. The radix is 2; hence these values specify bits.	24
DBL_MANT_DIG		53
LDBL_MANT_DIG		53
FLT_MAX	Maximum representable floating-point number.	3.402823466e+38F
DBL_MAX		1.7976931348623158e+308
LDBL_MAX		1.7976931348623158e+308
FLT_MAX_10_EXP	Maximum integer such that 10 raised to that number is a representable floating-point number.	38
DBL_MAX_10_EXP		308
LDBL_MAX_10_EXP		308
FLT_MAX_EXP	Maximum integer such that FLT_RADIX raised to that number is a representable floating- point number.	128
DBL_MAX_EXP		1024
LDBL_MAX_EXP		1024
FLT_MIN	Minimum positive value.	1.175494351e-38F
DBL_MIN		2.2250738585072014e-308
LDBL_MIN		2.2250738585072014e-308
FLT_MIN_10_EXP	Minimum negative integer such that 10 raised to that number is a representable floating- point number.	-37
DBL_MIN_10_EXP		-307
LDBL_MIN_10_EXP		-307
FLT_MIN_EXP	Minimum negative integer such that FLT_RADIX raised to that number is a representable floating-point number.	-125
DBL_MIN_EXP		-1021
LDBL_MIN_EXP		-1021

Constant	Meaning	Value
FLT_NORMALIZE		0
FLT_RADIX		2
_DBL_RADIX	Radix of exponent representation.	2
_LDBL_RADIX		2
FLT_ROUNDS		1 (near)
_DBL_ROUNDS	Rounding mode for floating-point addition.	1 (near)
_LDBL_ROUNDS		1 (near)

[!NOTE] The information in the table may differ in future versions of the product.

**END Microsoft Specific**

See also

[Integer Limits](#)

# Declarations and definitions (C++)

---

A C++ program consists of various entities such as variables, functions, types, and namespaces. Each of these entities must be *declared* before they can be used. A declaration specifies a unique name for the entity, along with information about its type and other characteristics. In C++ the point at which a name is declared is the point at which it becomes visible to the compiler. You cannot refer to a function or class that is declared at some later point in the compilation unit. Variables should be declared as close as possible before the point at which they are used.

The following example shows some declarations:

```
#include <string>

void f(); // forward declaration

int main()
{
    const double pi = 3.14; //OK
    int i = f(2); //OK. f is forward-declared
    std::string str; // OK std::string is declared in <string> header
    C obj; // error! C not yet declared.
    j = 0; // error! No type specified.
    auto k = 0; // OK. type inferred as int by compiler.
}

int f(int i)
{
    return i + 42;
}

namespace N {
    class C{/*...*/};
}
```

On line 5, the `main` function is declared. On line 7, a **const** variable named `pi` is declared and *initialized*. On line 8, an integer `i` is declared and initialized with the value produced by the function `f`. The name `f` is visible to the compiler because of the *forward declaration* on line 3.

In line 9, a variable named `obj` of type `C` is declared. However, this declaration raises an error because `C` is not declared until later in the program, and is not forward-declared. To fix the error, you can either move the entire *definition* of `C` before `main` or else add a forward-declaration for it. This behavior is different from other languages such as C#, in which functions and classes can be used before their point of declaration in a source file.

In line 10, a variable named `str` of type `std::string` is declared. The name `std::string` is visible because it is introduced in the `string` header file which is merged into the source file in line 1. `std` is the namespace in which the `string` class is declared.

In line 11, an error is raised because the name `j` has not been declared. A declaration must provide a type, unlike other languages such as JavaScript. In line 12, the `auto` keyword is used, which tells the compiler to infer the type of `k` based on the value that it is initialized with. The compiler in this case chooses `int` for the type.

## Declaration scope

The name that is introduced by a declaration is valid within the *scope* where the declaration occurs. In the previous example, the variables that are declared inside the `main` function are *local variables*. You could declare another variable named `i` outside of `main`, at *global scope*, and it would be a completely separate entity. However, such duplication of names can lead to programmer confusion and errors, and should be avoided. In line 21, the class `C` is declared in the scope of the namespace `N`. The use of namespaces helps to avoid *name collisions*. Most C++ Standard Library names are declared within the `std` namespace. For more information about how scope rules interact with declarations, see [Scope](#).

## Definitions

Some entities, including functions, classes, enums, and constant variables, must be defined in addition to being declared. A *definition* provides the compiler with all the information it needs to generate machine code when the entity is used later in the program. In the previous example, line 3 contains a declaration for the function `f` but the *definition* for the function is provided in lines 15 through 18. On line 21, the class `C` is both declared and defined (although as defined the class doesn't do anything). A constant variable must be defined, in other words assigned a value, in the same statement in which it is declared. A declaration of a built-in type such as `int` is automatically a definition because the compiler knows how much space to allocate for it.

The following example shows declarations that are also definitions:

```
// Declare and define int variables i and j.
int i;
int j = 10;

// Declare enumeration suits.
enum suits { Spades = 1, Clubs, Hearts, Diamonds };

// Declare class CheckBox.
class CheckBox : public Control
{
public:
    Boolean IsChecked();
    virtual int ChangeState() = 0;
};
```

Here are some declarations that are not definitions:

```
extern int i;
char *strchr( const char *Str, const char Target );
```

---

## Typedefs and using statements

In older versions of C++, the [typedef](#) keyword is used to declare a new name that is an *alias* for another name. For example the type `std::string` is another name for `std::basic_string<char>`. It should be obvious why programmers use the typedef name and not the actual name. In modern C++, the [using](#) keyword is preferred over typedef, but the idea is the same: a new name is declared for an entity which is already declared and defined.

## Static class members

Because static class data members are discrete variables shared by all objects of the class, they must be defined and initialized outside the class definition. (For more information, see [Classes](#).)

## extern declarations

A C++ program might contain more than one [compilation unit](#). To declare an entity that is defined in a separate compilation unit, use the [extern](#) keyword. The information in the declaration is sufficient for the compiler, but if the definition of the entity cannot be found in the linking step, then the linker will raise an error.

## In this section

[Storage classes](#)

[const](#)

[constexpr](#)

[extern](#)

[Initializers](#)

[Aliases and typedefs](#)

[using declaration](#)

[volatile](#)

[decltype](#)

[Attributes in C++](#)

## See also

[Basic Concepts](#)

# Storage classes

---

A *storage class* in the context of C++ variable declarations is a type specifier that governs the lifetime, linkage, and memory location of objects. A given object can have only one storage class. Variables defined within a block have automatic storage unless otherwise specified using the **extern**, **static**, or **thread\_local** specifiers. Automatic objects and variables have no linkage; they are not visible to code outside the block. Memory is allocated for them automatically when execution enters the block and de-allocated when the block is exited.

## Notes

1. The **mutable** keyword may be considered a storage class specifier. However, it is only available in the member list of a class definition.
2. **Visual Studio 2010 and later:** The **auto** keyword is no longer a C++ storage-class specifier, and the **register** keyword is deprecated. **Visual Studio 2017 version 15.7 and later:** (available with `/std:c++17`): The **register** keyword is removed from the C++ language.

```
register int val; // warning C5033: 'register' is no longer a supported storage
class
```

## static

The **static** keyword can be used to declare variables and functions at global scope, namespace scope, and class scope. Static variables can also be declared at local scope.

Static duration means that the object or variable is allocated when the program starts and is deallocated when the program ends. External linkage means that the name of the variable is visible from outside the file in which the variable is declared. Conversely, internal linkage means that the name is not visible outside the file in which the variable is declared. By default, an object or variable that is defined in the global namespace has static duration and external linkage. The **static** keyword can be used in the following situations.

1. When you declare a variable or function at file scope (global and/or namespace scope), the **static** keyword specifies that the variable or function has internal linkage. When you declare a variable, the variable has static duration and the compiler initializes it to 0 unless you specify another value.
2. When you declare a variable in a function, the **static** keyword specifies that the variable retains its state between calls to that function.
3. When you declare a data member in a class declaration, the **static** keyword specifies that one copy of the member is shared by all instances of the class. A static data member must be defined at file scope. An integral data member that you declare as **const static** can have an initializer.
4. When you declare a member function in a class declaration, the **static** keyword specifies that the function is shared by all instances of the class. A static member function cannot access an instance member because the function does not have an implicit **this** pointer. To access an instance member, declare the function with a parameter that is an instance pointer or reference.



5. You cannot declare the members of a union as static. However, a globally declared anonymous union must be explicitly declared **static**.

This example shows how a variable declared **static** in a function retains its state between calls to that function.

```
// static1.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;
void showstat( int curr ) {
    static int nStatic;    // Value of nStatic is retained
                          // between each function call

    nStatic += curr;
    cout << "nStatic is " << nStatic << endl;
}

int main() {
    for ( int i = 0; i < 5; i++ )
        showstat( i );
}
```

```
nStatic is 0
nStatic is 1
nStatic is 3
nStatic is 6
nStatic is 10
```

This example shows the use of **static** in a class.

```
// static2.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;
class CMyClass {
public:
    static int m_i;
};

int CMyClass::m_i = 0;
CMyClass myObject1;
CMyClass myObject2;

int main() {
    cout << myObject1.m_i << endl;
    cout << myObject2.m_i << endl;
}
```

```

myObject1.m_i = 1;
cout << myObject1.m_i << endl;
cout << myObject2.m_i << endl;

myObject2.m_i = 2;
cout << myObject1.m_i << endl;
cout << myObject2.m_i << endl;

CMyClass::m_i = 3;
cout << myObject1.m_i << endl;
cout << myObject2.m_i << endl;
}

```

```

0
0
1
1
2
2
3
3

```

This example shows a local variable declared **static** in a member function. The static variable is available to the whole program; all instances of the type share the same copy of the static variable.

```

// static3.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;
struct C {
    void Test(int value) {
        static int var = 0;
        if (var == value)
            cout << "var == value" << endl;
        else
            cout << "var != value" << endl;

        var = value;
    }
};

int main() {
    C c1;
    C c2;
    c1.Test(100);
    c2.Test(100);
}

```

```
var != value
var == value
```

Starting in C++11, a static local variable initialization is guaranteed to be thread-safe. This feature is sometimes called *magic statics*. However, in a multithreaded application all subsequent assignments must be synchronized. The thread-safe static initialization feature can be disabled by using the `/Zc:threadSafeInit-` flag to avoid taking a dependency on the CRT.

## extern

Objects and variables declared as **extern** declare an object that is defined in another translation unit or in an enclosing scope as having external linkage. For more information, see [extern](#) and [Translation units and linkage](#).

## thread\_local (C++11)

A variable declared with the **thread\_local** specifier is accessible only on the thread on which it is created. The variable is created when the thread is created, and destroyed when the thread is destroyed. Each thread has its own copy of the variable. On Windows, **thread\_local** is functionally equivalent to the Microsoft-specific `__declspec( thread )` attribute.

```
thread_local float f = 42.0; // Global namespace. Not implicitly static.

struct S // cannot be applied to type definition
{
    thread_local int i; // Illegal. The member must be static.
    thread_local static char buf[10]; // OK
};

void DoSomething()
{
    // Apply thread_local to a local variable.
    // Implicitly "thread_local static S my_struct".
    thread_local S my_struct;
}
```

Things to note about the **thread\_local** specifier:

- Dynamically initialized thread-local variables in DLLs may not be correctly initialized on all calling threads. For more information, see [thread](#).
- The **thread\_local** specifier may be combined with **static** or **extern**.
- You can apply **thread\_local** only to data declarations and definitions; **thread\_local** cannot be used on function declarations or definitions.
- You can specify **thread\_local** only on data items with static storage duration. This includes global data objects (both **static** and **extern**), local static objects, and static data members of classes. Any local

variable declared **thread\_local** is implicitly static if no other storage class is provided; in other words, at block scope **thread\_local** is equivalent to `thread_local static`.

- You must specify **thread\_local** for both the declaration and the definition of a thread local object, whether the declaration and definition occur in the same file or separate files.

On Windows, **thread\_local** is functionally equivalent to `__declspec(thread)` except that `__declspec(thread)` can be applied to a type definition and is valid in C code. Whenever possible, use **thread\_local** because it is part of the C++ standard and is therefore more portable.

## register

**Visual Studio 2017 version 15.3 and later** (available with `/std:c++17`): The **register** keyword is no longer a supported storage class. The keyword is still reserved in the standard for future use.

```
register int val; // warning C5033: 'register' is no longer a supported storage
class
```

## Example: automatic vs. static initialization

A local automatic object or variable is initialized every time the flow of control reaches its definition. A local static object or variable is initialized the first time the flow of control reaches its definition.

Consider the following example, which defines a class that logs initialization and destruction of objects and then defines three objects, **I1**, **I2**, and **I3**:

```
// initialization_of_objects.cpp
// compile with: /EHsc
#include <iostream>
#include <string.h>
using namespace std;

// Define a class that logs initializations and destructions.
class InitDemo {
public:
    InitDemo( const char *szWhat );
    ~InitDemo();

private:
    char *szObjName;
    size_t sizeofObjName;
};

// Constructor for class InitDemo
InitDemo::InitDemo( const char *szWhat ) :
    szObjName(NULL), sizeofObjName(0) {
    if ( szWhat != 0 && strlen( szWhat ) > 0 ) {
        // Allocate storage for szObjName, then copy
        // initializer szWhat into szObjName, using
```

```

        // secured CRT functions.
        sizeofObjName = strlen( szWhat ) + 1;

        szObjName = new char[ sizeofObjName ];
        strcpy_s( szObjName, sizeofObjName, szWhat );

        cout << "Initializing: " << szObjName << "\n";
    }
    else {
        szObjName = 0;
    }
}

// Destructor for InitDemo
InitDemo::~InitDemo() {
    if( szObjName != 0 ) {
        cout << "Destroying: " << szObjName << "\n";
        delete szObjName;
    }
}

// Enter main function
int main() {
    InitDemo I1( "Auto I1" ); {
        cout << "In block.\n";
        InitDemo I2( "Auto I2" );
        static InitDemo I3( "Static I3" );
    }
    cout << "Exited block.\n";
}

```

```

Initializing: Auto I1
In block.
Initializing: Auto I2
Initializing: Static I3
Destroying: Auto I2
Exited block.
Destroying: Auto I1
Destroying: Static I3

```

This example demonstrates how and when the objects **I1**, **I2**, and **I3** are initialized and when they are destroyed.

There are several points to note about the program:

- First, **I1** and **I2** are automatically destroyed when the flow of control exits the block in which they are defined.
- Second, in C++, it is not necessary to declare objects or variables at the beginning of a block. Furthermore, these objects are initialized only when the flow of control reaches their definitions. (**I2**

and **I3** are examples of such definitions.) The output shows exactly when they are initialized.

- Finally, static local variables such as **I3** retain their values for the duration of the program, but are destroyed as the program terminates.

## See also

[Declarations and Definitions](#)

# auto (C++)

---

Deduces the type of a declared variable from its initialization expression.

[!NOTE] The C++ standard defines an original and a revised meaning for this keyword. Before Visual Studio 2010, the **auto** keyword declares a variable in the *automatic* storage class; that is, a variable that has a local lifetime. Starting with Visual Studio 2010, the **auto** keyword declares a variable whose type is deduced from the initialization expression in its declaration. The `/Zc:auto[-]` compiler option controls the meaning of the **auto** keyword.

## Syntax

```
auto declarator initializer;
```

```
[(auto param1, auto param2) {}];
```

## Remarks

The **auto** keyword directs the compiler to use the initialization expression of a declared variable, or lambda expression parameter, to deduce its type.

We recommend that you use the **auto** keyword for most situations—unless you really want a conversion—because it provides these benefits:

- **Robustness:** If the expression's type is changed—this includes when a function return type is changed—it just works.
- **Performance:** You're guaranteed that there will be no conversion.
- **Usability:** You don't have to worry about type name spelling difficulties and typos.
- **Efficiency:** Your coding can be more efficient.

Conversion cases in which you might not want to use **auto**:

- When you want a specific type and nothing else will do.
- Expression template helper types—for example, `(valarray+valarray)`.

To use the **auto** keyword, use it instead of a type to declare a variable, and specify an initialization expression. In addition, you can modify the **auto** keyword by using specifiers and declarators such as **const**, **volatile**, pointer (\*), reference (&), and rvalue reference (&&). The compiler evaluates the initialization expression and then uses that information to deduce the type of the variable.

The initialization expression can be an assignment (equal-sign syntax), a direct initialization (function-style syntax), an `operator new` expression, or the initialization expression can be the *for-range-declaration*

parameter in a [Range-based for Statement \(C++\)](#) statement. For more information, see [Initializers](#) and the code examples later in this document.

The **auto** keyword is a placeholder for a type, but it is not itself a type. Therefore, the **auto** keyword cannot be used in casts or operators such as [sizeof](#) and (for C++/CLI) [typeid](#).

## Usefulness

The **auto** keyword is a simple way to declare a variable that has a complicated type. For example, you can use **auto** to declare a variable where the initialization expression involves templates, pointers to functions, or pointers to members.

You can also use **auto** to declare and initialize a variable to a lambda expression. You can't declare the type of the variable yourself because the type of a lambda expression is known only to the compiler. For more information, see [Examples of Lambda Expressions](#).

## Trailing Return Types

You can use **auto**, together with the **decltype** type specifier, to help write template libraries. Use **auto** and **decltype** to declare a template function whose return type depends on the types of its template arguments. Or, use **auto** and **decltype** to declare a template function that wraps a call to another function, and then returns whatever is the return type of that other function. For more information, see [decltype](#).

## References and cv-qualifiers

Note that using **auto** drops references, const qualifiers, and volatile qualifiers. Consider the following example:

```
// cl.exe /analyze /EHsc /W4
#include <iostream>

using namespace std;

int main( )
{
    int count = 10;
    int& countRef = count;
    auto myAuto = countRef;

    countRef = 11;
    cout << count << " ";

    myAuto = 12;
    cout << count << endl;
}
```

In the previous example, myAuto is an int, not an int reference, so the output is 11 11, not 11 12 as would be the case if the reference qualifier had not been dropped by **auto**.



## Type deduction with braced initializers (C++14)

The following code example shows how to initialize an auto variable using braces. Note the difference between B and C and between A and E.

```
#include <initializer_list>

int main()
{
    // std::initializer_list<int>
    auto A = { 1, 2 };

    // std::initializer_list<int>
    auto B = { 3 };

    // int
    auto C{ 4 };

    // C3535: cannot deduce type for 'auto' from initializer list'
    auto D = { 5, 6.7 };

    // C3518 in a direct-list-initialization context the type for 'auto'
    // can only be deduced from a single initializer expression
    auto E{ 8, 9 };

    return 0;
}
```

## Restrictions and Error Messages

The following table lists the restrictions on the use of the **auto** keyword, and the corresponding diagnostic error message that the compiler emits.

Error number	Description
C3530	The <b>auto</b> keyword cannot be combined with any other type-specifier.
C3531	A symbol that is declared with the <b>auto</b> keyword must have an initializer.
C3532	You incorrectly used the <b>auto</b> keyword to declare a type. For example, you declared a method return type or an array.
C3533, C3539	A parameter or template argument cannot be declared with the <b>auto</b> keyword.
C3535	A method or template parameter cannot be declared with the <b>auto</b> keyword.
C3536	A symbol cannot be used before it is initialized. In practice, this means that a variable cannot be used to initialize itself.

Error number	Description
<a href="#">C3537</a>	You cannot cast to a type that is declared with the <b>auto</b> keyword.
<a href="#">C3538</a>	All the symbols in a declarator list that is declared with the <b>auto</b> keyword must resolve to the same type. For more information, see <a href="#">Declarations and Definitions</a> .
<a href="#">C3540</a> , <a href="#">C3541</a>	The <a href="#">sizeof</a> and <a href="#">typeid</a> operators cannot be applied to a symbol that is declared with the <b>auto</b> keyword.

## Examples

These code fragments illustrate some of the ways in which the **auto** keyword can be used.

The following declarations are equivalent. In the first statement, variable `j` is declared to be type **int**. In the second statement, variable `k` is deduced to be type **int** because the initialization expression (0) is an integer.

```
int j = 0; // Variable j is explicitly type int.
auto k = 0; // Variable k is implicitly type int because 0 is an integer.
```

The following declarations are equivalent, but the second declaration is simpler than the first. One of the most compelling reasons to use the **auto** keyword is simplicity.

```
map<int, list<string>>::iterator i = m.begin();
auto i = m.begin();
```

The following code fragment declares the type of variables `iter` and `elem` when the **for** and range **for** loops start.

```
// cl /EHsc /nologo /W4
#include <deque>
using namespace std;

int main()
{
    deque<double> dqDoubleData(10, 0.1);

    for (auto iter = dqDoubleData.begin(); iter != dqDoubleData.end(); ++iter)
    { /* ... */ }

    // prefer range-for loops with the following information in mind
    // (this applies to any range-for with auto, not just deque)

    for (auto elem : dqDoubleData) // COPIES elements, not much better than the
previous examples
    { /* ... */ }
```

```

    for (auto& elem : dqDoubleData) // observes and/or modifies elements IN-PLACE
    { /* ... */ }

    for (const auto& elem : dqDoubleData) // observes elements IN-PLACE
    { /* ... */ }
}

```

The following code fragment uses the **new** operator and pointer declaration to declare pointers.

```

double x = 12.34;
auto *y = new auto(x), **z = new auto(&x);

```

The next code fragment declares multiple symbols in each declaration statement. Notice that all of the symbols in each statement resolve to the same type.

```

auto x = 1, *y = &x, **z = &y; // Resolves to int.
auto a(2.01), *b (&a);          // Resolves to double.
auto c = 'a', *d(&c);           // Resolves to char.
auto m = 1, &n = m;             // Resolves to int.

```

This code fragment uses the conditional operator (**?:**) to declare variable **x** as an integer that has a value of 200:

```

int v1 = 100, v2 = 200;
auto x = v1 > v2 ? v1 : v2;

```

The following code fragment initializes variable **x** to type **int**, variable **y** to a reference to type **const int**, and variable **fp** to a pointer to a function that returns type **int**.

```

int f(int x) { return x; }
int main()
{
    auto x = f(0);
    const auto& y = f(1);
    int (*p)(int x);
    p = f;
    auto fp = p;
    //...
}

```

See also

auto Keyword

Keywords

/Zc:auto (Deduce Variable Type)

sizeof Operator

typeid

operator new

Declarations and Definitions

Examples of Lambda Expressions

Initializers

decltype

# const (C++)

---

When modifying a data declaration, the **const** keyword specifies that the object or variable is not modifiable.

## Syntax

```
const declaration ;  
member-function const ;
```

## const values

The **const** keyword specifies that a variable's value is constant and tells the compiler to prevent the programmer from modifying it.

```
// constant_values1.cpp  
int main() {  
    const int i = 5;  
    i = 10;    // C3892  
    i++;      // C2105  
}
```

In C++, you can use the **const** keyword instead of the `#define` preprocessor directive to define constant values. Values defined with **const** are subject to type checking, and can be used in place of constant expressions. In C++, you can specify the size of an array with a **const** variable as follows:

```
// constant_values2.cpp  
// compile with: /c  
const int maxarray = 255;  
char store_char[maxarray]; // allowed in C++; not allowed in C
```

In C, constant values default to external linkage, so they can appear only in source files. In C++, constant values default to internal linkage, which allows them to appear in header files.

The **const** keyword can also be used in pointer declarations.

```
// constant_values3.cpp  
int main() {  
    char *mybuf = 0, *yourbuf;  
    char *const aptr = mybuf;  
    *aptr = 'a';    // OK  
    aptr = yourbuf; // C3892  
}
```

A pointer to a variable declared as **const** can be assigned only to a pointer that is also declared as **const**.

```
// constant_values4.cpp
#include <stdio.h>
int main() {
    const char *mybuf = "test";
    char *yourbuf = "test2";
    printf_s("%s\n", mybuf);

    const char *bptr = mybuf;    // Pointer to constant data
    printf_s("%s\n", bptr);

    // *bptr = 'a';    // Error
}
```

You can use pointers to constant data as function parameters to prevent the function from modifying a parameter passed through a pointer.

For objects that are declared as **const**, you can only call constant member functions. This ensures that the constant object is never modified.

```
birthday.getMonth();    // Okay
birthday.setMonth( 4 ); // Error
```

You can call either constant or nonconstant member functions for a nonconstant object. You can also overload a member function using the **const** keyword; this allows a different version of the function to be called for constant and nonconstant objects.

You cannot declare constructors or destructors with the **const** keyword.

## const member functions

Declaring a member function with the **const** keyword specifies that the function is a "read-only" function that does not modify the object for which it is called. A constant member function cannot modify any non-static data members or call any member functions that aren't constant. To declare a constant member function, place the **const** keyword after the closing parenthesis of the argument list. The **const** keyword is required in both the declaration and the definition.

```
// constant_member_function.cpp
class Date
{
public:
    Date( int mn, int dy, int yr );
    int getMonth() const;    // A read-only function
    void setMonth( int mn ); // A write function; can't be const
private:
    int month;
```

```
};

int Date::getMonth() const
{
    return month;          // Doesn't modify anything
}
void Date::setMonth( int mn )
{
    month = mn;            // Modifies data member
}
int main()
{
    Date MyDate( 7, 4, 1998 );
    const Date BirthDate( 1, 18, 1953 );
    MyDate.setMonth( 4 );   // Okay
    BirthDate.getMonth();   // Okay
    BirthDate.setMonth( 4 ); // C2662 Error
}
```

## C and C++ const differences

When you declare a variable as **const** in a C source code file, you do so as:

```
const int i = 2;
```

You can then use this variable in another module as follows:

```
extern const int i;
```

But to get the same behavior in C++, you must declare your **const** variable as:

```
extern const int i = 2;
```

If you wish to declare an **extern** variable in a C++ source code file for use in a C source code file, use:

```
extern "C" const int x=10;
```

to prevent name mangling by the C++ compiler.

## Remarks

When following a member function's parameter list, the **const** keyword specifies that the function does not modify the object for which it is invoked.

For more information on **const**, see the following topics:

- [const and volatile Pointers](#)
- [Type Qualifiers \(C Language Reference\)](#)
- [volatile](#)
- [#define](#)

See also

[Keywords](#)



# constexpr (C++)

---

The keyword **constexpr** was introduced in C++11 and improved in C++14. It means *constant expression*. Like **const**, it can be applied to variables: A compiler error is raised when any code attempts to modify the value. Unlike **const**, **constexpr** can also be applied to functions and class constructors. **constexpr** indicates that the value, or return value, is constant and, where possible, is computed at compile time.

A **constexpr** integral value can be used wherever a const integer is required, such as in template arguments and array declarations. And when a value is computed at compile time instead of run time, it helps your program run faster and use less memory.

To limit the complexity of compile-time constant computations, and their potential impacts on compilation time, the C++14 standard requires the types in constant expressions to be [literal types](#).

## Syntax

```
constexpr literal-type identifier = constant-expression ;  
constexpr literal-type identifier { constant-expression } ;  
constexpr literal-type identifier ( params ) ;  
constexpr ctor ( params ) ;
```

## Parameters

*params*

One or more parameters, each of which must be a literal type and must itself be a constant expression.

## Return value

A **constexpr** variable or function must return a [literal type](#).

## constexpr variables

The primary difference between **const** and **constexpr** variables is that the initialization of a **const** variable can be deferred until run time. A **constexpr** variable must be initialized at compile time. All **constexpr** variables are **const**.

- A variable can be declared with **constexpr**, when it has a literal type and is initialized. If the initialization is performed by a constructor, the constructor must be declared as **constexpr**.
- A reference may be declared as **constexpr** when both these conditions are met: The referenced object is initialized by a constant expression, and any implicit conversions invoked during initialization are also constant expressions.
- All declarations of a **constexpr** variable or function must have the **constexpr** specifier.

```
constexpr float x = 42.0;  
constexpr float y{108};  
constexpr float z = exp(5, 3);
```

```
constexpr int i; // Error! Not initialized
int j = 0;
constexpr int k = j + 1; //Error! j not a constant expression
```

## constexpr functions

A **constexpr** function is one whose return value is computable at compile time when consuming code requires it. Consuming code requires the return value at compile time to initialize a **constexpr** variable, or to provide a non-type template argument. When its arguments are **constexpr** values, a **constexpr** function produces a compile-time constant. When called with non-**constexpr** arguments, or when its value isn't required at compile time, it produces a value at run time like a regular function. (This dual behavior saves you from having to write **constexpr** and non-**constexpr** versions of the same function.)

A **constexpr** function or constructor is implicitly **inline**.

The following rules apply to constexpr functions:

- A **constexpr** function must accept and return only [literal types](#).
- A **constexpr** function can be recursive.
- It can't be [virtual](#). A constructor can't be defined as **constexpr** when the enclosing class has any virtual base classes.
- The body can be defined as `= default` or `= delete`.
- The body can contain no **goto** statements or **try** blocks.
- An explicit specialization of a non-**constexpr** template can be declared as **constexpr**:
- An explicit specialization of a **constexpr** template doesn't also have to be **constexpr**:

The following rules apply to **constexpr** functions in Visual Studio 2017 and later:

- It may contain **if** and **switch** statements, and all looping statements including **for**, range-based **for**, **while**, and **do-while**.
- It may contain local variable declarations, but the variable must be initialized. It must be a literal type, and can't be **static** or thread-local. The locally declared variable isn't required to be **const**, and may mutate.
- A **constexpr** non-**static** member function isn't required to be implicitly **const**.

```
constexpr float exp(float x, int n)
{
    return n == 0 ? 1 :
        n % 2 == 0 ? exp(x * x, n / 2) :
        exp(x * x, (n - 1) / 2) * x;
};
```

[!TIP] In the Visual Studio debugger, when debugging a non-optimised Debug build, you can tell whether a **constexpr** function is being evaluated at compile time by putting a breakpoint inside it. If the breakpoint is hit, the function was called at run-time. If not, then the function was called at compile time.

## extern constexpr

The `/Zc:externConstexpr` compiler option causes the compiler to apply [external linkage](#) to variables declared by using **extern constexpr**. In earlier versions of Visual Studio, either by default or when `/Zc:externConstexpr-` is specified, Visual Studio applies internal linkage to **constexpr** variables even when the **extern** keyword is used. The `/Zc:externConstexpr` option is available starting in Visual Studio 2017 Update 15.6, and is off by default. The `/permissive-` option doesn't enable `/Zc:externConstexpr`.

## Example

The following example shows **constexpr** variables, functions, and a user-defined type. In the last statement in `main()`, the **constexpr** member function `GetValue()` is a run-time call because the value isn't required to be known at compile time.

```
// constexpr.cpp
// Compile with: cl /EHsc /W4 constexpr.cpp
#include <iostream>

using namespace std;

// Pass by value
constexpr float exp(float x, int n)
{
    return n == 0 ? 1 :
           n % 2 == 0 ? exp(x * x, n / 2) :
           exp(x * x, (n - 1) / 2) * x;
};

// Pass by reference
constexpr float exp2(const float& x, const int& n)
{
    return n == 0 ? 1 :
           n % 2 == 0 ? exp2(x * x, n / 2) :
           exp2(x * x, (n - 1) / 2) * x;
};

// Compile-time computation of array length
template<typename T, int N>
constexpr int length(const T(&)[N])
{
    return N;
}

// Recursive constexpr function
constexpr int fac(int n)
{
    if (n <= 1)
        return 1;
    return n * fac(n - 1);
}
```

```

    return n == 1 ? 1 : n * fac(n - 1);
}

// User-defined type
class Foo
{
public:
    constexpr explicit Foo(int i) : _i(i) {}
    constexpr int GetValue() const
    {
        return _i;
    }
private:
    int _i;
};

int main()
{
    // foo is const:
    constexpr Foo foo(5);
    // foo = Foo(6); //Error!

    // Compile time:
    constexpr float x = exp(5, 3);
    constexpr float y { exp(2, 5) };
    constexpr int val = foo.GetValue();
    constexpr int f5 = fac(5);
    const int nums[] { 1, 2, 3, 4 };
    const int nums2[length(nums) * 2] { 1, 2, 3, 4, 5, 6, 7, 8 };

    // Run time:
    cout << "The value of foo is " << foo.GetValue() << endl;
}

```

## Requirements

Visual Studio 2015 or later.

## See also

[Declarations and definitions](#)

[const](#)

# extern (C++)

---

The **extern** keyword may be applied to a global variable, function, or template declaration. It specifies that the symbol has *external linkage*. For background information on linkage and why the use of global variables is discouraged, see [Translation units and linkage](#).

The **extern** keyword has four meanings depending on the context:

- In a non-**const** global variable declaration, **extern** specifies that the variable or function is defined in another translation unit. The **extern** must be applied in all files except the one where the variable is defined.
- In a **const** variable declaration, it specifies that the variable has external linkage. The **extern** must be applied to all declarations in all files. (Global **const** variables have internal linkage by default.)
- **extern "C"** specifies that the function is defined elsewhere and uses the C-language calling convention. The **extern "C"** modifier may also be applied to multiple function declarations in a block.
- In a template declaration, **extern** specifies that the template has already been instantiated elsewhere. **extern** tells the compiler it can reuse the other instantiation, rather than create a new one at the current location. For more information about this use of **extern**, see [Explicit instantiation](#).

## extern linkage for non-const globals

When the linker sees **extern** before a global variable declaration, it looks for the definition in another translation unit. Declarations of non-**const** variables at global scope are external by default. Only apply **extern** to the declarations that don't provide the definition.

```
//fileA.cpp
int i = 42; // declaration and definition

//fileB.cpp
extern int i; // declaration only. same as i in FileA

//fileC.cpp
extern int i; // declaration only. same as i in FileA

//fileD.cpp
int i = 43; // LNK2005! 'i' already has a definition.
extern int i = 43; // same error (extern is ignored on definitions)
```

## extern linkage for const globals

A **const** global variable has internal linkage by default. If you want the variable to have external linkage, apply the **extern** keyword to the definition, and to all other declarations in other files:

```
//fileA.cpp
extern const int i = 42; // extern const definition

//fileB.cpp
extern const int i; // declaration only. same as i in FileA
```

## extern constexpr linkage

In Visual Studio 2017 version 15.3 and earlier, the compiler always gave a **constexpr** variable internal linkage, even when the variable was marked **extern**. In Visual Studio 2017 version 15.5 and later, the `/Zc:externConstexpr` compiler switch enables correct standards-conforming behavior. Eventually the option will become the default. The `/permissive-` option doesn't enable `/Zc:externConstexpr`.

```
extern constexpr int x = 10; //error LNK2005: "int const x" already defined
```

If a header file contains a variable declared **extern constexpr**, it must be marked `__declspec(selectany)` to correctly have its duplicate declarations combined:

```
extern constexpr __declspec(selectany) int x = 10;
```

## extern "C" and extern "C++" function declarations

In C++, when used with a string, **extern** specifies that the linkage conventions of another language are being used for the declarator(s). C functions and data can be accessed only if they're previously declared as having C linkage. However, they must be defined in a separately compiled translation unit.

Microsoft C++ supports the strings **"C"** and **"C++"** in the *string-literal* field. All of the standard include files use the **extern "C"** syntax to allow the run-time library functions to be used in C++ programs.

## Example

The following example shows how to declare names that have C linkage:

```
// Declare printf with C linkage.
extern "C" int printf(const char *fmt, ...);

// Cause everything in the specified
// header files to have C linkage.
extern "C" {
    // add your #include statements here
    #include <stdio.h>
}

// Declare the two functions ShowChar
// and GetChar with C linkage.
```

```

extern "C" {
    char ShowChar(char ch);
    char GetChar(void);
}

// Define the two functions
// ShowChar and GetChar with C linkage.
extern "C" char ShowChar(char ch) {
    putchar(ch);
    return ch;
}

extern "C" char GetChar(void) {
    char ch;
    ch = getchar();
    return ch;
}

// Declare a global variable, errno, with C linkage.
extern "C" int errno;

```

If a function has more than one linkage specification, they must agree. It's an error to declare functions as having both C and C++ linkage. Furthermore, if two declarations for a function occur in a program — one with a linkage specification and one without — the declaration with the linkage specification must be first. Any redundant declarations of functions that already have linkage specification are given the linkage specified in the first declaration. For example:

```

extern "C" int CFunc1();
...
int CFunc1();           // Redeclaration is benign; C linkage is
                        // retained.

int CFunc2();
...
extern "C" int CFunc2(); // Error: not the first declaration of
                        // CFunc2; cannot contain linkage
                        // specifier.

```

## See also

[Keywords](#)

[Translation units and linkage](#)

[extern Storage-Class Specifier in C](#)

[Behavior of Identifiers in C](#)

[Linkage in C](#)

# Initializers

---

An initializer specifies the initial value of a variable. You can initialize variables in these contexts:

- In the definition of a variable:

```
int i = 3;
Point p1{ 1, 2 };
```

- As one of the parameters of a function:

```
set_point(Point{ 5, 6 });
```

- As the return value of a function:

```
Point get_new_point(int x, int y) { return { x, y }; }
Point get_new_point(int x, int y) { return Point{ x, y }; }
```

Initializers may take these forms:

- An expression (or a comma-separated list of expressions) in parentheses:

```
Point p1(1, 2);
```

- An equals sign followed by an expression:

```
string s = "hello";
```

- A braced initializer list. The list may be empty or may consist of a set of lists, as in the following example:

```
struct Point{
    int x;
    int y;
};
class PointConsumer{
public:
    void set_point(Point p){};
    void set_points(initializer_list<Point> my_list){};
};
```



```
int main() {
    PointConsumer pc{};
    pc.set_point({});
    pc.set_point({ 3, 4 });
    pc.set_points({ { 3, 4 }, { 5, 6 } });
}
```

## Kinds of initialization

There are several kinds of initialization, which may occur at different points in program execution. Different kinds of initialization are not mutually exclusive—for example, list initialization can trigger value initialization and in other circumstances, it can trigger aggregate initialization.

### Zero initialization

Zero initialization is the setting of a variable to a zero value implicitly converted to the type:

- Numeric variables are initialized to 0 (or 0.0, or 0.0000000000, etc.).
- Char variables are initialized to `'\0'`.
- Pointers are initialized to **nullptr**.
- Arrays, POD classes, structs, and unions have their members initialized to a zero value.

Zero initialization is performed at different times:

- At program startup, for all named variables that have static duration. These variables may later be initialized again.
- During value initialization, for scalar types and POD class types that are initialized by using empty braces.
- For arrays that have only a subset of their members initialized.

Here are some examples of zero initialization:

```
struct my_struct{
    int i;
    char c;
};

int i0;           // zero-initialized to 0
int main() {
    static float f1; // zero-initialized to 0.000000000
    double d{};      // zero-initialized to 0.000000000000000000
    int* ptr{};       // initialized to nullptr
    char s_array[3]{'a', 'b'}; // the third char is initialized to '\0'
    int int_array[5] = { 8, 9, 10 }; // the fourth and fifth ints are initialized
to 0
    my_struct a_struct{}; // i = 0, c = '\0'
}
```

## Default initialization

Default initialization for classes, structs, and unions is initialization with a default constructor. The default constructor can be called with no initialization expression or with the **new** keyword:

```
MyClass mc1;  
MyClass* mc3 = new MyClass;
```

If the class, struct, or union does not have a default constructor, the compiler emits an error.

Scalar variables are default initialized when they are defined with no initialization expression. They have indeterminate values.

```
int i1;  
float f;  
char c;
```

Arrays are default initialized when they are defined with no initialization expression. When an array is default-initialized, its members are default initialized and have indeterminate values, as in the following example:

```
int int_arr[3];
```

If the array members do not have a default constructor, the compiler emits an error.

## Default initialization of constant variables

Constant variables must be declared together with an initializer. If they are scalar types they cause a compiler error, and if they are class types that have a default constructor they cause a warning:

```
class MyClass{};  
int main() {  
    //const int i2;    // compiler error C2734: const object must be initialized if  
not extern  
    //const char c2;  // same error  
    const MyClass mc1; // compiler error C4269: 'const automatic data initialized  
with compiler generated default constructor produces unreliable results  
}
```

## Default initialization of static variables

Static variables that are declared with no initializer are initialized to 0 (implicitly converted to the type).

```

class MyClass {
private:
    int m_int;
    char m_char;
};

int main() {
    static int int1;        // 0
    static char char1;      // '\0'
    static bool bool1;      // false
    static MyClass mc1;     // {0, '\0'}
}

```

For more information about initialization of global static objects, see [main function and command-line arguments](#).

## Value initialization

Value initialization occurs in the following cases:

- a named value is initialized using empty brace initialization
- an anonymous temporary object is initialized using empty parentheses or braces
- an object is initialized with the **new** keyword plus empty parentheses or braces

Value initialization does the following:

- for classes with at least one public constructor, the default constructor is called
- for non-union classes with no declared constructors, the object is zero-initialized and the default constructor is called
- for arrays, every element is value-initialized
- in all other cases, the variable is zero initialized

```

class BaseClass {
private:
    int m_int;
};

int main() {
    BaseClass bc{};        // class is initialized
    BaseClass* bc2 = new BaseClass(); // class is initialized, m_int value is 0
    int int_arr[3]{};      // value of all members is 0
    int a{};               // value of a is 0
    double b{};            // value of b is 0.000000000000000000
}

```

## Copy initialization

Copy initialization is the initialization of one object using a different object. It occurs in the following cases:

- a variable is initialized using an equals sign
- an argument is passed to a function
- an object is returned from a function
- an exception is thrown or caught
- a non-static data member is initialized using an equals sign
- class, struct, and union members are initialized by copy initialization during aggregate initialization. See [Aggregate initialization](#) for examples.

The following code shows several examples of copy initialization:

```
#include <iostream>
using namespace std;

class MyClass{
public:
    MyClass(int myInt) {}
    void set_int(int myInt) { m_int = myInt; }
    int get_int() const { return m_int; }
private:
    int m_int = 7; // copy initialization of m_int
};

class MyException : public exception{};
int main() {
    int i = 5;           // copy initialization of i
    MyClass mc1{ i };
    MyClass mc2 = mc1;   // copy initialization of mc2 from mc1
    MyClass mc1.set_int(i); // copy initialization of parameter from i
    int i2 = mc2.get_int(); // copy initialization of i2 from return value of
    get_int()

    try{
        throw MyException();
    }
    catch (MyException ex){ // copy initialization of ex
        cout << ex.what();
    }
}
```

Copy initialization cannot invoke explicit constructors.

```
vector<int> v = 10; // the constructor is explicit; compiler error C2440: cannot
convert from 'int' to 'std::vector<int,std::allocator<_Ty>>'
regex r = "a.*b"; // the constructor is explicit; same error
shared_ptr<int> sp = new int(1729); // the constructor is explicit; same error
```

In some cases, if the copy constructor of the class is deleted or inaccessible, copy initialization causes a compiler error.

## Direct initialization

Direct initialization is initialization using (non-empty) braces or parentheses. Unlike copy initialization, it can invoke explicit constructors. It occurs in the following cases:

- a variable is initialized with non-empty braces or parentheses
- a variable is initialized with the **new** keyword plus non-empty braces or parentheses
- a variable is initialized with **static\_cast**
- in a constructor, base classes and non-static members are initialized with an initializer list
- in the copy of a captured variable inside a lambda expression

The following code shows some examples of direct initialization:

```
class BaseClass{
public:
    BaseClass(int n) :m_int(n){} // m_int is direct initialized
private:
    int m_int;
};

class DerivedClass : public BaseClass{
public:
    // BaseClass and m_char are direct initialized
    DerivedClass(int n, char c) : BaseClass(n), m_char(c) {}
private:
    char m_char;
};

int main(){
    BaseClass bc1(5);
    DerivedClass dc1{ 1, 'c' };
    BaseClass* bc2 = new BaseClass(7);
    BaseClass bc3 = static_cast<BaseClass>(dc1);

    int a = 1;
    function<int()> func = [a]() { return a + 1; }; // a is direct initialized
    int n = func();
}
```

## List initialization

List initialization occurs when a variable is initialized using a braced initializer list. Braced initializer lists can be used in the following cases:

- a variable is initialized
- a class is initialized with the **new** keyword
- an object is returned from a function
- an argument passed to a function
- one of the arguments in a direct initialization
- in a non-static data member initializer
- in a constructor initializer list

The following code shows some examples of list initialization:

```
class MyClass {
public:
    MyClass(int myInt, char myChar) {}
private:
    int m_int[]{ 3 };
    char m_char;
};
class MyClassConsumer{
public:
    void set_class(MyClass c) {}
    MyClass get_class() { return MyClass{ 0, '\0' }; }
};
struct MyStruct{
    int my_int;
    char my_char;
    MyClass my_class;
};
int main() {
    MyClass mc1{ 1, 'a' };
    MyClass* mc2 = new MyClass{ 2, 'b' };
    MyClass mc3 = { 3, 'c' };

    MyClassConsumer mcc;
    mcc.set_class(MyClass{ 3, 'c' });
    mcc.set_class({ 4, 'd' });

    MyStruct ms1{ 1, 'a', { 2, 'b' } };
}
```

## Aggregate initialization

Aggregate initialization is a form of list initialization for arrays or class types (often structs or unions) that have:

- no private or protected members
- no user-provided constructors, except for explicitly defaulted or deleted constructors
- no base classes
- no virtual member functions

[!NOTE]

In Visual Studio 2015 and earlier, an aggregate is not allowed to have brace-or-equal initializers for non-static members. This restriction was removed in the C++14 standard and implemented in Visual Studio 2017.

Aggregate initializers consist of a braced initialization list, with or without an equals sign, as in the following example:

```
#include <iostream>
using namespace std;

struct MyAggregate{
    int myInt;
    char myChar;
};

struct MyAggregate2{
    int myInt;
    char myChar = 'Z'; // member-initializer OK in C++14
};

int main() {
    MyAggregate agg1{ 1, 'c' };
    MyAggregate2 agg2{2};
    cout << "agg1: " << agg1.myChar << ": " << agg1.myInt << endl;
    cout << "agg2: " << agg2.myChar << ": " << agg2.myInt << endl;

    int myArr1[] { 1, 2, 3, 4 };
    int myArr2[3] = { 5, 6, 7 };
    int myArr3[5] = { 8, 9, 10 };

    cout << "myArr1: ";
    for (int i : myArr1){
        cout << i << " ";
    }
    cout << endl;

    cout << "myArr3: ";
    for (auto const &i : myArr3) {
        cout << i << " ";
    }
}
```

```
    cout << endl;
}
```

You should see the following output:

```
agg1: c: 1
agg2: Z: 2
myArr1: 1 2 3 4
myArr3: 8 9 10 0 0
```

[!IMPORTANT] Array members that are declared but not explicitly initialized during aggregate initialization are zero-initialized, as in `myArr3` above.

## Initializing unions and structs

If a union does not have a constructor, you can initialize it with a single value (or with another instance of a union). The value is used to initialize the first non-static field. This is different from struct initialization, in which the first value in the initializer is used to initialize the first field, the second to initialize the second field, and so on. Compare the initialization of unions and structs in the following example:

```
struct MyStruct {
    int myInt;
    char myChar;
};
union MyUnion {
    int my_int;
    char my_char;
    bool my_bool;
    MyStruct my_struct;
};

int main() {
    MyUnion mu1{ 'a' }; // my_int = 97, my_char = 'a', my_bool = true, {myInt =
97, myChar = '\0'}
    MyUnion mu2{ 1 }; // my_int = 1, my_char = 'x1', my_bool = true, {myInt = 1,
myChar = '\0'}
    MyUnion mu3{}; // my_int = 0, my_char = '\0', my_bool = false, {myInt =
0, myChar = '\0'}
    MyUnion mu4 = mu3; // my_int = 0, my_char = '\0', my_bool = false, {myInt =
0, myChar = '\0'}
    //MyUnion mu5{ 1, 'a', true }; // compiler error: C2078: too many
initializers
    //MyUnion mu6 = 'a'; // compiler error: C2440: cannot convert from
'char' to 'MyUnion'
    //MyUnion mu7 = 1; // compiler error: C2440: cannot convert from
'int' to 'MyUnion'

    MyStruct ms1{ 'a' }; // myInt = 97, myChar = '\0'
```



```

MyStruct ms2{ 1 };           // myInt = 1, myChar = '\0'
MyStruct ms3{};             // myInt = 0, myChar = '\0'
MyStruct ms4{1, 'a'};       // myInt = 1, myChar = 'a'
MyStruct ms5 = { 2, 'b' };   // myInt = 2, myChar = 'b'
}

```

## Initializing aggregates that contain aggregates

Aggregate types can contain other aggregate types, for example arrays of arrays, arrays of structs, and so on. These types are initialized by using nested sets of braces, for example:

```

struct MyStruct {
    int myInt;
    char myChar;
};
int main() {
    int intArr1[2][2]{{ 1, 2 }, { 3, 4 }};
    int intArr3[2][2] = {1, 2, 3, 4};
    MyStruct structArr[]{{ 1, 'a' }, { 2, 'b' }, {3, 'c' }};
}

```

## Reference initialization

Variables of reference type must be initialized with an object of the type from which the reference type is derived, or with an object of a type that can be converted to the type from which the reference type is derived. For example:

```

// initializing_references.cpp
int iVar;
long lVar;
int main()
{
    long& LongRef1 = lVar;           // No conversion required.
    long& LongRef2 = iVar;           // Error C2440
    const long& LongRef3 = iVar;     // OK
    LongRef1 = 23L;                  // Change lVar through a reference.
    LongRef2 = 11L;                  // Change iVar through a reference.
    LongRef3 = 11L;                  // Error C3892
}

```

The only way to initialize a reference with a temporary object is to initialize a constant temporary object. Once initialized, a reference-type variable always points to the same object; it cannot be modified to point to another object.

Although the syntax can be the same, initialization of reference-type variables and assignment to reference-type variables are semantically different. In the preceding example, the assignments that change `iVar` and

**lVar** look similar to the initializations, but have different effects. The initialization specifies the object to which the reference-type variable points; the assignment assigns to the referred-to object through the reference.

Because both passing an argument of reference type to a function and returning a value of reference type from a function are initializations, the formal arguments to a function are initialized correctly, as are the references returned.

Reference-type variables can be declared without initializers only in the following:

- Function declarations (prototypes). For example:

```
int func( int& );
```

- Function-return type declarations. For example:

```
int& func( int& );
```

- Declaration of a reference-type class member. For example:

```
class c {public:  int& i;};
```

- Declaration of a variable explicitly specified as **extern**. For example:

```
extern int& iVal;
```

When initializing a reference-type variable, the compiler uses the decision graph shown in the following figure to select between creating a reference to an object or creating a temporary object to which the reference points.



Decision graph for initialization of reference types

Decision graph for initialization of reference types

References to **volatile** types (declared as **volatile** *typename& identifier*) can be initialized with **volatile** objects of the same type or with objects that have not been declared as **volatile**. They cannot, however, be initialized with **const** objects of that type. Similarly, references to **const** types (declared as **const** *typename& identifier*) can be initialized with **const** objects of the same type (or anything that has a conversion to that type or with objects that have not been declared as **const**). They cannot, however, be initialized with **volatile** objects of that type.

References that are not qualified with either the **const** or **volatile** keyword can be initialized only with objects declared as neither **const** nor **volatile**.

Initialization of external variables

Declarations of automatic, static, and external variables can contain initializers. However, declarations of external variables can contain initializers only if the variables are not declared as **extern**.

# Aliases and typedefs (C++)

---

You can use an *alias declaration* to declare a name to use as a synonym for a previously declared type. (This mechanism is also referred to informally as a *type alias*). You can also use this mechanism to create an *alias template*, which can be particularly useful for custom allocators.

## Syntax

```
using identifier = type;
```

## Remarks

*identifier*

The name of the alias.

*type*

The type identifier you are creating an alias for.

An alias does not introduce a new type and cannot change the meaning of an existing type name.

The simplest form of an alias is equivalent to the **typedef** mechanism from C++03:

```
// C++11
using counter = long;

// C++03 equivalent:
// typedef long counter;
```

Both of these enable the creation of variables of type "counter". Something more useful would be a type alias like this one for `std::ios_base::fmtflags`:

```
// C++11
using fmtfl = std::ios_base::fmtflags;

// C++03 equivalent:
// typedef std::ios_base::fmtflags fmtfl;

fmtfl fl_orig = std::cout.flags();
fmtfl fl_hex = (fl_orig & ~std::cout.basefield) | std::cout.showbase |
std::cout.hex;
// ...
std::cout.flags(fl_hex);
```

Aliases also work with function pointers, but are much more readable than the equivalent typedef:

```
// C++11
using func = void (*)(int);

// C++03 equivalent:
// typedef void (*func)(int);

// func can be assigned to a function pointer value
void actual_function(int arg) { /* some code */ }
func fptr = &actual_function;
```

A limitation of the **typedef** mechanism is that it doesn't work with templates. However, the type alias syntax in C++11 enables the creation of alias templates:

```
template<typename T> using ptr = T*;

// the name 'ptr<T>' is now an alias for pointer to T
ptr<int> ptr_int;
```

## Example

The following example demonstrates how to use an alias template with a custom allocator—in this case, an integer vector type. You can substitute any type for **int** to create a convenient alias to hide the complex parameter lists in your main functional code. By using the custom allocator throughout your code you can improve readability and reduce the risk of introducing bugs caused by typos.

```
#include <stdlib.h>
#include <new>

template <typename T> struct MyAlloc {
    typedef T value_type;

    MyAlloc() { }
    template <typename U> MyAlloc(const MyAlloc<U>&) { }

    bool operator==(const MyAlloc&) const { return true; }
    bool operator!=(const MyAlloc&) const { return false; }

    T * allocate(const size_t n) const {
        if (n == 0) {
            return nullptr;
        }

        if (n > static_cast<size_t>(-1) / sizeof(T)) {
            throw std::bad_array_new_length();
        }

        void * const pv = malloc(n * sizeof(T));
```

```

        if (!pv) {
            throw std::bad_alloc();
        }

        return static_cast<T *>(pv);
    }

    void deallocate(T * const p, size_t) const {
        free(p);
    }
};

#include <vector>
using MyIntVector = std::vector<int, MyAlloc<int>>;

#include <iostream>

int main ()
{
    MyIntVector foov = { 1701, 1764, 1664 };

    for (auto a: foov) std::cout << a << " ";
    std::cout << "\n";

    return 0;
}

```

```
1701 1764 1664
```

## Typedefs

A **typedef** declaration introduces a name that, within its scope, becomes a synonym for the type given by the *type-declaration* portion of the declaration.

You can use typedef declarations to construct shorter or more meaningful names for types already defined by the language or for types that you have declared. Typedef names allow you to encapsulate implementation details that may change.

In contrast to the **class**, **struct**, **union**, and **enum** declarations, **typedef** declarations do not introduce new types — they introduce new names for existing types.

Names declared using **typedef** occupy the same namespace as other identifiers (except statement labels). Therefore, they cannot use the same identifier as a previously declared name, except in a class-type declaration. Consider the following example:

```

// typedef_names1.cpp
// C2377 expected

```

```
typedef unsigned long UL;    // Declare a typedef name, UL.
int UL;                     // C2377: redefined.
```

The name-hiding rules that pertain to other identifiers also govern the visibility of names declared using **typedef**. Therefore, the following example is legal in C++:

```
// typedef_names2.cpp
typedef unsigned long UL;    // Declare a typedef name, UL
int main()
{
    unsigned int UL;        // Redeclaration hides typedef name
}

// typedef UL back in scope
```

```
// typedef_specifier1.cpp
typedef char FlagType;

int main()
{
}

void myproc( int )
{
    int FlagType;
}
```

When declaring a local-scope identifier by the same name as a typedef, or when declaring a member of a structure or union in the same scope or in an inner scope, the type specifier must be specified. For example:

```
typedef char FlagType;
const FlagType x;
```

To reuse the `FlagType` name for an identifier, a structure member, or a union member, the type must be provided:

```
const int FlagType; // Type specifier required
```

It is not sufficient to say

```
const FlagType; // Incomplete specification
```

because the `FlagType` is taken to be part of the type, not an identifier that is being redeclared. This declaration is taken to be an illegal declaration like

```
int; // Illegal declaration
```

You can declare any type with `typedef`, including pointer, function, and array types. You can declare a `typedef` name for a pointer to a structure or union type before you define the structure or union type, as long as the definition has the same visibility as the declaration.

## Examples

One use of **`typedef`** declarations is to make declarations more uniform and compact. For example:

```
typedef char CHAR;           // Character type.
typedef CHAR * PSTR;        // Pointer to a string (char *).
PSTR strchr( PSTR source, CHAR target );
typedef unsigned long ulong;
ulong ul; // Equivalent to "unsigned long ul;"
```

To use **`typedef`** to specify fundamental and derived types in the same declaration, you can separate declarators with commas. For example:

```
typedef char CHAR, *PSTR;
```

The following example provides the type `DRAWF` for a function returning no value and taking two `int` arguments:

```
typedef void DRAWF( int, int );
```

After the above **`typedef`** statement, the declaration

```
DRAWF box;
```

would be equivalent to the declaration

```
void box( int, int );
```

**`typedef`** is often combined with **`struct`** to declare and name user-defined types:



```
// typedef_specifier2.cpp
#include <stdio.h>

typedef struct mystructtag
{
    int    i;
    double f;
} mystruct;

int main()
{
    mystruct ms;
    ms.i = 10;
    ms.f = 0.99;
    printf_s("%d    %f\n", ms.i, ms.f);
}
```

```
10    0.990000
```

## Re-declaration of typedefs

The **typedef** declaration can be used to redeclare the same name to refer to the same type. For example:

```
// FILE1.H
typedef char CHAR;

// FILE2.H
typedef char CHAR;

// PROG.CPP
#include "file1.h"
#include "file2.h"    // OK
```

The program *PROG.CPP* includes two header files, both of which contain **typedef** declarations for the name **CHAR**. As long as both declarations refer to the same type, such redeclaration is acceptable.

A **typedef** cannot redefine a name that was previously declared as a different type. Therefore, if *FILE2.H* contains

```
// FILE2.H
typedef int CHAR;    // Error
```

the compiler issues an error because of the attempt to redeclare the name **CHAR** to refer to a different type. This extends to constructs such as:

```
typedef char CHAR;
typedef CHAR CHAR;      // OK: redeclared as same type

typedef union REGS      // OK: name REGS redeclared
{
    // by typedef name with the
    struct wordregs x; // same meaning.
    struct byteregs h;
} REGS;
```

## typedefs in C++ vs. C

Use of the **typedef** specifier with class types is supported largely because of the ANSI C practice of declaring unnamed structures in **typedef** declarations. For example, many C programmers use the following:

```
// typedef_with_class_types1.cpp
// compile with: /c
typedef struct {    // Declare an unnamed structure and give it the
                    // typedef name POINT.
    unsigned x;
    unsigned y;
} POINT;
```

The advantage of such a declaration is that it enables declarations like:

```
POINT ptOrigin;
```

instead of:

```
struct point_t ptOrigin;
```

In C++, the difference between **typedef** names and real types (declared with the **class**, **struct**, **union**, and **enum** keywords) is more distinct. Although the C practice of declaring a nameless structure in a **typedef** statement still works, it provides no notational benefits as it does in C.

```
// typedef_with_class_types2.cpp
// compile with: /c /W1
typedef struct {
    int POINT();
    unsigned x;
    unsigned y;
} POINT;
```

The preceding example declares a class named **POINT** using the unnamed class **typedef** syntax. **POINT** is treated as a class name; however, the following restrictions apply to names introduced this way:

- The name (the synonym) cannot appear after a **class**, **struct**, or **union** prefix.
- The name cannot be used as constructor or destructor names within a class declaration.

In summary, this syntax does not provide any mechanism for inheritance, construction, or destruction.

# using declaration

---

The **using** declaration introduces a name into the declarative region in which the using declaration appears.

## Syntax

```
using [typename] nested-name-specifier unqualified-id ;  
using declarator-list ;
```

## Parameters

*nested-name-specifier* A sequence of namespace, class, or enumeration names and scope resolution operators (::), terminated by a scope resolution operator. A single scope resolution operator may be used to introduce a name from the global namespace. The keyword **typename** is optional and may be used to resolve dependent names when introduced into a class template from a base class.

*unqualified-id* An unqualified id-expression, which may be an identifier, an overloaded operator name, a user-defined literal operator or conversion function name, a class destructor name, or a template name and argument list.

*declarator-list* A comma-separated list of [**typename**] *nested-name-specifier unqualified-id* declarators, followed optionally by an ellipsis.

## Remarks

A using declaration introduces an unqualified name as a synonym for an entity declared elsewhere. It allows a single name from a specific namespace to be used without explicit qualification in the declaration region in which it appears. This is in contrast to the [using directive](#), which allows *all* the names in a namespace to be used without qualification. The **using** keyword is also used for [type aliases](#).

## Example

A using declaration can be used in a class definition.

```
// using_declaration1.cpp  
#include <stdio.h>  
class B {  
public:  
    void f(char) {  
        printf_s("In B::f()\n");  
    }  
  
    void g(char) {  
        printf_s("In B::g()\n");  
    }  
};
```

```

class D : B {
public:
    using B::f;    // B::f(char) is now visible as D::f(char)
    using B::g;    // B::g(char) is now visible as D::g(char)
    void f(int) {
        printf_s("In D::f()\n");
        f('c');    // Invokes B::f(char) instead of recursing
    }

    void g(int) {
        printf_s("In D::g()\n");
        g('c');    // Invokes B::g(char) instead of recursing
    }
};

int main() {
    D myD;
    myD.f(1);
    myD.g('a');
}

```

```

In D::f()
In B::f()
In B::g()

```

## Example

When used to declare a member, a using declaration must refer to a member of a base class.

```

// using_declaration2.cpp
#include <stdio.h>

class B {
public:
    void f(char) {
        printf_s("In B::f()\n");
    }

    void g(char) {
        printf_s("In B::g()\n");
    }
};

class C {
public:
    int g();
};

```

```

class D2 : public B {
public:
    using B::f;    // ok: B is a base of D2
    // using C::g;  // error: C isn't a base of D2
};

int main() {
    D2 MyD2;
    MyD2.f('a');
}

```

In B::f()

## Example

Members declared by using a using declaration can be referenced by using explicit qualification. The `::` prefix refers to the global namespace.

```

// using_declaration3.cpp
#include <stdio.h>

void f() {
    printf_s("In f\n");
}

namespace A {
    void g() {
        printf_s("In A::g\n");
    }
}

namespace X {
    using ::f;    // global f is also visible as X::f
    using A::g;   // A's g is now visible as X::g
}

void h() {
    printf_s("In h\n");
    X::f();    // calls ::f
    X::g();    // calls A::g
}

int main() {
    h();
}

```

```
In h
In f
In A::g
```

## Example

When a using declaration is made, the synonym created by the declaration refers only to definitions that are valid at the point of the using declaration. Definitions added to a namespace after the using declaration are not valid synonyms.

A name defined by a **using** declaration is an alias for its original name. It does not affect the type, linkage or other attributes of the original declaration.

```
// post_declaration_namespace_additions.cpp
// compile with: /c
namespace A {
    void f(int) {}
}

using A::f;    // f is a synonym for A::f(int) only

namespace A {
    void f(char) {}
}

void f() {
    f('a');    // refers to A::f(int), even though A::f(char) exists
}

void b() {
    using A::f;    // refers to A::f(int) AND A::f(char)
    f('a');    // calls A::f(char);
}
```

## Example

With respect to functions in namespaces, if a set of local declarations and using declarations for a single name are given in a declarative region, they must all refer to the same entity, or they must all refer to functions.

```
// functions_in_namespaces1.cpp
// C2874 expected
namespace B {
    int i;
    void f(int);
    void f(double);
}
```

```

void g() {
    int i;
    using B::i;    // error: i declared twice
    void f(char);
    using B::f;    // ok: each f is a function
}

```

In the example above, the `using B::i` statement causes a second `int i` to be declared in the `g()` function. The `using B::f` statement does not conflict with the `f(char)` function because the function names introduced by `B::f` have different parameter types.

## Example

A local function declaration cannot have the same name and type as a function introduced by using declaration. For example:

```

// functions_in_namespaces2.cpp
// C2668 expected
namespace B {
    void f(int);
    void f(double);
}

namespace C {
    void f(int);
    void f(double);
    void f(char);
}

void h() {
    using B::f;           // introduces B::f(int) and B::f(double)
    using C::f;           // C::f(int), C::f(double), and C::f(char)
    f('h');              // calls C::f(char)
    f(1);                 // C2668 ambiguous: B::f(int) or C::f(int)?
    void f(int);          // C2883 conflicts with B::f(int) and C::f(int)
}

```

## Example

With respect to inheritance, when a using declaration introduces a name from a base class into a derived class scope, member functions in the derived class override virtual member functions with the same name and argument types in the base class.

```

// using_declaration_inheritance1.cpp
#include <stdio.h>
struct B {
    virtual void f(int) {
        printf_s("In B::f(int)\n");
    }
}

```



```

    }

    virtual void f(char) {
        printf_s("In B::f(char)\n");
    }

    void g(int) {
        printf_s("In B::g\n");
    }

    void h(int);
};

struct D : B {
    using B::f;
    void f(int) {    // ok: D::f(int) overrides B::f(int)
        printf_s("In D::f(int)\n");
    }

    using B::g;
    void g(char) {    // ok: there is no B::g(char)
        printf_s("In D::g(char)\n");
    }

    using B::h;
    void h(int) {}    // Note: D::h(int) hides non-virtual B::h(int)
};

void f(D* pd) {
    pd->f(1);        // calls D::f(int)
    pd->f('a');       // calls B::f(char)
    pd->g(1);         // calls B::g(int)
    pd->g('a');       // calls D::g(char)
}

int main() {
    D * myd = new D();
    f(myd);
}

```

```

In D::f(int)
In B::f(char)
In B::g
In D::g(char)

```

## Example

All instances of a name mentioned in a using declaration must be accessible. In particular, if a derived class uses a using declaration to access a member of a base class, the member name must be accessible. If the name is that of an overloaded member function, then all functions named must be accessible.

For more information on accessibility of members, see [Member-Access Control](#).

```
// using_declaration_inheritance2.cpp
// C2876 expected
class A {
private:
    void f(char);
public:
    void f(int);
protected:
    void g();
};

class B : public A {
    using A::f;    // C2876: A::f(char) is inaccessible
public:
    using A::g;    // B::g is a public synonym for A::g
};
```

## See also

[Namespaces](#)

[Keywords](#)

# volatile (C++)

---

A type qualifier that you can use to declare that an object can be modified in the program by the hardware.

## Syntax

```
volatile declarator ;
```

## Remarks

You can use the [/volatile](#) compiler switch to modify how the compiler interprets this keyword.

Visual Studio interprets the **volatile** keyword differently depending on the target architecture. For ARM, if no **/volatile** compiler option is specified, the compiler performs as if **/volatile:iso** were specified. For architectures other than ARM, if no **/volatile** compiler option is specified, the compiler performs as if **/volatile:ms** were specified; therefore, for architectures other than ARM we strongly recommend that you specify **/volatile:iso**, and use explicit synchronization primitives and compiler intrinsics when you are dealing with memory that is shared across threads.

You can use the **volatile** qualifier to provide access to memory locations that are used by asynchronous processes such as interrupt handlers.

When **volatile** is used on a variable that also has the [\\_\\_restrict](#) keyword, **volatile** takes precedence.

If a **struct** member is marked as **volatile**, then **volatile** is propagated to the whole structure. If a structure does not have a length that can be copied on the current architecture by using one instruction, **volatile** may be completely lost on that structure.

The **volatile** keyword may have no effect on a field if one of the following conditions is true:

- The length of the volatile field exceeds the maximum size that can be copied on the current architecture by using one instruction.
- The length of the outermost containing **struct**—or if it's a member of a possibly nested **struct**—exceeds the maximum size that can be copied on the current architecture by using one instruction.

Although the processor does not reorder un-cacheable memory accesses, un-cacheable variables must be marked as **volatile** to guarantee that the compiler does not reorder the memory accesses.

Objects that are declared as **volatile** are not used in certain optimizations because their values can change at any time. The system always reads the current value of a volatile object when it is requested, even if a previous instruction asked for a value from the same object. Also, the value of the object is written immediately on assignment.

## ISO Compliant

If you are familiar with the C# `volatile` keyword, or familiar with the behavior of **`volatile`** in earlier versions of the Microsoft C++ compiler (MSVC), be aware that the C++11 ISO Standard **`volatile`** keyword is different and is supported in MSVC when the `/volatile:iso` compiler option is specified. (For ARM, it's specified by default). The **`volatile`** keyword in C++11 ISO Standard code is to be used only for hardware access; do not use it for inter-thread communication. For inter-thread communication, use mechanisms such as `std::atomic<T>` from the [C++ Standard Library](#).

## End of ISO Compliant

### Microsoft Specific

When the `/volatile:ms` compiler option is used—by default when architectures other than ARM are targeted—the compiler generates extra code to maintain ordering among references to volatile objects in addition to maintaining ordering to references to other global objects. In particular:

- A write to a volatile object (also known as volatile write) has Release semantics; that is, a reference to a global or static object that occurs before a write to a volatile object in the instruction sequence will occur before that volatile write in the compiled binary.
- A read of a volatile object (also known as volatile read) has Acquire semantics; that is, a reference to a global or static object that occurs after a read of volatile memory in the instruction sequence will occur after that volatile read in the compiled binary.

This enables volatile objects to be used for memory locks and releases in multithreaded applications.

[!NOTE] When it relies on the enhanced guarantee that's provided when the `/volatile:ms` compiler option is used, the code is non-portable.

### END Microsoft Specific

## See also

[Keywords](#)

[const](#)

[const and volatile Pointers](#)

# decltype (C++)

---

The **decltype** type specifier yields the type of a specified expression. The **decltype** type specifier, together with the [auto keyword](#), is useful primarily to developers who write template libraries. Use **auto** and **decltype** to declare a template function whose return type depends on the types of its template arguments. Or, use **auto** and **decltype** to declare a template function that wraps a call to another function, and then returns the return type of the wrapped function.

## Syntax

```
decltype( expression )
```

## Parameters

Parameter	Description
<i>expression</i>	An expression. For more information, see <a href="#">Expressions</a> .

## Return Value

The type of the *expression* parameter.

## Remarks

The **decltype** type specifier is supported in Visual Studio 2010 or later versions, and can be used with native or managed code. **decltype(auto)** (C++ 14) is supported in Visual Studio 2015 and later.

The compiler uses the following rules to determine the type of the *expression* parameter.

- If the *expression* parameter is an identifier or a [class member access](#), **decltype(expression)** is the type of the entity named by *expression*. If there is no such entity or the *expression* parameter names a set of overloaded functions, the compiler yields an error message.
- If the *expression* parameter is a call to a function or an overloaded operator function, **decltype(expression)** is the return type of the function. Parentheses around an overloaded operator are ignored.
- If the *expression* parameter is an [rvalue](#), **decltype(expression)** is the type of *expression*. If the *expression* parameter is an [lvalue](#), **decltype(expression)** is an [lvalue reference](#) to the type of *expression*.

The following code example demonstrates some uses of the **decltype** type specifier. First, assume that you have coded the following statements.

```
int var;  
const int&& fx();
```

```
struct A { double x; }
const A* a = new A();
```

Next, examine the types that are returned by the four **decltype** statements in the following table.

Statement	Type	Notes
<code>decltype(fx());</code>	<code>const int&amp;&amp;</code>	An <a href="#">rvalue reference</a> to a <b>const int</b> .
<code>decltype(var);</code>	<code>int</code>	The type of variable <code>var</code> .
<code>decltype(a-&gt;x);</code>	<code>double</code>	The type of the member access.
<code>decltype((a-&gt;x));</code>	<code>const double&amp;</code>	The inner parentheses cause the statement to be evaluated as an expression instead of a member access. And because <code>a</code> is declared as a <b>const</b> pointer, the type is a reference to <b>const double</b> .

## Decltype and Auto

In C++14, you can use `decltype(auto)` with no trailing return type to declare a template function whose return type depends on the types of its template arguments.

In C++11, you can use the **decltype** type specifier on a trailing return type, together with the **auto** keyword, to declare a template function whose return type depends on the types of its template arguments. For example, consider the following code example in which the return type of the template function depends on the types of the template arguments. In the code example, the *UNKNOWN* placeholder indicates that the return type cannot be specified.

```
template<typename T, typename U>
UNKNOWN func(T&& t, U&& u){ return t + u; };
```

The introduction of the **decltype** type specifier enables a developer to obtain the type of the expression that the template function returns. Use the *alternative function declaration syntax* that is shown later, the **auto** keyword, and the **decltype** type specifier to declare a *late-specified* return type. The late-specified return type is determined when the declaration is compiled, instead of when it is coded.

The following prototype illustrates the syntax of an alternative function declaration. Note that the **const** and **volatile** qualifiers, and the **throw exception specification** are optional. The *function\_body* placeholder represents a compound statement that specifies what the function does. As a best coding practice, the *expression* placeholder in the **decltype** statement should match the expression specified by the **return** statement, if any, in the *function\_body*.

```
auto function_name ( parametersopt ) constopt volatileopt -> decltype( expression ) throwopt { function_body
};
```

In the following code example, the late-specified return type of the `myFunc` template function is determined by the types of the `t` and `u` template arguments. As a best coding practice, the code example also uses *rvalue*

references and the `forward` function template, which support *perfect forwarding*. For more information, see [Rvalue Reference Declarator: &&](#).

```
//C++11
template<typename T, typename U>
auto myFunc(T&& t, U&& u) -> decltype (forward<T>(t) + forward<U>(u))
    { return forward<T>(t) + forward<U>(u); };

//C++14
template<typename T, typename U>
decltype(auto) myFunc(T&& t, U&& u)
    { return forward<T>(t) + forward<U>(u); };
```

## Decltype and Forwarding Functions (C++11)

Forwarding functions wrap calls to other functions. Consider a function template that forwards its arguments, or the results of an expression that involves those arguments, to another function. Furthermore, the forwarding function returns the result of calling the other function. In this scenario, the return type of the forwarding function should be the same as the return type of the wrapped function.

In this scenario, you cannot write an appropriate type expression without the **decltype** type specifier. The **decltype** type specifier enables generic forwarding functions because it does not lose required information about whether a function returns a reference type. For a code example of a forwarding function, see the previous `myFunc` template function example.

### Example

The following code example declares the late-specified return type of template function `Plus()`. The `Plus` function processes its two operands with the **operator+** overload. Consequently, the interpretation of the plus operator (+) and the return type of the `Plus` function depends on the types of the function arguments.

```
// decltype_1.cpp
// compile with: cl /EHsc decltype_1.cpp

#include <iostream>
#include <string>
#include <utility>
#include <iomanip>

using namespace std;

template<typename T1, typename T2>
auto Plus(T1&& t1, T2&& t2) ->
    decltype(forward<T1>(t1) + forward<T2>(t2))
{
    return forward<T1>(t1) + forward<T2>(t2);
}

class X
```

```

{
    friend X operator+(const X& x1, const X& x2)
    {
        return X(x1.m_data + x2.m_data);
    }

public:
    X(int data) : m_data(data) {}
    int Dump() const { return m_data;}
private:
    int m_data;
};

int main()
{
    // Integer
    int i = 4;
    cout <<
        "Plus(i, 9) = " <<
        Plus(i, 9) << endl;

    // Floating point
    float dx = 4.0;
    float dy = 9.5;
    cout <<
        setprecision(3) <<
        "Plus(dx, dy) = " <<
        Plus(dx, dy) << endl;

    // String
    string hello = "Hello, ";
    string world = "world!";
    cout << Plus(hello, world) << endl;

    // Custom type
    X x1(20);
    X x2(22);
    X x3 = Plus(x1, x2);
    cout <<
        "x3.Dump() = " <<
        x3.Dump() << endl;
}

```

```

Plus(i, 9) = 13
Plus(dx, dy) = 13.5
Hello, world!
x3.Dump() = 42

```

## Example



**Visual Studio 2017 and later:** The compiler parses decltype arguments when the templates are declared rather than instantiated. Consequently, if a non-dependent specialization is found in the decltype argument, it will not be deferred to instantiation-time and will be processed immediately and any resulting errors will be diagnosed at that time.

The following example shows such a compiler error that is raised at the point of declaration:

```
#include <utility>
template <class T, class ReturnT, class... ArgsT> class IsCallable
{
public:
    struct BadType {};
    template <class U>
        static decltype(std::declval<T>()(std::declval<ArgsT>()...)) Test(int);
    //C2064. Should be declval<U>
    template <class U>
        static BadType Test(...);
    static constexpr bool value = std::is_convertible<decltype(Test<T>(0)),
ReturnT>::value;
};

constexpr bool test1 = IsCallable<int(), int>::value;
static_assert(test1, "PASS1");
constexpr bool test2 = !IsCallable<int*, int>::value;
static_assert(test2, "PASS2");
```

## Requirements

Visual Studio 2010 or later versions.

`decltype(auto)` requires Visual Studio 2015 or later.

# Attributes in C++

---

The C++ Standard defines a set of attributes and also allows compiler vendors to define their own attributes (within a vendor-specific namespace), but compilers are required to recognize only those attributes defined in the standard.

In some cases, standard attributes overlap with compiler-specific `declspec` parameters. In Visual C++, you can use the `[[deprecated]]` attribute instead of using `declspec(deprecated)` and the attribute will be recognized by any conforming compiler. For all other `declspec` parameters such as `dllimport` and `dllexport`, there is as yet no attribute equivalent so you must continue to use `declspec` syntax. Attributes do not affect the type system, and they don't change the meaning of a program. Compilers ignore attribute values they don't recognize.

**Visual Studio 2017 version 15.3 and later** (available with [/std:c++17](#)): In the scope of an attribute list, you can specify the namespace for all names with a single **using** introducer:

```
void g() {
    [[using rpr: kernel, target(cpu,gpu)]] // equivalent to [[ rpr::kernel,
    rpr::target(cpu,gpu) ]]
    do task();
}
```

## C++ Standard Attributes

In C++11, attributes provide a standardized way to annotate C++ constructs (including but not limited to classes, functions, variables, and blocks) with additional information that may or may not be vendor-specific. A compiler can use this information to generate informational messages, or to apply special logic when compiling the attributed code. The compiler ignores any attributes that it does not recognize, which means that you cannot define your own custom attributes using this syntax. Attributes are enclosed by double square brackets:

```
[[deprecated]]
void Foo(int);
```

Attributes represent a standardized alternative to vendor-specific extensions such as `#pragma` directives, `__declspec()` (Visual C++), or `__attribute__` (GNU). However, you will still need to use the vendor-specific constructs for most purposes. The standard currently specifies the following attributes that a conforming compiler should recognize:

- `[[noreturn]]` Specifies that a function never returns; in other words it always throws an exception. The compiler can adjust its compilation rules for `[[noreturn]]` entities.
- `[[carries_dependency]]` Specifies that the function propagates data dependency ordering with respect to thread synchronization. The attribute can be applied to one or more parameters, to specify

that the passed-in argument carries a dependency into the function body. The attribute can be applied to the function itself, to specify that the return value carries a dependency out of the function. The compiler can use this information to generate more efficient code.

- **[[deprecated]] Visual Studio 2015 and later:** Specifies that a function is not intended to be used, and might not exist in future versions of a library interface. The compiler can use this to generate an informational message when client code attempts to call the function. Can be applied to declaration of a class, a typedef-name, a variable, a non-static data member, a function, a namespace, an enumeration, an enumerator, or a template specialization.
- **[[fallthrough]] Visual Studio 2017 and later:** (available with `/std:c++17`) The **[[fallthrough]]** attribute can be used in the context of `switch` statements as a hint to the compiler (or anyone reading the code) that the fallthrough behavior is intended. The Microsoft C++ compiler currently does not warn on fallthrough behavior, so this attribute has no effect compiler behavior.
- **[[nodiscard]] Visual Studio 2017 version 15.3 and later:** (available with `/std:c++17`) Specifies that a function's return value is not intended to be discarded. Raises warning C4834, as shown in this example:

```
[[nodiscard]]
int foo(int i) { return i * i; }

int main()
{
    foo(42); //warning C4834: discarding return value of function with
    'nodiscard' attribute
    return 0;
}
```

- **[[maybe\_unused]] Visual Studio 2017 version 15.3 and later:** (available with `/std:c++17`) Specifies that a variable, function, class, typedef, non-static data member, enum, or template specialization may intentionally not be used. The compiler does not warn when an entity marked **[[maybe\_unused]]** is not used. An entity that is declared without the attribute can later be redeclared with the attribute and vice versa. An entity is considered marked after its first declaration that is marked is analyzed, and for the remainder of translation of the current translation unit.

## Microsoft-specific attributes

- **[[gsl::suppress(rules)]]** This Microsoft-specific attribute is used for suppressing warnings from checkers that enforce [Guidelines Support Library \(GSL\)](#) rules in code. For example, consider this code snippet:

```
int main()
{
    int arr[10]; // GSL warning C26494 will be fired
    int* p = arr; // GSL warning C26485 will be fired
    [[gsl::suppress(bounds.1)]] // This attribute suppresses Bounds rule #1
    {
```

```
int* q = p + 1; // GSL warning C26481 suppressed
p = q--; // GSL warning C26481 suppressed
    }
}
```

The example raises these warnings:

- 26494 (Type Rule 5: Always initialize an object.)
- 26485 (Bounds Rule 3: No array to pointer decay.)
- 26481 (Bounds Rule 1: Don't use pointer arithmetic. Use span instead.)

The first two warnings fire when you compile this code with the CppCoreCheck code analysis tool installed and activated. But the third warning doesn't fire because of the attribute. You can suppress the entire bounds profile by writing `[[gsl::suppress(bounds)]]` without including a specific rule number. The C++ Core Guidelines are designed to help you write better and safer code. The suppress attribute makes it easy to turn off the warnings when they are not wanted.

# C++ Built-in Operators, Precedence and Associativity

---

The C++ language includes all C operators and adds several new operators. Operators specify an evaluation to be performed on one or more operands.

Operator *precedence* specifies the order of operations in expressions that contain more than one operator. Operator *associativity* specifies whether, in an expression that contains multiple operators with the same precedence, an operand is grouped with the one on its left or the one on its right. The following table shows the precedence and associativity of C++ operators (from highest to lowest precedence). Operators with the same precedence number have equal precedence unless another relationship is explicitly forced by parentheses.

## C++ Operator Precedence and Associativity

Operator Description	Operator
<b>Group 1 precedence, no associativity</b>	
Scope resolution	::
<b>Group 2 precedence, left to right associativity</b>	
Member selection (object or pointer)	. or ->
Array subscript	[]
Function call	()
Postfix increment	++
Postfix decrement	--
Type name	typeid
Constant type conversion	const_cast
Dynamic type conversion	dynamic_cast
Reinterpreted type conversion	reinterpret_cast
Static type conversion	static_cast
<b>Group 3 precedence, right to left associativity</b>	
Size of object or type	sizeof
Prefix increment	++
Prefix decrement	--
One's complement	~
Logical not	!
Unary negation	-

Operator Description	Operator
Unary plus	+
Address-of	&
Indirection	*
Create object	new
Destroy object	delete
Cast	()
<b>Group 4 precedence, left to right associativity</b>	
Pointer-to-member (objects or pointers)	.* or ->*
<b>Group 5 precedence, left to right associativity</b>	
Multiplication	*
Division	/
Modulus	%
<b>Group 6 precedence, left to right associativity</b>	
Addition	+
Subtraction	-
<b>Group 7 precedence, left to right associativity</b>	
Left shift	<<
Right shift	>>
<b>Group 8 precedence, left to right associativity</b>	
Less than	<
Greater than	>
Less than or equal to	<=
Greater than or equal to	>=
<b>Group 9 precedence, left to right associativity</b>	
Equality	==
Inequality	!=
<b>Group 10 precedence left to right associativity</b>	
Bitwise AND	&
<b>Group 11 precedence, left to right associativity</b>	
Bitwise exclusive OR	^

Operator Description	Operator
<b>Group 12 precedence, left to right associativity</b>	
Bitwise inclusive OR	
<b>Group 13 precedence, left to right associativity</b>	
Logical AND	&&
<b>Group 14 precedence, left to right associativity</b>	
Logical OR	
<b>Group 15 precedence, right to left associativity</b>	
Conditional	? :
<b>Group 16 precedence, right to left associativity</b>	
Assignment	=
Multiplication assignment	*=
Division assignment	/=
Modulus assignment	%=
Addition assignment	+=
Subtraction assignment	-=
Left-shift assignment	<<=
Right-shift assignment	>>=
Bitwise AND assignment	&=
Bitwise inclusive OR assignment	=
Bitwise exclusive OR assignment	^=
<b>Group 17 precedence, right to left associativity</b>	
throw expression	throw
<b>Group 18 precedence, left to right associativity</b>	
Comma	,

See also

[Operator Overloading](#)

# \_\_alignof Operator

---

C++11 introduces the **alignof** operator that returns the alignment, in bytes, of the specified type. For maximum portability, you should use the alignof operator instead of the Microsoft-specific \_\_alignof operator.

## Microsoft Specific

Returns a value of type `size_t` that is the alignment requirement of the type.

## Syntax

```
__alignof( type )
```

## Remarks

For example:

Expression	Value
<code>__alignof( char )</code>	1
<code>__alignof( short )</code>	2
<code>__alignof( int )</code>	4
<code>__alignof( __int64 )</code>	8
<code>__alignof( float )</code>	4
<code>__alignof( double )</code>	8
<code>__alignof( char* )</code>	4

The **\_\_alignof** value is the same as the value for `sizeof` for basic types. Consider, however, this example:

```
typedef struct { int a; double b; } S;  
// __alignof(S) == 8
```

In this case, the **\_\_alignof** value is the alignment requirement of the largest element in the structure.

Similarly, for

```
typedef __declspec(align(32)) struct { int a; } S;
```

`__alignof(S)` is equal to 32.



One use for **\_\_alignof** would be as a parameter to one of your own memory-allocation routines. For example, given the following defined structure *S*, you could call a memory-allocation routine named `aligned_malloc` to allocate memory on a particular alignment boundary.

```
typedef __declspec(align(32)) struct { int a; double b; } S;
int n = 50; // array size
S* p = (S*)aligned_malloc(n * sizeof(S), __alignof(S));
```

For compatibility with previous versions, **\_alignof** is a synonym for **\_\_alignof** unless compiler option `/Za` (Disable language extensions) is specified.

For more information on modifying alignment, see:

- [pack](#)
- [align](#)
- [\\_\\_unaligned](#)
- [/Zp \(Struct Member Alignment\)](#)
- [Examples of Structure Alignment](#) (x64 specific)

For more information on differences in alignment in code for x86 and x64, see:

- [Conflicts with the x86 Compiler](#)

## END Microsoft Specific

## See also

[Expressions with Unary Operators](#)

[Keywords](#)

# \_\_uuidof Operator

---

## Microsoft Specific

Retrieves the GUID attached to the expression.

## Syntax

```
__uuidof (expression)
```

## Remarks

The *expression* can be a type name, pointer, reference, or array of that type, a template specialized on these types, or a variable of these types. The argument is valid as long as the compiler can use it to find the attached GUID.

A special case of this intrinsic is when either **0** or NULL is supplied as the argument. In this case, **\_\_uuidof** will return a GUID made up of zeros.

Use this keyword to extract the GUID attached to:

- An object by the [uuid](#) extended attribute.
- A library block created with the [module](#) attribute.

[!NOTE] In a debug build, **\_\_uuidof** always initializes an object dynamically (at runtime). In a release build, **\_\_uuidof** can statically (at compile time) initialize an object.

For compatibility with previous versions, **\_uuidof** is a synonym for **\_\_uuidof** unless compiler option [/Za \(Disable language extensions\)](#) is specified.

## Example

The following code (compiled with ole32.lib) will display the uuid of a library block created with the module attribute:

```
// expre_uuidof.cpp
// compile with: ole32.lib
#include "stdio.h"
#include "windows.h"

[emitidl];
[module(name="MyLib")];
[export]
struct stuff {
    int i;
};
```

```
int main() {
    LPOLESTR lpolestr;
    StringFromCLSID(__uuidof(MyLib), &lpolestr);
    wprintf_s(L"%s", lpolestr);
    CoTaskMemFree(lpolestr);
}
```

## Comments

In cases where the library name is no longer in scope, you can use `__LIBID_` instead of `__uuidof`. For example:

```
StringFromCLSID(__LIBID_, &lpolestr);
```

### END Microsoft Specific

## See also

[Expressions with Unary Operators](#)

[Keywords](#)

# Additive Operators: + and -

---

## Syntax

```
expression + expression  
expression - expression
```

## Remarks

The additive operators are:

- Addition (+)
- Subtraction (-)

These binary operators have left-to-right associativity.

The additive operators take operands of arithmetic or pointer types. The result of the addition (+) operator is the sum of the operands. The result of the subtraction (-) operator is the difference between the operands. If one or both of the operands are pointers, they must be pointers to objects, not to functions. If both operands are pointers, the results are not meaningful unless both are pointers to objects in the same array.

Additive operators take operands of *arithmetic*, *integral*, and *scalar* types. These are defined in the following table.

## Types Used with Additive Operators

Type	Meaning
<i>arithmetic</i>	Integral and floating types are collectively called "arithmetic" types.
<i>integral</i>	Types char and int of all sizes (long, short) and enumerations are "integral" types.
<i>scalar</i>	Scalar operands are operands of either arithmetic or pointer type.

The legal combinations for these operators are:

*arithmetic* + *arithmetic*

*scalar* + *integral*

*integral* + *scalar*

*arithmetic* - *arithmetic*

*scalar* - *scalar*

Note that addition and subtraction are not equivalent operations.

If both operands are of arithmetic type, the conversions covered in [Standard Conversions](#) are applied to the operands, and the result is of the converted type.

## Example

```
// expre_Additive_Operators.cpp
// compile with: /EHsc
#include <iostream>
#define SIZE 5
using namespace std;
int main() {
    int i = 5, j = 10;
    int n[SIZE] = { 0, 1, 2, 3, 4 };
    cout << "5 + 10 = " << i + j << endl
         << "5 - 10 = " << i - j << endl;

    // use pointer arithmetic on array

    cout << "n[3] = " << *( n + 3 ) << endl;
}
```

## Pointer addition

If one of the operands in an addition operation is a pointer to an array of objects, the other must be of integral type. The result is a pointer that is of the same type as the original pointer and that points to another array element. The following code fragment illustrates this concept:

```
short IntArray[10]; // Objects of type short occupy 2 bytes
short *pIntArray = IntArray;

for( int i = 0; i < 10; ++i )
{
    *pIntArray = i;
    cout << *pIntArray << "\n";
    pIntArray = pIntArray + 1;
}
```

Although the integral value 1 is added to `pIntArray`, it does not mean "add 1 to the address"; rather it means "adjust the pointer to point to the next object in the array" that happens to be 2 bytes (or `sizeof( int )`) away.

[!NOTE] Code of the form `pIntArray = pIntArray + 1` is rarely found in C++ programs; to perform an increment, these forms are preferable: `pIntArray++` or `pIntArray += 1`.

## Pointer subtraction

If both operands are pointers, the result of subtraction is the difference (in array elements) between the operands. The subtraction expression yields a signed integral result of type `ptrdiff_t` (defined in the

standard include file `<stddef.h>`).

One of the operands can be of integral type, as long as it is the second operand. The result of the subtraction is of the same type as the original pointer. The value of the subtraction is a pointer to the  $(n - i)$ th array element, where  $n$  is the element pointed to by the original pointer and  $i$  is the integral value of the second operand.

## See also

[Expressions with Binary Operators](#)

[C++ Built-in Operators, Precedence and Associativity](#)

[C Additive Operators](#)

# Address-of Operator: &

---

## Syntax

```
& cast-expression
```

## Remarks

The unary address-of operator (**&**) takes the address of its operand. The operand of the address-of operator can be either a function designator or an l-value that designates an object that is not a bit field.

The address-of operator can only be applied to variables with fundamental, structure, class, or union types that are declared at the file-scope level, or to subscripted array references. In these expressions, a constant expression that does not include the address-of operator can be added to or subtracted from the address-of expression.

When applied to functions or l-values, the result of the expression is a pointer type (an r-value) derived from the type of the operand. For example, if the operand is of type **char**, the result of the expression is of type pointer to **char**. The address-of operator, applied to **const** or **volatile** objects, evaluates to **const type \*** or **volatile type \***, where **type** is the type of the original object.

When the address-of operator is applied to a qualified name, the result depends on whether the *qualified-name* specifies a static member. If so, the result is a pointer to the type specified in the declaration of the member. If the member is not static, the result is a pointer to the member *name* of the class indicated by *qualified-class-name*. (See [Primary Expressions](#) for more about *qualified-class-name*.) The following code fragment shows how the result differs, depending on whether the member is static:

```
// expre_Address_Of_Operator.cpp
// C2440 expected
class PTM {
public:
    int iValue;
    static float fValue;
};

int main() {
    int    PTM::*piValue = &PTM::iValue; // OK: non-static
    float  PTM::*pfValue = &PTM::fValue; // C2440 error: static
    float *spfValue      = &PTM::fValue; // OK
}
```

In this example, the expression **&PTM::fValue** yields type **float \*** instead of type **float PTM::\*** because **fValue** is a static member.

The address of an overloaded function can be taken only when it is clear which version of the function is being referenced. See [Function Overloading](#) for information about how to obtain the address of a particular overloaded function.

Applying the address-of operator to a reference type gives the same result as applying the operator to the object to which the reference is bound. For example:

## Example

```
// expre_Address_Of_Operator2.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;
int main() {
    double d;          // Define an object of type double.
    double& rd = d;    // Define a reference to the object.

    // Obtain and compare their addresses
    if( &d == &rd )
        cout << "&d equals &rd" << endl;
}
```

## Output

```
&d equals &rd
```

The following example uses the address-of operator to pass a pointer argument to a function:

```
// expre_Address_Of_Operator3.cpp
// compile with: /EHsc
// Demonstrate address-of operator &

#include <iostream>
using namespace std;

// Function argument is pointer to type int
int square( int *n ) {
    return (*n) * (*n);
}

int main() {
    int mynum = 5;
    cout << square( &mynum ) << endl;    // pass address of int
}
```

## Output



## See also

[Expressions with Unary Operators](#)

[C++ Built-in Operators, Precedence and Associativity](#)

[Lvalue Reference Declarator: &](#)

[Indirection and Address-of Operators](#)

# Assignment Operators

---

## Syntax

*expression assignment-operator expression*

*assignment-operator* : one of

`= *= /= %= += -= <<= >>= &= ^= |=`

## Remarks

Assignment operators store a value in the object designated by the left operand. There are two kinds of assignment operations:

1. *simple assignment*, in which the value of the second operand is stored in the object specified by the first operand.
2. *compound assignment*, in which an arithmetic, shift, or bitwise operation is performed prior to storing the result.

All assignment operators in the following table except the `=` operator are compound assignment operators.

### Assignment operators table

Operator	Meaning
<code>=</code>	Store the value of the second operand in the object specified by the first operand (simple assignment).
<code>*=</code>	Multiply the value of the first operand by the value of the second operand; store the result in the object specified by the first operand.
<code>/=</code>	Divide the value of the first operand by the value of the second operand; store the result in the object specified by the first operand.
<code>%=</code>	Take modulus of the first operand specified by the value of the second operand; store the result in the object specified by the first operand.
<code>+=</code>	Add the value of the second operand to the value of the first operand; store the result in the object specified by the first operand.
<code>-=</code>	Subtract the value of the second operand from the value of the first operand; store the result in the object specified by the first operand.
<code>&lt;&lt;=</code>	Shift the value of the first operand left the number of bits specified by the value of the second operand; store the result in the object specified by the first operand.
<code>&gt;&gt;=</code>	Shift the value of the first operand right the number of bits specified by the value of the second operand; store the result in the object specified by the first operand.

Operator	Meaning
<b>&amp;=</b>	Obtain the bitwise AND of the first and second operands; store the result in the object specified by the first operand.
<b>^=</b>	Obtain the bitwise exclusive OR of the first and second operands; store the result in the object specified by the first operand.
<b> =</b>	Obtain the bitwise inclusive OR of the first and second operands; store the result in the object specified by the first operand.

## Operator keywords

Three of the compound assignment operators have text equivalents. They are:

Operator	Equivalent
<b>&amp;=</b>	<code>and_eq</code>
<b> =</b>	<code>or_eq</code>
<b>^=</b>	<code>xor_eq</code>

There are two ways to access these operator keywords in your programs: include the header file `iso646.h`, or compile with the `/Za` (Disable language extensions) compiler option.

## Example

```
// expre_Assignment_Operators.cpp
// compile with: /EHsc
// Demonstrate assignment operators
#include <iostream>
using namespace std;
int main() {
    int a = 3, b = 6, c = 10, d = 0xAAAA, e = 0x5555;

    a += b;      // a is 9
    b %= a;      // b is 6
    c >>= 1;     // c is 5
    d |= e;      // Bitwise--d is 0xFFFF

    cout << "a = 3, b = 6, c = 10, d = 0xAAAA, e = 0x5555" << endl
         << "a += b yields " << a << endl
         << "b %= a yields " << b << endl
         << "c >>= 1 yields " << c << endl
         << "d |= e yields " << hex << d << endl;
}
```

## Simple assignment

The simple assignment operator (=) causes the value of the second operand to be stored in the object specified by the first operand. If both objects are of arithmetic types, the right operand is converted to the type of the left, prior to storing the value.

Objects of **const** and **volatile** types can be assigned to l-values of types that are just **volatile** or that are neither **const** nor **volatile**.

Assignment to objects of class type (struct, union, and class types) is performed by a function named `operator=`. The default behavior of this operator function is to perform a bitwise copy; however, this behavior can be modified using overloaded operators. See [Operator overloading](#) for more information. In addition, class types can have *copy assignment* and *move assignment* operators. For more information, see [Copy constructors and copy assignment operators](#) and [Move constructors and move assignment operators](#).

An object of any unambiguously derived class from a given base class can be assigned to an object of the base class. The reverse is not true because there is an implicit conversion from derived class to base class but not from base class to derived class. For example:

```
// expre_SimpleAssignment.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;
class ABase
{
public:
    ABase() { cout << "constructing ABase\n"; }
};

class ADerived : public ABase
{
public:
    ADerived() { cout << "constructing ADerived\n"; }
};

int main()
{
    ABase aBase;
    ADerived aDerived;

    aBase = aDerived; // OK
    aDerived = aBase; // C2679
}
```

Assignments to reference types behave as if the assignment were being made to the object to which the reference points.

For class-type objects, assignment is different from initialization. To illustrate how different assignment and initialization can be, consider the code

```
UserType1 A;  
UserType2 B = A;
```

The preceding code shows an initializer; it calls the constructor for `UserType2` that takes an argument of type `UserType1`. Given the code

```
UserType1 A;  
UserType2 B;  
  
B = A;
```

the assignment statement

```
B = A;
```

can have one of the following effects:

- Call the function `operator=` for `UserType2`, provided `operator=` is provided with a `UserType1` argument.
- Call the explicit conversion function `UserType1::operator UserType2`, if such a function exists.
- Call a constructor `UserType2::UserType2`, provided such a constructor exists, that takes a `UserType1` argument and copies the result.

## Compound assignment

The compound assignment operators, shown in the [Assignment operators table](#), are specified in the form `e1 op= e2`, where `e1` is a modifiable l-value not of **const** type and `e2` is one of the following:

- An arithmetic type
- A pointer, if `op` is `+` or `-`

The `e1 op= e2` form behaves as `e1 = e1 op e2`, but `e1` is evaluated only once.

Compound assignment to an enumerated type generates an error message. If the left operand is of a pointer type, the right operand must be of a pointer type or it must be a constant expression that evaluates to 0. If the left operand is of an integral type, the right operand must not be of a pointer type.

## Result of assignment operators

The assignment operators return the value of the object specified by the left operand after the assignment. The resultant type is the type of the left operand. The result of an assignment expression is always an l-value. These operators have right-to-left associativity. The left operand must be a modifiable l-value.

In ANSI C, the result of an assignment expression is not an l-value. Therefore, the legal C++ expression `(a += b) += c` is illegal in C.

## See also

[Expressions with Binary Operators](#)

[C++ Built-in Operators, Precedence and Associativity](#)

[C Assignment Operators](#)

# Bitwise AND Operator: &

---

## Syntax

```
expression & expression
```

## Remarks

The expressions may be other and-expressions, or (subject to the type restrictions mentioned below) equality expressions, relational expressions, additive expressions, multiplicative expressions, pointer to member expressions, cast expressions, unary expressions, postfix expressions, or primary expressions.

The bitwise AND operator (**&**) compares each bit of the first operand to the corresponding bit of the second operand. If both bits are 1, the corresponding result bit is set to 1. Otherwise, the corresponding result bit is set to 0.

Both operands to the bitwise AND operator must be of integral types. The usual arithmetic conversions covered in [Standard Conversions](#), are applied to the operands.

## Operator Keyword for &

The **bitand** operator is the text equivalent of **&**. There are two ways to access the **bitand** operator in your programs: include the header file `iso646.h`, or compile with the `/Za` (Disable language extensions) compiler option.

## Example

```
// expre_Bitwise_AND_Operator.cpp
// compile with: /EHsc
// Demonstrate bitwise AND
#include <iostream>
using namespace std;
int main() {
    unsigned short a = 0xFFFF;    // pattern 1111 ...
    unsigned short b = 0xAAAA;    // pattern 1010 ...

    cout << hex << ( a & b ) << endl; // prints "aaaa", pattern 1010 ...
}
```

## See also

[C++ Built-in Operators, Precedence and Associativity](#)

[C++ Built-in Operators, Precedence and Associativity](#)

[C Bitwise Operators](#)

# Bitwise Exclusive OR Operator: ^

---

## Syntax

```
expression ^ expression
```

## Remarks

The bitwise exclusive OR operator (^) compares each bit of its first operand to the corresponding bit of its second operand. If one bit is 0 and the other bit is 1, the corresponding result bit is set to 1. Otherwise, the corresponding result bit is set to 0.

Both operands to the bitwise exclusive OR operator must be of integral types. The usual arithmetic conversions covered in [Standard Conversions](#) are applied to the operands.

## Operator Keyword for ^

The **xor** operator is the text equivalent of ^. There are two ways to access the **xor** operator in your programs: include the header file `iso646.h`, or compile with the `/Za` (Disable language extensions) compiler option.

## Example

```
// expre_Bitwise_Exclusive_OR_Operator.cpp
// compile with: /EHsc
// Demonstrate bitwise exclusive OR
#include <iostream>
using namespace std;
int main() {
    unsigned short a = 0x5555;    // pattern 0101 ...
    unsigned short b = 0xFFFF;    // pattern 1111 ...

    cout << hex << ( a ^ b ) << endl;    // prints "aaaa" pattern 1010 ...
}
```

## See also

[C++ Built-in Operators, Precedence and Associativity](#)



# Bitwise inclusive OR operator: |

---

## Syntax

```
expression1 | expression2
```

## Remarks

The bitwise inclusive OR operator (|) compares each bit of its first operand to the corresponding bit of its second operand. If either bit is 1, the corresponding result bit is set to 1. Otherwise, the corresponding result bit is set to 0.

Both operands to the bitwise inclusive OR operator must be of integral types. The usual arithmetic conversions covered in [Standard Conversions](#) are applied to the operands.

## Operator keyword for |

The **bitor** operator is the text equivalent of |. There are two ways to access the **bitor** operator in your programs: include the header file <iso646.h>, or compile with the [/Za](#) (Disable language extensions) compiler option.

## Example

```
// expre_Bitwise_Inclusive_OR_Operator.cpp
// compile with: /EHsc
// Demonstrate bitwise inclusive OR
#include <iostream>
using namespace std;

int main() {
    unsigned short a = 0x5555;    // pattern 0101 ...
    unsigned short b = 0xAAAA;    // pattern 1010 ...

    cout << hex << ( a | b ) << endl; // prints "ffff" pattern 1111 ...
}
```

## See also

[C++ Built-in Operators, Precedence and Associativity](#)

[C Bitwise Operators](#)

# Cast Operator: ()

---

A type cast provides a method for explicit conversion of the type of an object in a specific situation.

## Syntax

```
unary-expression ( type-name ) cast-expression
```

## Remarks

Any unary expression is considered a cast expression.

The compiler treats *cast-expression* as type *type-name* after a type cast has been made. Casts can be used to convert objects of any scalar type to or from any other scalar type. Explicit type casts are constrained by the same rules that determine the effects of implicit conversions. Additional restraints on casts may result from the actual sizes or representation of specific types.

## Example

```
// expre_CastOperator.cpp
// compile with: /EHsc
// Demonstrate cast operator
#include <iostream>

using namespace std;

int main()
{
    double x = 3.1;
    int i;
    cout << "x = " << x << endl;
    i = (int)x;    // assign i the integer part of x
    cout << "i = " << i << endl;
}
```

## Example

```
// expre_CastOperator2.cpp
// The following sample shows how to define and use a cast operator.
#include <string.h>
#include <stdio.h>

class CountedAnsiString
{
```

```

public:
    // Assume source is not null terminated
    CountedAnsiString(const char *pStr, size_t nSize) :
        m_nSize(nSize)
    {
        m_pStr = new char[sizeofBuffer];

        strncpy_s(m_pStr, sizeofBuffer, pStr, m_nSize);
        memset(&m_pStr[m_nSize], '!', 9); // for demonstration purposes.
    }

    // Various string-like methods...

    const char *GetRawBytes() const
    {
        return(m_pStr);
    }

    //
    // operator to cast to a const char *
    //
    operator const char *()
    {
        m_pStr[m_nSize] = '\\0';
        return(m_pStr);
    }

    enum
    {
        sizeofBuffer = 20
    } size;

private:
    char *m_pStr;
    const size_t m_nSize;
};

int main()
{
    const char *kStr = "Excitingggg";
    CountedAnsiString myStr(kStr, 8);

    const char *pRaw = myStr.GetRawBytes();
    printf_s("RawBytes truncated to 10 chars:  %.10s\\n", pRaw);

    const char *pCast = myStr; // or (const char *)myStr;
    printf_s("Casted Bytes:  %s\\n", pCast);

    puts("Note that the cast changed the raw internal string");
    printf_s("Raw Bytes after cast:  %s\\n", pRaw);
}

```

```
RawBytes truncated to 10 chars:  Exciting!!  
Casted Bytes:  Exciting  
Note that the cast changed the raw internal string  
Raw Bytes after cast:  Exciting
```

## See also

[Expressions with Unary Operators](#)

[C++ Built-in Operators, Precedence and Associativity](#)

[Explicit Type Conversion Operator: \(\)](#)

[Casting Operators](#)

[Cast Operators](#)

# Comma Operator: ,

---

Allows grouping two statements where one is expected.

## Syntax

```
expression , expression
```

## Remarks

The comma operator has left-to-right associativity. Two expressions separated by a comma are evaluated left to right. The left operand is always evaluated, and all side effects are completed before the right operand is evaluated.

Commas can be used as separators in some contexts, such as function argument lists. Do not confuse the use of the comma as a separator with its use as an operator; the two uses are completely different.

Consider the expression `e1, e2`. The type and value of the expression are the type and value of `e2`; the result of evaluating `e1` is discarded. The result is an l-value if the right operand is an l-value.

Where the comma is normally used as a separator (for example in actual arguments to functions or aggregate initializers), the comma operator and its operands must be enclosed in parentheses. For example:

```
func_one( x, y + 2, z );  
func_two( (x--, y + 2), z );
```

In the function call to `func_one` above, three arguments, separated by commas, are passed: `x`, `y + 2`, and `z`. In the function call to `func_two`, parentheses force the compiler to interpret the first comma as the sequential-evaluation operator. This function call passes two arguments to `func_two`. The first argument is the result of the sequential-evaluation operation `(x--, y + 2)`, which has the value and type of the expression `y + 2`; the second argument is `z`.

## Example

```
// cpp_comma_operator.cpp  
#include <stdio.h>  
int main () {  
    int i = 10, b = 20, c = 30;  
    i = b, c;  
    printf("%i\n", i);  
  
    i = (b, c);  
    printf("%i\n", i);  
}
```

20  
30

## See also

[Expressions with Binary Operators](#)

[C++ Built-in Operators, Precedence and Associativity](#)

[Sequential-Evaluation Operator](#)

# Conditional Operator: ? :

---

## Syntax

```
expression ? expression : expression
```

## Remarks

The conditional operator (**? :**) is a ternary operator (it takes three operands). The conditional operator works as follows:

- The first operand is implicitly converted to **bool**. It is evaluated and all side effects are completed before continuing.
- If the first operand evaluates to **true** (1), the second operand is evaluated.
- If the first operand evaluates to **false** (0), the third operand is evaluated.

The result of the conditional operator is the result of whichever operand is evaluated — the second or the third. Only one of the last two operands is evaluated in a conditional expression.

Conditional expressions have right-to-left associativity. The first operand must be of integral or pointer type. The following rules apply to the second and third operands:

- If both operands are of the same type, the result is of that type.
- If both operands are of arithmetic or enumeration types, the usual arithmetic conversions (covered in [Standard Conversions](#)) are performed to convert them to a common type.
- If both operands are of pointer types or if one is a pointer type and the other is a constant expression that evaluates to 0, pointer conversions are performed to convert them to a common type.
- If both operands are of reference types, reference conversions are performed to convert them to a common type.
- If both operands are of type void, the common type is type void.
- If both operands are of the same user-defined type, the common type is that type.
- If the operands have different types and at least one of the operands has user-defined type then the language rules are used to determine the common type. (See warning below.)

Any combinations of second and third operands not in the preceding list are illegal. The type of the result is the common type, and it is an l-value if both the second and third operands are of the same type and both are l-values.

[!WARNING] If the types of the second and third operands are not identical, then complex type conversion rules, as specified in the C++ Standard, are invoked. These conversions may lead to

unexpected behavior including construction and destruction of temporary objects. For this reason, we strongly advise you to either (1) avoid using user-defined types as operands with the conditional operator or (2) if you do use user-defined types, then explicitly cast each operand to a common type.

## Example

```
// expre_Expressions_with_the_Conditional_Operator.cpp
// compile with: /EHsc
// Demonstrate conditional operator
#include <iostream>
using namespace std;
int main() {
    int i = 1, j = 2;
    cout << ( i > j ? i : j ) << " is greater." << endl;
}
```

## See also

[C++ Built-in Operators, Precedence and Associativity](#)  
[Conditional-Expression Operator](#)



# delete Operator (C++)

---

Deallocates a block of memory.

## Syntax

```
[::] delete cast-expression  
[::] delete [] cast-expression
```

## Remarks

The *cast-expression* argument must be a pointer to a block of memory previously allocated for an object created with the [new operator](#). The **delete** operator has a result of type **void** and therefore does not return a value. For example:

```
CDialog* MyDialog = new CDialog;  
// use MyDialog  
delete MyDialog;
```

Using **delete** on a pointer to an object not allocated with **new** gives unpredictable results. You can, however, use **delete** on a pointer with the value 0. This provision means that, when **new** returns 0 on failure, deleting the result of a failed **new** operation is harmless. For more information, see [The new and delete Operators](#).

The **new** and **delete** operators can also be used for built-in types, including arrays. If *pointer* refers to an array, place empty brackets ([]) before *pointer*:

```
int* set = new int[100];  
//use set[]  
delete [] set;
```

Using the **delete** operator on an object deallocates its memory. A program that dereferences a pointer after the object is deleted can have unpredictable results or crash.

When **delete** is used to deallocate memory for a C++ class object, the object's destructor is called before the object's memory is deallocated (if the object has a destructor).

If the operand to the **delete** operator is a modifiable l-value, its value is undefined after the object is deleted.

If the [/sdl \(Enable additional security checks\)](#) compiler option is specified, the operand to the **delete** operator is set to an invalid value after the object is deleted.

## Using delete

There are two syntactic variants for the [delete operator](#): one for single objects and the other for arrays of objects. The following code fragment shows how they differ:

```

// expre_Using_delete.cpp
struct UDTType
{
};

int main()
{
    // Allocate a user-defined object, UDObject, and an object
    // of type double on the free store using the
    // new operator.
    UDTType *UDObject = new UDTType;
    double *dObject = new double;
    // Delete the two objects.
    delete UDObject;
    delete dObject;
    // Allocate an array of user-defined objects on the
    // free store using the new operator.
    UDTType (*UDArr)[7] = new UDTType[5][7];
    // Use the array syntax to delete the array of objects.
    delete [] UDArr;
}

```

The following two cases produce undefined results: using the array form of delete (`delete []`) on an object, and using the nonarray form of delete on an array.

## Example

For examples of using **delete**, see [new operator](#).

## How delete works

The delete operator invokes the function **operator delete**.

For objects not of class type ([class](#), [struct](#), or [union](#)), the global delete operator is invoked. For objects of class type, the name of the deallocation function is resolved in global scope if the delete expression begins with the unary scope resolution operator (`::`). Otherwise, the delete operator invokes the destructor for an object prior to deallocating memory (if the pointer is not null). The delete operator can be defined on a per-class basis; if there is no such definition for a given class, the global operator delete is invoked. If the delete expression is used to deallocate a class object whose static type has a virtual destructor, the deallocation function is resolved through the virtual destructor of the dynamic type of the object.

## See also

[Expressions with Unary Operators](#)

[Keywords](#)

[new and delete Operators](#)

# Equality Operators: == and !=

---

## Syntax

```
expression == expression
expression != expression
```

## Remarks

The binary equality operators compare their operands for strict equality or inequality.

The equality operators, equal to (==) and not equal to (!=), have lower precedence than the relational operators, but they behave similarly. The result type for these operators is **bool**.

The equal-to operator (==) returns **true** (1) if both operands have the same value; otherwise, it returns **false** (0). The not-equal-to operator (!=) returns **true** if the operands do not have the same value; otherwise, it returns **false**.

## Operator Keyword for !=

The `not_eq` operator is the text equivalent of `!=`. There are two ways to access the `not_eq` operator in your programs: include the header file `iso646.h`, or compile with the `/Za` (Disable language extensions) compiler option.

## Example

```
// expre_Equality_Operators.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;

int main() {
    cout << boolalpha
        << "The true expression 3 != 2 yields: "
        << (3 != 2) << endl
        << "The false expression 20 == 10 yields: "
        << (20 == 10) << endl;
}
```

Equality operators can compare pointers to members of the same type. In such a comparison, pointer-to-member conversions are performed. Pointers to members can also be compared to a constant expression that evaluates to 0.

## See also

Expressions with Binary Operators

C++ Built-in Operators, Precedence and Associativity

C Relational and Equality Operators

# Explicit Type Conversion Operator: ()

---

C++ allows explicit type conversion using syntax similar to the function-call syntax.

## Syntax

```
simple-type-name ( expression-list )
```

## Remarks

A *simple-type-name* followed by an *expression-list* enclosed in parentheses constructs an object of the specified type using the specified expressions. The following example shows an explicit type conversion to type `int`:

```
int i = int( d );
```

The following example shows a `Point` class.

## Example

```
// expre_Explicit_Type_Conversion_Operator.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;
class Point
{
public:
    // Define default constructor.
    Point() { _x = _y = 0; }
    // Define another constructor.
    Point( int X, int Y ) { _x = X; _y = Y; }

    // Define "accessor" functions as
    // reference types.
    unsigned& x() { return _x; }
    unsigned& y() { return _y; }
    void Show() { cout << "x = " << _x << ", "
                     << "y = " << _y << "\n"; }

private:
    unsigned _x;
    unsigned _y;
};

int main()
```

```

{
    Point Point1, Point2;

    // Assign Point1 the explicit conversion
    //   of ( 10, 10 ).
    Point1 = Point( 10, 10 );

    // Use x() as an l-value by assigning an explicit
    //   conversion of 20 to type unsigned.
    Point1.x() = unsigned( 20 );
    Point1.Show();

    // Assign Point2 the default Point object.
    Point2 = Point();
    Point2.Show();
}

```

## Output

```

x = 20, y = 10
x = 0, y = 0

```

Although the preceding example demonstrates explicit type conversion using constants, the same technique works to perform these conversions on objects. The following code fragment demonstrates this:

```

int i = 7;
float d;

d = float( i );

```

Explicit type conversions can also be specified using the "cast" syntax. The previous example, rewritten using the cast syntax, is:

```

d = (float)i;

```

Both cast and function-style conversions have the same results when converting from single values. However, in the function-style syntax, you can specify more than one argument for conversion. This difference is important for user-defined types. Consider a `Point` class and its conversions:

```

struct Point
{
    Point( short x, short y ) { _x = x; _y = y; }
    ...
    short _x, _y;
}

```

```
};  
...  
Point pt = Point( 3, 10 );
```

The preceding example, which uses function-style conversion, shows how to convert two values (one for *x* and one for *y*) to the user-defined type `Point`.

[!CAUTION] Use the explicit type conversions with care, since they override the C++ compiler's built-in type checking.

The `cast` notation must be used for conversions to types that do not have a *simple-type-name* (pointer or reference types, for example). Conversion to types that can be expressed with a *simple-type-name* can be written in either form.

Type definition within casts is illegal.

## See also

[Postfix Expressions](#)

[C++ Built-in Operators, Precedence and Associativity](#)

# Function Call Operator: ()

---

A postfix-expression followed by the function-call operator, **()**, specifies a function call.

## Syntax

```
postfix-expression  
( [argument-expression-list ] )
```

## Remarks

The arguments to the function-call operator are zero or more expressions separated by commas — the actual arguments to the function.

The *postfix-expression* must evaluate to a function address (for example, a function identifier or the value of a function pointer), and *argument-expression-list* is a list of expressions (separated by commas) whose values (the arguments) are passed to the function. The *argument-expression-list* argument can be empty.

The *postfix-expression* must be of one of these types:

- Function returning type **T**. An example declaration is

```
T func( int i )
```

- Pointer to a function returning type **T**. An example declaration is

```
T (*func)( int i )
```

- Reference to a function returning type **T**. An example declaration is

```
T (&func)(int i)
```

- Pointer-to-member function dereference returning type **T**. Example function calls are

```
(pObject->*pmf)();  
(Object.*pmf)();
```

## Example

The following example calls the standard library function **strcat\_s** with three arguments:



```

// expre_Function_Call_Operator.cpp
// compile with: /EHsc

#include <iostream>
#include <string>

// C++ Standard Library name space
using namespace std;

int main()
{
    enum
    {
        sizeOfBuffer = 20
    };

    char s1[ sizeOfBuffer ] = "Welcome to ";
    char s2[ ] = "C++";

    strcat_s( s1, sizeOfBuffer, s2 );

    cout << s1 << endl;
}

```

Welcome to C++

## Function call results

A function call evaluates to an r-value unless the function is declared as a reference type. Functions with reference return type evaluate to l-values, and can be used on the left side of an assignment statement as follows:

```

// expre_Function_Call_Results.cpp
// compile with: /EHsc
#include <iostream>
class Point
{
public:
    // Define "accessor" functions as
    // reference types.
    unsigned& x() { return _x; }
    unsigned& y() { return _y; }
private:
    unsigned _x;
    unsigned _y;
};

```

```

using namespace std;
int main()
{
    Point ThePoint;

    ThePoint.x() = 7;           // Use x() as an l-value.
    unsigned y = ThePoint.y(); // Use y() as an r-value.

    // Use x() and y() as r-values.
    cout << "x = " << ThePoint.x() << "\n"
         << "y = " << ThePoint.y() << "\n";
}

```

The preceding code defines a class called `Point`, which contains private data objects that represent `x` and `y` coordinates. These data objects must be modified and their values retrieved. This program is only one of several designs for such a class; use of the `GetX` and `SetX` or `GetY` and `SetY` functions is another possible design.

Functions that return class types, pointers to class types, or references to class types can be used as the left operand to member-selection operators. Therefore, the following code is legal:

```

// expre_Function_Results2.cpp
class A {
public:
    A() {}
    A(int i) {}
    int SetA( int i ) {
        return (I = i);
    }

    int GetA() {
        return I;
    }

private:
    int I;
};

A func1() {
    A a = 0;
    return a;
}

A* func2() {
    A *a = new A();
    return a;
}

A& func3() {
    A *a = new A();
    A &b = *a;
}

```

```
    return b;
}

int main() {
    int iResult = func1().GetA();
    func2()->SetA( 3 );
    func3().SetA( 7 );
}
```

Functions can be called recursively. For more information about function declarations, see [Functions](#). Related material is in [Translation units and linkage](#).

## See also

[Postfix Expressions](#)

[C++ Built-in Operators, Precedence and Associativity](#)

[Function Call](#)

# Indirection Operator: \*

---

## Syntax

```
* cast-expression
```

## Remarks

The unary indirection operator (\*) dereferences a pointer; that is, it converts a pointer value to an l-value. The operand of the indirection operator must be a pointer to a type. The result of the indirection expression is the type from which the pointer type is derived. The use of the \* operator in this context is different from its meaning as a binary operator, which is multiplication.

If the operand points to a function, the result is a function designator. If it points to a storage location, the result is an l-value designating the storage location.

The indirection operator may be used cumulatively to dereference pointers to pointers. For example:

```
// expre_Indirection_Operator.cpp
// compile with: /EHsc
// Demonstrate indirection operator
#include <iostream>
using namespace std;
int main() {
    int n = 5;
    int *pn = &n;
    int **ppn = &pn;

    cout << "Value of n:\n"
         << "direct value: " << n << endl
         << "indirect value: " << *pn << endl
         << "doubly indirect value: " << **ppn << endl
         << "address of n: " << pn << endl
         << "address of n via indirection: " << *ppn << endl;
}
```

If the pointer value is invalid, the result is undefined. The following list includes some of the most common conditions that invalidate a pointer value.

- The pointer is a null pointer.
- The pointer specifies the address of a local item that is not visible at the time of the reference.
- The pointer specifies an address that is inappropriately aligned for the type of the object pointed to.
- The pointer specifies an address not used by the executing program.

## See also

[Expressions with Unary Operators](#)

[C++ Built-in Operators, Precedence and Associativity](#)

[Address-of Operator: &](#)

[Indirection and Address-of Operators](#)

# Left Shift and Right Shift Operators (>> and <<)

The bitwise shift operators are the right-shift operator (>>), which moves the bits of *shift-expression* to the right, and the left-shift operator (<<), which moves the bits of *shift-expression* to the left. <sup>1</sup>

## Syntax

```
shift-expression << additive-expression shift-expression >> additive-expression
```

## Remarks

[!IMPORTANT] The following descriptions and examples are valid on Windows for x86 and x64 architectures. The implementation of left-shift and right-shift operators is significantly different on Windows for ARM devices. For more information, see the "Shift Operators" section of the [Hello ARM](#) blog post.

## Left Shifts

The left-shift operator causes the bits in *shift-expression* to be shifted to the left by the number of positions specified by *additive-expression*. The bit positions that have been vacated by the shift operation are zero-filled. A left shift is a logical shift (the bits that are shifted off the end are discarded, including the sign bit). For more information about the kinds of bitwise shifts, see [Bitwise shifts](#).

The following example shows left-shift operations using unsigned numbers. The example shows what is happening to the bits by representing the value as a bitset. For more information, see [bitset Class](#).

```
#include <iostream>
#include <bitset>

using namespace std;

int main() {
    unsigned short short1 = 4;
    bitset<16> bitset1{short1};    // the bitset representation of 4
    cout << bitset1 << endl;    // 0b00000000'00000100

    unsigned short short2 = short1 << 1;    // 4 left-shifted by 1 = 8
    bitset<16> bitset2{short2};
    cout << bitset2 << endl;    // 0b00000000'00001000

    unsigned short short3 = short1 << 2;    // 4 left-shifted by 2 = 16
    bitset<16> bitset3{short3};
    cout << bitset3 << endl;    // 0b00000000'00010000
}
```

If you left-shift a signed number so that the sign bit is affected, the result is undefined. The following example shows what happens when a 1 bit is left-shifted into the sign bit position.

```

#include <iostream>
#include <bitset>

using namespace std;

int main() {
    short short1 = 16384;
    bitset<16> bitset1(short1);
    cout << bitset1 << endl; // 0b01000000'00000000

    short short3 = short1 << 1;
    bitset<16> bitset3(short3); // 16384 left-shifted by 1 = -32768
    cout << bitset3 << endl; // 0b10000000'00000000

    short short4 = short1 << 14;
    bitset<16> bitset4(short4); // 4 left-shifted by 14 = 0
    cout << bitset4 << endl; // 0b00000000'00000000
}

```

## Right Shifts

The right-shift operator causes the bit pattern in *shift-expression* to be shifted to the right by the number of positions specified by *additive-expression*. For unsigned numbers, the bit positions that have been vacated by the shift operation are zero-filled. For signed numbers, the sign bit is used to fill the vacated bit positions. In other words, if the number is positive, 0 is used, and if the number is negative, 1 is used.

[!IMPORTANT] The result of a right-shift of a signed negative number is implementation-dependent. Although the Microsoft C++ compiler uses the sign bit to fill vacated bit positions, there is no guarantee that other implementations also do so.

This example shows right-shift operations using unsigned numbers:

```

#include <iostream>
#include <bitset>

using namespace std;

int main() {
    unsigned short short11 = 1024;
    bitset<16> bitset11{short11};
    cout << bitset11 << endl; // 0b00000100'00000000

    unsigned short short12 = short11 >> 1; // 512
    bitset<16> bitset12{short12};
    cout << bitset12 << endl; // 0b00000010'00000000

    unsigned short short13 = short11 >> 10; // 1
    bitset<16> bitset13{short13};
    cout << bitset13 << endl; // 0b00000000'00000001
}

```

```

    unsigned short short14 = short11 >> 11; // 0
    bitset<16> bitset14{short14};
    cout << bitset14 << endl; // 0b00000000'00000000
}

```

The next example shows right-shift operations with positive signed numbers.

```

#include <iostream>
#include <bitset>

using namespace std;

int main() {
    short short1 = 1024;
    bitset<16> bitset1(short1);
    cout << bitset1 << endl; // 0b00000100'00000000

    short short2 = short1 >> 1; // 512
    bitset<16> bitset2(short2);
    cout << bitset2 << endl; // 0b00000010'00000000

    short short3 = short1 >> 11; // 0
    bitset<16> bitset3(short3);
    cout << bitset3 << endl; // 0b00000000'00000000
}

```

The next example shows right-shift operations with negative signed integers.

```

#include <iostream>
#include <bitset>

using namespace std;

int main() {
    short neg1 = -16;
    bitset<16> bn1(neg1);
    cout << bn1 << endl; // 0b11111111'11110000

    short neg2 = neg1 >> 1; // -8
    bitset<16> bn2(neg2);
    cout << bn2 << endl; // 0b11111111'11111000

    short neg3 = neg1 >> 2; // -4
    bitset<16> bn3(neg3);
    cout << bn3 << endl; // 0b11111111'11111100

    short neg4 = neg1 >> 4; // -1
    bitset<16> bn4(neg4);
}

```



```

    cout << bn4 << endl; // 0b11111111'11111111

    short neg5 = neg1 >> 5; // -1
    bitset<16> bn5(neg5);
    cout << bn5 << endl; // 0b11111111'11111111
}

```

## Shifts and Promotions

The expressions on both sides of a shift operator must be integral types. Integral promotions are performed according to the rules described in [Standard Conversions](#). The type of the result is the same as the type of the promoted *shift-expression*.

In the following example, a variable of type **char** is promoted to an **int**.

```

#include <iostream>
#include <typeinfo>

using namespace std;

int main() {
    char char1 = 'a';

    auto promoted1 = char1 << 1; // 194
    cout << typeid(promoted1).name() << endl; // int

    auto promoted2 = char1 << 10; // 99328
    cout << typeid(promoted2).name() << endl; // int
}

```

## Additional Details

The result of a shift operation is undefined if *additive-expression* is negative or if *additive-expression* is greater than or equal to the number of bits in the (promoted) *shift-expression*. No shift operation is performed if *additive-expression* is 0.

```

#include <iostream>
#include <bitset>

using namespace std;

int main() {
    unsigned int int1 = 4;
    bitset<32> b1{int1};
    cout << b1 << endl; // 0b00000000'00000000'00000000'00000100

    unsigned int int2 = int1 << -3; // C4293: '<<': shift count negative or too
    big, undefined behavior
}

```

```

    unsigned int int3 = int1 >> -3; // C4293: '>>' : shift count negative or too
big, undefined behavior
    unsigned int int4 = int1 << 32; // C4293: '<<' : shift count negative or too
big, undefined behavior
    unsigned int int5 = int1 >> 32; // C4293: '>>' : shift count negative or too
big, undefined behavior
    unsigned int int6 = int1 << 0;
    bitset<32> b6{int6};
    cout << b6 << endl; // 0b00000000'00000000'00000000'00000100 (no change)
}

```

## Footnotes

<sup>1</sup> The following is the description of the shift operators in the C++11 ISO specification (INCITS/ISO/IEC 14882-2011[2012]), sections 5.8.2 and 5.8.3.

The value of  $E1 \ll E2$  is  $E1$  left-shifted  $E2$  bit positions; vacated bits are zero-filled. If  $E1$  has an unsigned type, the value of the result is  $E1 \times 2^{E2}$ , reduced modulo one more than the maximum value representable in the result type. Otherwise, if  $E1$  has a signed type and non-negative value, and  $E1 \times 2^{E2}$  is representable in the corresponding unsigned type of the result type, then that value, converted to the result type, is the resulting value; otherwise, the behavior is undefined.

The value of  $E1 \gg E2$  is  $E1$  right-shifted  $E2$  bit positions. If  $E1$  has an unsigned type or if  $E1$  has a signed type and a non-negative value, the value of the result is the integral part of the quotient of  $E1/2^{E2}$ . If  $E1$  has a signed type and a negative value, the resulting value is implementation-defined.

## See also

[Expressions with Binary Operators](#)

[C++ Built-in Operators, Precedence and Associativity](#)

# Logical AND Operator: &&

---

## Syntax

```
expression && expression
```

## Remarks

The logical AND operator (**&&**) returns the boolean value TRUE if both operands are TRUE and returns FALSE otherwise. The operands are implicitly converted to type **bool** prior to evaluation, and the result is of type **bool**. Logical AND has left-to-right associativity.

The operands to the logical AND operator need not be of the same type, but they must be of integral or pointer type. The operands are commonly relational or equality expressions.

The first operand is completely evaluated and all side effects are completed before continuing evaluation of the logical AND expression.

The second operand is evaluated only if the first operand evaluates to true (nonzero). This evaluation eliminates needless evaluation of the second operand when the logical AND expression is false. You can use this short-circuit evaluation to prevent null-pointer dereferencing, as shown in the following example:

```
char *pch = 0;
...
(pch) && (*pch = 'a');
```

If **pch** is null (0), the right side of the expression is never evaluated. Therefore, the assignment through a null pointer is impossible.

## Operator Keyword for &&

The **and** operator is the text equivalent of **&&**. There are two ways to access the **and** operator in your programs: include the header file **iso646.h**, or compile with the **/Za** (Disable language extensions) compiler option.

## Example

```
// expre_Logical_AND_Operator.cpp
// compile with: /EHsc
// Demonstrate logical AND
#include <iostream>

using namespace std;
```

```
int main() {  
    int a = 5, b = 10, c = 15;  
    cout << boolalpha  
        << "The true expression "  
        << "a < b && b < c yields "  
        << (a < b && b < c) << endl  
        << "The false expression "  
        << "a > b && b < c yields "  
        << (a > b && b < c) << endl;  
}
```

## See also

[C++ Built-in Operators Precedence and Associativity](#)

[C++ Built-in Operators, Precedence and Associativity](#)

[C Logical Operators](#)

# Logical Negation Operator: !

---

## Syntax

```
! cast-expression
```

## Remarks

The logical negation operator (!) reverses the meaning of its operand. The operand must be of arithmetic or pointer type (or an expression that evaluates to arithmetic or pointer type). The operand is implicitly converted to type **bool**. The result is TRUE if the converted operand is FALSE; the result is FALSE if the converted operand is TRUE. The result is of type **bool**.

For an expression *e*, the unary expression **!e** is equivalent to the expression **(e == 0)**, except where overloaded operators are involved.

## Operator Keyword for !

The **not** operator is an alternative spelling of **!**. There are two ways to access the **not** operator in your programs: include the header file `<iso646.h>`, or compile with the [/Za](#) (Disable language extensions) compiler option.

## Example

```
// expre_Logical_NOT_Operator.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

int main() {
    int i = 0;
    if (!i)
        cout << "i is zero" << endl;
}
```

## See also

[Expressions with Unary Operators](#)

[C++ Built-in Operators, Precedence and Associativity](#)

[Unary Arithmetic Operators](#)

# Logical OR operator: ||

---

## Syntax

```
logical-or-expression || logical-and-expression
```

## Remarks

The logical OR operator (||) returns the boolean value TRUE if either or both operands is TRUE and returns FALSE otherwise. The operands are implicitly converted to type **bool** prior to evaluation, and the result is of type **bool**. Logical OR has left-to-right associativity.

The operands to the logical OR operator need not be of the same type, but they must be of integral or pointer type. The operands are commonly relational or equality expressions.

The first operand is completely evaluated and all side effects are completed before continuing evaluation of the logical OR expression.

The second operand is evaluated only if the first operand evaluates to false (0). This eliminates needless evaluation of the second operand when the logical OR expression is true.

```
printf( "%d" , (x == w || x == y || x == z) );
```

In the above example, if **x** is equal to either **w**, **y**, or **z**, the second argument to the **printf** function evaluates to true and the value 1 is printed. Otherwise, it evaluates to false and the value 0 is printed. As soon as one of the conditions evaluates to true, evaluation ceases.

## Operator Keyword for ||

The **or** operator is the text equivalent of ||. There are two ways to access the **or** operator in your programs: include the header file <iso646.h>, or compile with the [/Za](#) (Disable language extensions) compiler option.

## Example

```
// expre_Logical_OR_Operator.cpp
// compile with: /EHsc
// Demonstrate logical OR
#include <iostream>
using namespace std;
int main() {
    int a = 5, b = 10, c = 15;
    cout << boolalpha
         << "The true expression "
         << "a < b || b > c yields "
         << (a < b || b > c) << endl
         << "The false expression "
         << "a > b || b > c yields "
```

```
} << (a > b || b > c) << endl;
```

## See also

[C++ Built-in Operators Precedence and Associativity](#)

[C++ Built-in Operators, Precedence and Associativity](#)

[C Logical Operators](#)

# Member Access Operators: . and ->

---

## Syntax

```
postfix-expression . name
postfix-expression -> name
```

## Remarks

The member access operators . and -> are used to refer to members of structures, unions, and classes. Member access expressions have the value and type of the selected member.

There are two forms of member access expressions:

1. In the first form, *postfix-expression* represents a value of struct, class, or union type, and *name* names a member of the specified structure, union, or class. The value of the operation is that of *name* and is an l-value if *postfix-expression* is an l-value.
2. In the second form, *postfix-expression* represents a pointer to a structure, union, or class, and *name* names a member of the specified structure, union, or class. The value is that of *name* and is an l-value. The -> operator dereferences the pointer. Therefore, the expressions *e->member* and *(\*e).member* (where *e* represents a pointer) yield identical results (except when the operators -> or \* are overloaded).

## Example

The following example demonstrates both forms of the member access operator.

```
// expre_Selection_Operator.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

struct Date {
    Date(int i, int j, int k) : day(i), month(j), year(k){}
    int month;
    int day;
    int year;
};

int main() {
    Date mydate(1,1,1900);
    mydate.month = 2;
    cout << mydate.month << "/" << mydate.day
         << "/" << mydate.year << endl;

    Date *mydate2 = new Date(1,1,2000);
```



```
mydate2->month = 2;  
cout << mydate2->month << "/" << mydate2->day  
      << "/" << mydate2->year << endl;  
delete mydate2;  
}
```

```
2/1/1900  
2/1/2000
```

## See also

[Postfix Expressions](#)

[C++ Built-in Operators, Precedence and Associativity](#)

[Classes and Structs](#)

[Structure and Union Members](#)

# Multiplicative Operators and the Modulus Operator

---

## Syntax

```
expression * expression  
expression / expression  
expression % expression
```

## Remarks

The multiplicative operators are:

- Multiplication (\*)
- Division (/)
- Modulus (remainder from division) (%)

These binary operators have left-to-right associativity.

The multiplicative operators take operands of arithmetic types. The modulus operator (%) has a stricter requirement in that its operands must be of integral type. (To get the remainder of a floating-point division, use the run-time function, [fmod](#).) The conversions covered in [Standard Conversions](#) are applied to the operands, and the result is of the converted type.

The multiplication operator yields the result of multiplying the first operand by the second.

The division operator yields the result of dividing the first operand by the second.

The modulus operator yields the remainder given by the following expression, where *e1* is the first operand and *e2* is the second:  $e1 - (e1 / e2) * e2$ , where both operands are of integral types.

Division by 0 in either a division or a modulus expression is undefined and causes a run-time error. Therefore, the following expressions generate undefined, erroneous results:

```
i % 0  
f / 0.0
```

If both operands to a multiplication, division, or modulus expression have the same sign, the result is positive. Otherwise, the result is negative. The result of a modulus operation's sign is implementation-defined.

[!NOTE] Since the conversions performed by the multiplicative operators do not provide for overflow or underflow conditions, information may be lost if the result of a multiplicative operation cannot be represented in the type of the operands after conversion.

## Microsoft Specific

In Microsoft C++, the result of a modulus expression is always the same as the sign of the first operand.

### END Microsoft Specific

If the computed division of two integers is inexact and only one operand is negative, the result is the largest integer (in magnitude, disregarding the sign) that is less than the exact value the division operation would yield. For example, the computed value of  $-11 / 3$  is  $-3.666666666$ . The result of that integral division is  $-3$ .

The relationship between the multiplicative operators is given by the identity  $(e1 / e2) * e2 + e1 \% e2 == e1$ .

## Example

The following program demonstrates the multiplicative operators. Note that either operand of  $10 / 3$  must be explicitly cast to type **float** to avoid truncation so that both operands are of type **float** before division.

```
// expre_Multiplicative_Operators.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;
int main() {
    int x = 3, y = 6, z = 10;
    cout << "3 * 6 is " << x * y << endl
         << "6 / 3 is " << y / x << endl
         << "10 % 3 is " << z % x << endl
         << "10 / 3 is " << (float) z / x << endl;
}
```

## See also

[Expressions with Binary Operators](#)

[C++ Built-in Operators, Precedence and Associativity](#)

[C Multiplicative Operators](#)

# new Operator (C++)

---

Allocates memory for an object or array of objects of *type-name* from the free store and returns a suitably typed, nonzero pointer to the object.

[!NOTE] Microsoft C++ Component Extensions provides support for the **new** keyword to add vtable slot entries. For more information, see [new \(new slot in vtable\)](#)

## Syntax

```
[::] new [placement] new-type-name [new-initializer]  
[::] new [placement] ( type-name ) [new-initializer]
```

## Remarks

If unsuccessful, **new** returns zero or throws an exception; see [The new and delete Operators](#) for more information. You can change this default behavior by writing a custom exception-handling routine and calling the [\\_set\\_new\\_handler](#) run-time library function with your function name as its argument.

For information on how to create an object on the managed heap, see [gcnew](#).

When **new** is used to allocate memory for a C++ class object, the object's constructor is called after the memory is allocated.

Use the [delete](#) operator to deallocate the memory allocated with the **new** operator.

The following example allocates and then frees a two-dimensional array of characters of size *dim* by 10. When allocating a multidimensional array, all dimensions except the first must be constant expressions that evaluate to positive values; the leftmost array dimension can be any expression that evaluates to a positive value. When allocating an array using the **new** operator, the first dimension can be zero — the **new** operator returns a unique pointer.

```
char (*pchar)[10] = new char[dim][10];  
delete [] pchar;
```

The *type-name* cannot contain **const**, **volatile**, class declarations, or enumeration declarations. Therefore, the following expression is illegal:

```
volatile char *vch = new volatile char[20];
```

The **new** operator does not allocate reference types because they are not objects.

The **new** operator cannot be used to allocate a function, but it can be used to allocate pointers to functions. The following example allocates and then frees an array of seven pointers to functions that return integers.

```
int (**p) () = new (int (*)(7)) ();  
delete *p;
```

If you use the operator **new** without any extra arguments, and compile with the `/GX`, `/EHa`, or `/EHs` option, the compiler will generate code to call operator **delete** if the constructor throws an exception.

The following list describes the grammar elements of **new**:

*placement*

Provides a way of passing additional arguments if you overload **new**.

*type-name*

Specifies type to be allocated; it can be either a built-in or user-defined type. If the type specification is complicated, it can be surrounded by parentheses to force the order of binding.

*initializer*

Provides a value for the initialized object. Initializers cannot be specified for arrays. The **new** operator will create arrays of objects only if the class has a default constructor.

## Example

The following code example allocates a character array and an object of class `CName` and then frees them.

```
// expre_new_Operator.cpp  
// compile with: /EHsc  
#include <string.h>  
  
class CName {  
public:  
    enum {  
        sizeofBuffer = 256  
    };  
  
    char m_szFirst[sizeofBuffer];  
    char m_szLast[sizeofBuffer];  
  
public:  
    void SetName(char* pszFirst, char* pszLast) {  
        strcpy_s(m_szFirst, sizeofBuffer, pszFirst);  
        strcpy_s(m_szLast, sizeofBuffer, pszLast);  
    }  
};  
  
int main() {  
    // Allocate memory for the array  
    char* pCharArray = new char[CName::sizeofBuffer];
```

```

strcpy_s(pCharArray, CName::sizeofBuffer, "Array of characters");

// Deallocate memory for the array
delete [] pCharArray;
pCharArray = NULL;

// Allocate memory for the object
CName* pName = new CName;
pName->SetName("Firstname", "Lastname");

// Deallocate memory for the object
delete pName;
pName = NULL;
}

```

## Example

If you use the placement new form of the **new** operator, the form with arguments in addition to the size of the allocation, the compiler does not support a placement form of the **delete** operator if the constructor throws an exception. For example:

```

// expre_new_Operator2.cpp
// C2660 expected
class A {
public:
    A(int) { throw "Fail!"; }
};

void F(void) {
    try {
        // heap memory pointed to by pa1 will be deallocated
        // by calling ::operator delete(void*).
        A* pa1 = new A(10);
    } catch (...) {
    }
    try {
        // This will call ::operator new(size_t, char*, int).
        // When A::A(int) does a throw, we should call
        // ::operator delete(void*, char*, int) to deallocate
        // the memory pointed to by pa2. Since
        // ::operator delete(void*, char*, int) has not been implemented,
        // memory will be leaked when the deallocation cannot occur.

        A* pa2 = new(__FILE__, __LINE__) A(20);
    } catch (...) {
    }
}

int main() {
    A a;
}

```

## Initializing object allocated with new

An optional *initializer* field is included in the grammar for the **new** operator. This allows new objects to be initialized with user-defined constructors. For more information about how initialization is done, see [Initializers](#). The following example illustrates how to use an initialization expression with the **new** operator:

```
// expre_Initializing_Objects_Allocated_with_new.cpp
class Acct
{
public:
    // Define default constructor and a constructor that accepts
    // an initial balance.
    Acct() { balance = 0.0; }
    Acct( double init_balance ) { balance = init_balance; }
private:
    double balance;
};

int main()
{
    Acct *CheckingAcct = new Acct;
    Acct *SavingsAcct = new Acct ( 34.98 );
    double *HowMuch = new double ( 43.0 );
    // ...
}
```

In this example, the object `CheckingAcct` is allocated using the **new** operator, but no default initialization is specified. Therefore, the default constructor for the class, `Acct()`, is called. Then the object `SavingsAcct` is allocated the same way, except that it is explicitly initialized to 34.98. Because 34.98 is of type **double**, the constructor that takes an argument of that type is called to handle the initialization. Finally, the nonclass type `HowMuch` is initialized to 43.0.

If an object is of a class type and that class has constructors (as in the preceding example), the object can be initialized by the **new** operator only if one of these conditions is met:

- The arguments provided in the initializer agree with those of a constructor.
- The class has a default constructor (a constructor that can be called with no arguments).

No explicit per-element initialization can be done when allocating arrays using the **new** operator; only the default constructor, if present, is called. See [Default Arguments](#) for more information.

If the memory allocation fails (**operator new** returns a value of 0), no initialization is performed. This protects against attempts to initialize data that does not exist.

As with function calls, the order in which initialized expressions are evaluated is not defined. Furthermore, you should not rely on these expressions being completely evaluated before the memory allocation is performed. If the memory allocation fails and the **new** operator returns zero, some expressions in the initializer may not be completely evaluated.

## Lifetime of objects allocated with new

Objects allocated with the **new** operator are not destroyed when the scope in which they are defined is exited. Because the **new** operator returns a pointer to the objects it allocates, the program must define a pointer with suitable scope to access those objects. For example:

```
// expre_Lifetime_of_Objects_Allocated_with_new.cpp
// C2541 expected
int main()
{
    // Use new operator to allocate an array of 20 characters.
    char *AnArray = new char[20];

    for( int i = 0; i < 20; ++i )
    {
        // On the first iteration of the loop, allocate
        // another array of 20 characters.
        if( i == 0 )
        {
            char *AnotherArray = new char[20];
        }
    }

    delete [] AnotherArray; // Error: pointer out of scope.
    delete [] AnArray;      // OK: pointer still in scope.
}
```

Once the pointer `AnotherArray` goes out of scope in the example, the object can no longer be deleted.

## How new works

The *allocation-expression* — the expression containing the **new** operator — does three things:

- Locates and reserves storage for the object or objects to be allocated. When this stage is complete, the correct amount of storage is allocated, but it is not yet an object.
- Initializes the object(s). Once initialization is complete, enough information is present for the allocated storage to be an object.
- Returns a pointer to the object(s) of a pointer type derived from *new-type-name* or *type-name*. The program uses this pointer to access the newly allocated object.

The **new** operator invokes the function **operator new**. For arrays of any type, and for objects that are not of **class**, **struct**, or **union** types, a global function, **::operator new**, is called to allocate storage. Class-type objects can define their own **operator new** static member function on a per-class basis.

When the compiler encounters the **new** operator to allocate an object of type **type**, it issues a call to **type::operator new( sizeof( type ) )** or, if no user-defined **operator new** is defined, **::operator new( sizeof( type ) )**. Therefore, the **new** operator can allocate the correct amount of memory for the object.



[!NOTE] The argument to **operator new** is of type `size_t`. This type is defined in `<direct.h>`, `<malloc.h>`, `<memory.h>`, `<search.h>`, `<stddef.h>`, `<stdio.h>`, `<stdlib.h>`, `<string.h>`, and `<time.h>`.

An option in the grammar allows specification of *placement* (see the Grammar for [new Operator](#)). The *placement* parameter can be used only for user-defined implementations of **operator new**; it allows extra information to be passed to **operator new**. An expression with a *placement* field such as `T *TObject = new ( 0x0040 ) T;` is translated to `T *TObject = T::operator new( sizeof( T ), 0x0040 );` if class T has member operator new, otherwise to `T *TObject = ::operator new( sizeof( T ), 0x0040 );`.

The original intention of the *placement* field was to allow hardware-dependent objects to be allocated at user-specified addresses.

[!NOTE] Although the preceding example shows only one argument in the *placement* field, there is no restriction on how many extra arguments can be passed to **operator new** this way.

Even when **operator new** has been defined for a class type, the global operator can be used by using the form of this example:

```
T *TObject = ::new TObject;
```

The scope-resolution operator (`::`) forces use of the global **new** operator.

## See also

[Expressions with Unary Operators](#)

[Keywords](#)

[new and delete operators](#)

# One's Complement Operator: ~

---

## Syntax

```
~ cast-expression
```

## Remarks

The one's complement operator (`~`), sometimes called the "bitwise complement" operator, yields a bitwise one's complement of its operand. That is, every bit that is 1 in the operand is 0 in the result. Conversely, every bit that is 0 in the operand is 1 in the result. The operand to the one's complement operator must be an integral type.

## Operator Keyword for ~

The **compl** operator is the text equivalent of `~`. There are two ways to access the **compl** operator in your programs: include the header file `iso646.h`, or compile with `/Za`.

## Example

```
// expre_One_Complement_Operator.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;

int main () {
    unsigned short y = 0xFFFF;
    cout << hex << y << endl;
    y = ~y;    // Take one's complement
    cout << hex << y << endl;
}
```

In this example, the new value assigned to `y` is the one's complement of the unsigned value `0xFFFF`, or `0x0000`.

Integral promotion is performed on integral operands, and the resultant type is the type to which the operand is promoted. See [Standard Conversions](#) for more information on how the promotion is done.

## See also

[Expressions with Unary Operators](#)

[C++ Built-in Operators, Precedence and Associativity](#)

[Unary Arithmetic Operators](#)

# Pointer-to-Member Operators: .\* and ->\*

---

## Syntax

```
expression .* expression
expression ->* expression
```

## Remarks

The pointer-to-member operators, .\* and ->\*, return the value of a specific class member for the object specified on the left side of the expression. The right side must specify a member of the class. The following example shows how to use these operators:

```
// expre_Expressions_with_Pointer_Member_Operators.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;

class Testpm {
public:
    void m_func1() { cout << "m_func1\n"; }
    int m_num;
};

// Define derived types pmfn and pmd.
// These types are pointers to members m_func1() and
// m_num, respectively.
void (Testpm::*pmfn)() = &Testpm::m_func1;
int Testpm::*pmd = &Testpm::m_num;

int main() {
    Testpm ATestpm;
    Testpm *pTestpm = new Testpm;

    // Access the member function
    (ATestpm.*pmfn)();
    (pTestpm->*pmfn)();    // Parentheses required since * binds
                          // less tightly than the function call.

    // Access the member data
    ATestpm.*pmd = 1;
    pTestpm->*pmd = 2;

    cout << ATestpm.*pmd << endl
         << pTestpm->*pmd << endl;
```

```
    delete pTestpm;
}
```

## Output

```
m_func1
m_func1
1
2
```

In the preceding example, a pointer to a member, `pmfn`, is used to invoke the member function `m_func1`. Another pointer to a member, `pmd`, is used to access the `m_num` member.

The binary operator `.*` combines its first operand, which must be an object of class type, with its second operand, which must be a pointer-to-member type.

The binary operator `->*` combines its first operand, which must be a pointer to an object of class type, with its second operand, which must be a pointer-to-member type.

In an expression containing the `.*` operator, the first operand must be of the class type of, and be accessible to, the pointer to member specified in the second operand or of an accessible type unambiguously derived from and accessible to that class.

In an expression containing the `->*` operator, the first operand must be of the type "pointer to the class type" of the type specified in the second operand, or it must be of a type unambiguously derived from that class.

## Example

Consider the following classes and program fragment:

```
// expre_Expressions_with_Pointer_Member_Operators2.cpp
// C2440 expected
class BaseClass {
public:
    BaseClass(); // Base class constructor.
    void Func1();
};

// Declare a pointer to member function Func1.
void (BaseClass::*pmfnFunc1)() = &BaseClass::Func1;

class Derived : public BaseClass {
public:
    Derived(); // Derived class constructor.
    void Func2();
};

// Declare a pointer to member function Func2.
```

```

void (Derived::*pmfnFunc2)() = &Derived::Func2;

int main() {
    BaseClass ABase;
    Derived ADerived;

    (ABase.*pmfnFunc1)();    // OK: defined for BaseClass.
    (ABase.*pmfnFunc2)();    // Error: cannot use base class to
                             // access pointers to members of
                             // derived classes.

    (ADerived.*pmfnFunc1)(); // OK: Derived is unambiguously
                             // derived from BaseClass.
    (ADerived.*pmfnFunc2)(); // OK: defined for Derived.
}

```

The result of the `.*` or `->*` pointer-to-member operators is an object or function of the type specified in the declaration of the pointer to member. So, in the preceding example, the result of the expression `ADerived.*pmfnFunc1()` is a pointer to a function that returns void. This result is an l-value if the second operand is an l-value.

[!NOTE] If the result of one of the pointer-to-member operators is a function, then the result can be used only as an operand to the function call operator.

## See also

[C++ Built-in Operators, Precedence and Associativity](#)

# Postfix Increment and Decrement Operators: ++ and --

---

## Syntax

```
postfix-expression ++  
postfix-expression --
```

## Remarks

C++ provides prefix and postfix increment and decrement operators; this section describes only the postfix increment and decrement operators. (For more information, see [Prefix Increment and Decrement Operators](#).) The difference between the two is that in the postfix notation, the operator appears after *postfix-expression*, whereas in the prefix notation, the operator appears before *expression*. The following example shows a postfix-increment operator:

```
i++;
```

The effect of applying the postfix increment operator (++) is that the operand's value is increased by one unit of the appropriate type. Similarly, the effect of applying the postfix decrement operator (--) is that the operand's value is decreased by one unit of the appropriate type.

It is important to note that a postfix increment or decrement expression evaluates to the value of the expression *prior to* application of the respective operator. The increment or decrement operation occurs *after* the operand is evaluated. This issue arises only when the postfix increment or decrement operation occurs in the context of a larger expression.

When a postfix operator is applied to a function argument, the value of the argument is not guaranteed to be incremented or decremented before it is passed to the function. See section 1.9.17 in the C++ standard for more information.

Applying the postfix increment operator to a pointer to an array of objects of type **long** actually adds four to the internal representation of the pointer. This behavior causes the pointer, which previously referred to the *n*th element of the array, to refer to the (*n*+1)th element.

The operands to postfix increment and postfix decrement operators must be modifiable (not **const**) l-values of arithmetic or pointer type. The type of the result is the same as that of the *postfix-expression*, but it is no longer an l-value.

**Visual Studio 2017 version 15.3 and later** (available with [/std:c++17](#)): The operand of a postfix increment or decrement operator may not be of type **bool**.

The following code illustrates the postfix increment operator:

```
// expre_Postfix_Increment_and_Decrement_Operators.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

int main() {
    int i = 10;
    cout << i++ << endl;
    cout << i << endl;
}
```

Postincrement and postdecrement operations on enumerated types are not supported:

```
enum Compass { North, South, East, West };
Compass myCompass;
for( myCompass = North; myCompass != West; myCompass++ ) // Error
```

## See also

[Postfix Expressions](#)

[C++ Built-in Operators, Precedence and Associativity](#)

[C Postfix Increment and Decrement Operators](#)

# Prefix Increment and Decrement Operators: ++ and --

---

## Syntax

```
++ unary-expression  
-- unary-expression
```

## Remarks

The prefix increment operator (++) adds one to its operand; this incremented value is the result of the expression. The operand must be an l-value not of type **const**. The result is an l-value of the same type as the operand.

The prefix decrement operator (--) is analogous to the prefix increment operator, except that the operand is decremented by one and the result is this decremented value.

**Visual Studio 2017 version 15.3 and later** (available with [/std:c++17](#)): The operand of an increment or decrement operator may not be of type **bool**.

Both the prefix and postfix increment and decrement operators affect their operands. The key difference between them is the order in which the increment or decrement takes place in the evaluation of an expression. (For more information, see [Postfix Increment and Decrement Operators](#).) In the prefix form, the increment or decrement takes place before the value is used in expression evaluation, so the value of the expression is different from the value of the operand. In the postfix form, the increment or decrement takes place after the value is used in expression evaluation, so the value of the expression is the same as the value of the operand. For example, the following program prints "++i = 6":

```
// expre_Increment_and_Decrement_Operators.cpp  
// compile with: /EHsc  
#include <iostream>  
  
using namespace std;  
  
int main() {  
    int i = 5;  
    cout << "++i = " << ++i << endl;  
}
```

An operand of integral or floating type is incremented or decremented by the integer value 1. The type of the result is the same as the operand type. An operand of pointer type is incremented or decremented by the size of the object it addresses. An incremented pointer points to the next object; a decremented pointer points to the previous object.

Because increment and decrement operators have side effects, using expressions with increment or decrement operators in a [preprocessor macro](#) can have undesirable results. Consider this example:



```
// expre_Increment_and_Decrement_Operators2.cpp
#define max(a,b) ((a)<(b))?(b):(a)

int main()
{
    int i = 0, j = 0, k;
    k = max( ++i, j );
}
```

The macro expands to:

```
k = ((++i)<(j))?(j):(++i);
```

If `i` is greater than or equal to `j` or less than `j` by 1, it will be incremented twice.

[!NOTE] C++ inline functions are preferable to macros in many cases because they eliminate side effects such as those described here, and allow the language to perform more complete type checking.

## See also

[Expressions with Unary Operators](#)

[C++ Built-in Operators, Precedence and Associativity](#)

[Prefix Increment and Decrement Operators](#)

# Relational Operators: <, >, <=, and >=

---

## Syntax

```
expression < expression
expression > expression
expression <= expression
expression >= expression
```

## Remarks

The binary relational operators determine the following relationships:

- Less than (<)
- Greater than (>)
- Less than or equal to (<=)
- Greater than or equal to (>=)

The relational operators have left-to-right associativity. Both operands of relational operators must be of arithmetic or pointer type. They yield values of type **bool**. The value returned is **false** (0) if the relationship in the expression is false; otherwise, the value returned is **true** (1).

## Example

```
// expre_Relational_Operators.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;

int main() {
    cout << "The true expression 3 > 2 yields: "
         << (3 > 2) << endl
         << "The false expression 20 < 10 yields: "
         << (20 < 10) << endl;
}
```

The expressions in the preceding example must be enclosed in parentheses because the stream insertion operator (<<) has higher precedence than the relational operators. Therefore, the first expression without the parentheses would be evaluated as:

```
(cout << "The true expression 3 > 2 yields: " << 3) < (2 << "\n");
```

---

The usual arithmetic conversions covered in [Standard Conversions](#) are applied to operands of arithmetic types.

## Comparing pointers

When two pointers to objects of the same type are compared, the result is determined by the location of the objects pointed to in the program's address space. Pointers can also be compared to a constant expression that evaluates to 0 or to a pointer of type `void *`. If a pointer comparison is made against a pointer of type `void *`, the other pointer is implicitly converted to type `void *`. Then the comparison is made.

Two pointers of different types cannot be compared unless:

- One type is a class type derived from the other type.
- At least one of the pointers is explicitly converted (cast) to type `void *`. (The other pointer is implicitly converted to type `void *` for the conversion.)

Two pointers of the same type that point to the same object are guaranteed to compare equal. If two pointers to nonstatic members of an object are compared, the following rules apply:

- If the class type is not a **union**, and if the two members are not separated by an *access-specifier*, such as **public**, **protected**, or **private**, the pointer to the member declared last will compare greater than the pointer to the member declared earlier.
- If the two members are separated by an *access-specifier*, the results are undefined.
- If the class type is a **union**, pointers to different data members in that **union** compare equal.

If two pointers point to elements of the same array or to the element one beyond the end of the array, the pointer to the object with the higher subscript compares higher. Comparison of pointers is guaranteed valid only when the pointers refer to objects in the same array or to the location one past the end of the array.

## See also

[Expressions with Binary Operators](#)

[C++ Built-in Operators, Precedence and Associativity](#)

[C Relational and Equality Operators](#)

# Scope Resolution Operator: ::

---

The scope resolution operator `::` is used to identify and disambiguate identifiers used in different scopes. For more information about scope, see [Scope](#).

## Syntax

```
:: identifier  
class-name :: identifier  
namespace :: identifier  
enum class :: identifier  
enum struct :: identifier
```

## Remarks

The `identifier` can be a variable, a function, or an enumeration value.

## With Classes and Namespaces

The following example shows how the scope resolution operator is used with namespaces and classes:

```
namespace NamespaceA{  
    int x;  
    class ClassA {  
    public:  
        int x;  
    };  
}  
  
int main() {  
  
    // A namespace name used to disambiguate  
    NamespaceA::x = 1;  
  
    // A class name used to disambiguate  
    NamespaceA::ClassA a1;  
    a1.x = 2;  
}
```

A scope resolution operator without a scope qualifier refers to the global namespace.

```
namespace NamespaceA{  
    int x;  
}
```

```

int x;

int main() {
    int x;

    // the x in main()
    x = 0;
    // The x in the global namespace
    ::x = 1;

    // The x in the A namespace
    NamespaceA::x = 2;
}

```

You can use the scope resolution operator to identify a member of a namespace, or to identify a namespace that nominates the member's namespace in a using-directive. In the example below, you can use `NamespaceC` to qualify `ClassB`, even though `ClassB` was declared in namespace `NamespaceB`, because `NamespaceB` was nominated in `NamespaceC` by a using directive.

```

namespace NamespaceB {
    class ClassB {
    public:
        int x;
    };
}

namespace NamespaceC{
    using namespace B;
}

int main() {
    NamespaceB::ClassB c_b;
    NamespaceC::ClassB c_c;

    c_b.x = 3;
    c_c.x = 4;
}

```

You can use chains of scope resolution operators. In the following example, `NamespaceD::NamespaceD1` identifies the nested namespace `NamespaceD1`, and `NamespaceE::ClassE::ClassE1` identifies the nested class `ClassE1`.

```

namespace NamespaceD{
    namespace NamespaceD1{
        int x;
    }
}

namespace NamespaceE{
    class ClassE{

```

```

    public:
        class ClassE1{
        public:
            int x;
        };
    };

int main() {
    NamespaceD:: NamespaceD1::x = 6;
    NamespaceE::ClassE::ClassE1 e1;
    e1.x = 7 ;
}

```

## With Static Members

You must use the scope resolution operator to call static members of classes.

```

class ClassG {
public:
    static int get_x() { return x;}
    static int x;
};

int ClassG::x = 6;

int main() {

    int gx1 = ClassG::x;
    int gx2 = ClassG::get_x();
}

```

## With Scoped Enumerations

The scoped resolution operator is also used with the values of a scoped enumeration [Enumeration Declarations](#), as in the following example:

```

enum class EnumA{
    First,
    Second,
    Third
};

int main() {
    EnumA enum_value = EnumA::First;
}

```

## See also

[C++ Built-in Operators, Precedence and Associativity](#)  
[Namespaces](#)

# sizeof Operator

---

Yields the size of its operand with respect to the size of type **char**.

[!NOTE] For information about the `sizeof ...` operator, see [Ellipses and Variadic Templates](#).

## Syntax

```
sizeof unary-expression  
sizeof ( type-name )
```

## Remarks

The result of the **sizeof** operator is of type `size_t`, an integral type defined in the include file `<stddef.h>`. This operator allows you to avoid specifying machine-dependent data sizes in your programs.

The operand to **sizeof** can be one of the following:

- A type name. To use **sizeof** with a type name, the name must be enclosed in parentheses.
- An expression. When used with an expression, **sizeof** can be specified with or without the parentheses. The expression is not evaluated.

When the **sizeof** operator is applied to an object of type **char**, it yields 1. When the **sizeof** operator is applied to an array, it yields the total number of bytes in that array, not the size of the pointer represented by the array identifier. To obtain the size of the pointer represented by the array identifier, pass it as a parameter to a function that uses **sizeof**. For example:

## Example

```
#include <iostream>  
using namespace std;  
  
size_t getPtrSize( char *ptr )  
{  
    return sizeof( ptr );  
}  
  
int main()  
{  
    char szHello[] = "Hello, world!";  
  
    cout << "The size of a char is: "  
        << sizeof( char )  
        << "\nThe length of " << szHello << " is: "  
        << sizeof szHello  
        << "\nThe size of the pointer is "
```



```
    << getPtrSize( szHello ) << endl;  
}
```

## Sample Output

```
The size of a char is: 1  
The length of Hello, world! is: 14  
The size of the pointer is 4
```

When the **sizeof** operator is applied to a **class**, **struct**, or **union** type, the result is the number of bytes in an object of that type, plus any padding added to align members on word boundaries. The result does not necessarily correspond to the size calculated by adding the storage requirements of the individual members. The `/Zp` compiler option and the `pack` pragma affect alignment boundaries for members.

The **sizeof** operator never yields 0, even for an empty class.

The **sizeof** operator cannot be used with the following operands:

- Functions. (However, **sizeof** can be applied to pointers to functions.)
- Bit fields.
- Undefined classes.
- The type **void**.
- Dynamically allocated arrays.
- External arrays.
- Incomplete types.
- Parenthesized names of incomplete types.

When the **sizeof** operator is applied to a reference, the result is the same as if **sizeof** had been applied to the object itself.

If an unsized array is the last element of a structure, the **sizeof** operator returns the size of the structure without the array.

The **sizeof** operator is often used to calculate the number of elements in an array using an expression of the form:

```
sizeof array / sizeof array[0]
```

## See also

Expressions with Unary Operators

Keywords

# Subscript Operator []

---

## Syntax

```
postfix-expression [ expression ]
```

## Remarks

A postfix expression (which can also be a primary expression) followed by the subscript operator, **[]**, specifies array indexing.

For information about managed arrays in C++/CLI, see [Arrays](#).

Usually, the value represented by *postfix-expression* is a pointer value, such as an array identifier, and *expression* is an integral value (including enumerated types). However, all that is required syntactically is that one of the expressions be of pointer type and the other be of integral type. Thus the integral value could be in the *postfix-expression* position and the pointer value could be in the brackets in the *expression* or subscript position. Consider the following code fragment:

```
int nArray[5] = { 0, 1, 2, 3, 4 };
cout << nArray[2] << endl;           // prints "2"
cout << 2[nArray] << endl;           // prints "2"
```

In the preceding example, the expression `nArray[2]` is identical to `2[nArray]`. The reason is that the result of a subscript expression `e1[e2]` is given by:

$*((e2) + (e1))$

The address yielded by the expression is not `e2` bytes from the address `e1`. Rather, the address is scaled to yield the next object in the array `e2`. For example:

```
double aDb1[2];
```

The addresses of `aDb[0]` and `aDb[1]` are 8 bytes apart — the size of an object of type **double**. This scaling according to object type is done automatically by the C++ language and is defined in [Additive Operators](#) where addition and subtraction of operands of pointer type is discussed.

A subscript expression can also have multiple subscripts, as follows:

*expression1* [*expression2*] [*expression3*] ...

Subscript expressions associate from left to right. The leftmost subscript expression, *expression1* [*expression2*], is evaluated first. The address that results from adding *expression1* and *expression2* forms a pointer

expression; then *expression3* is added to this pointer expression to form a new pointer expression, and so on until the last subscript expression has been added. The indirection operator (\*) is applied after the last subscripted expression is evaluated, unless the final pointer value addresses an array type.

Expressions with multiple subscripts refer to elements of multidimensional arrays. A multidimensional array is an array whose elements are arrays. For example, the first element of a three-dimensional array is an array with two dimensions. The following example declares and initializes a simple two-dimensional array of characters:

```
// expre_Subscript_Operator.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;
#define MAX_ROWS 2
#define MAX_COLS 2

int main() {
    char c[ MAX_ROWS ][ MAX_COLS ] = { { 'a', 'b' }, { 'c', 'd' } };
    for ( int i = 0; i < MAX_ROWS; i++ )
        for ( int j = 0; j < MAX_COLS; j++ )
            cout << c[ i ][ j ] << endl;
}
```

## Positive and negative subscripts

The first element of an array is element 0. The range of a C++ array is from *array[0]* to *array[size - 1]*. However, C++ supports positive and negative subscripts. Negative subscripts must fall within array boundaries; if they do not, the results are unpredictable. The following code shows positive and negative array subscripts:

```
#include <iostream>
using namespace std;

int main() {
    int intArray[1024];
    for (int i = 0, j = 0; i < 1024; i++)
    {
        intArray[i] = j++;
    }

    cout << intArray[512] << endl;    // 512

    cout << 257[intArray] << endl;    // 257

    int *midArray = &intArray[512]; // pointer to the middle of the array

    cout << midArray[-256] << endl;    // 256
}
```

```
    cout << intArray[-256] << endl; // unpredictable, may crash  
}
```

The negative subscript in the last line can produce a run-time error because it points to an address 256 **int** positions lower in memory than the origin of the array. The pointer **midArray** is initialized to the middle of **intArray**; it is therefore possible (but dangerous) to use both positive and negative array indices on it. Array subscript errors do not generate compile-time errors, but they yield unpredictable results.

The subscript operator is commutative. Therefore, the expressions *array[index]* and *index[array]* are guaranteed to be equivalent as long as the subscript operator is not overloaded (see [Overloaded Operators](#)). The first form is the most common coding practice, but either works.

## See also

[Postfix Expressions](#)

[C++ Built-in Operators, Precedence and Associativity](#)

[Arrays](#)

[One-Dimensional Arrays](#)

[Multidimensional Arrays](#)

# typeid Operator

---

## Syntax

```
typeid(type-id)
typeid(expression)
```

## Remarks

The **typeid** operator allows the type of an object to be determined at run time.

The result of **typeid** is a `const type_info&`. The value is a reference to a `type_info` object that represents either the *type-id* or the type of the *expression*, depending on which form of **typeid** is used. For more information, see [type\\_info Class](#).

The **typeid** operator doesn't work with managed types (abstract declarators or instances). For information on getting the [xref:System.Type](#) of a specified type, see [typeid](#).

The **typeid** operator does a run-time check when applied to an l-value of a polymorphic class type, where the true type of the object can't be determined by the static information provided. Such cases are:

- A reference to a class
- A pointer, dereferenced with `*`
- A subscripted pointer (`[ ]`). (It's not safe to use a subscript with a pointer to a polymorphic type.)

If the *expression* points to a base class type, yet the object is actually of a type derived from that base class, a `type_info` reference for the derived class is the result. The *expression* must point to a polymorphic type (a class with virtual functions). Otherwise, the result is the `type_info` for the static class referred to in the *expression*. Further, the pointer must be dereferenced so that the object used is the one it points to. Without dereferencing the pointer, the result will be the `type_info` for the pointer, not what it points to. For example:

```
// expre_typeid_Operator.cpp
// compile with: /GR /EHsc
#include <iostream>
#include <typeinfo>

class Base {
public:
    virtual void vfunc() {}
};

class Derived : public Base {};

using namespace std;
int main() {
```

```

Derived* pd = new Derived;
Base* pb = pd;
cout << typeid( pb ).name() << endl;    //prints "class Base *"
cout << typeid( *pb ).name() << endl;    //prints "class Derived"
cout << typeid( pd ).name() << endl;    //prints "class Derived *"
cout << typeid( *pd ).name() << endl;    //prints "class Derived"
delete pd;
}

```

If the *expression* is dereferencing a pointer, and that pointer's value is zero, **typeid** throws a [bad\\_typeid exception](#). If the pointer doesn't point to a valid object, a [\\_\\_non\\_rtti\\_object](#) exception is thrown. It indicates an attempt to analyze the RTTI that triggered a fault because the object is somehow invalid. (For example, it's a bad pointer, or the code wasn't compiled with [/GR](#)).

If the *expression* is not a pointer, and not a reference to a base class of the object, the result is a [type\\_info](#) reference representing the static type of the *expression*. The *static type* of an expression refers to the type of an expression as it is known at compile time. Execution semantics are ignored when evaluating the static type of an expression. Furthermore, references are ignored when possible when determining the static type of an expression:

```

// expre_typeid_Operator_2.cpp
#include <typeinfo>

int main()
{
    typeid(int) == typeid(int&); // evaluates to true
}

```

**typeid** can also be used in templates to determine the type of a template parameter:

```

// expre_typeid_Operator_3.cpp
// compile with: /c
#include <typeinfo>
template < typename T >
T max( T arg1, T arg2 ) {
    cout << typeid( T ).name() << "s compared." << endl;
    return ( arg1 > arg2 ? arg1 : arg2 );
}

```

## See also

[Run-Time Type Information](#)

[Keywords](#)

# Unary Plus and Negation Operators: + and -

---

## Syntax

```
+ cast-expression  
- cast-expression
```

### + operator

The result of the unary plus operator (+) is the value of its operand. The operand to the unary plus operator must be of an arithmetic type.

Integral promotion is performed on integral operands. The resultant type is the type to which the operand is promoted. Thus, the expression `+ch`, where `ch` is of type **char**, results in type **int**; the value is unmodified. See [Standard Conversions](#) for more information about how the promotion is done.

### - operator

The unary negation operator (-) produces the negative of its operand. The operand to the unary negation operator must be an arithmetic type.

Integral promotion is performed on integral operands, and the resultant type is the type to which the operand is promoted. See [Standard Conversions](#) for more information about how the promotion is performed.

### Microsoft Specific

Unary negation of unsigned quantities is performed by subtracting the value of the operand from  $2^n$ , where  $n$  is the number of bits in an object of the given unsigned type.

### END Microsoft Specific

## See also

[Expressions with Unary Operators](#)

[C++ Built-in Operators, Precedence and Associativity](#)



# Expressions (C++)

---

This section describes C++ expressions. Expressions are sequences of operators and operands that are used for one or more of these purposes:

- Computing a value from the operands.
- Designating objects or functions.
- Generating "side effects." (Side effects are any actions other than the evaluation of the expression — for example, modifying the value of an object.)

In C++, operators can be overloaded and their meanings can be user-defined. However, their precedence and the number of operands they take cannot be modified. This section describes the syntax and semantics of operators as they are supplied with the language, not overloaded. In addition to [types of expressions](#) and [semantics of expressions](#), the following topics are covered:

- [Primary expressions](#)
- [Scope resolution operator](#)
- [Postfix expressions](#)
- [Expressions with unary operators](#)
- [Expressions with binary operators](#)
- [Conditional operator](#)
- [Constant expressions](#)
- [Casting operators](#)
- [Run-time type information](#)

Topics on operators in other sections:

- [C++ Built-in Operators, Precedence and Associativity](#)
- [Overloaded operators](#)
- [typeid](#) (C++/CLI)

[!NOTE] Operators for built-in types cannot be overloaded; their behavior is predefined.

See also

[C++ Language Reference](#)

# Types of Expressions

---

C++ expressions are divided into several categories:

- [Primary expressions](#). These are the building blocks from which all other expressions are formed.
- [Postfix expressions](#). These are primary expressions followed by an operator — for example, the array subscript or postfix increment operator.
- [Expressions formed with unary operators](#). Unary operators act on only one operand in an expression.
- [Expressions formed with binary operators](#). Binary operators act on two operands in an expression.
- [Expressions with the conditional operator](#). The conditional operator is a ternary operator — the only such operator in the C++ language — and takes three operands.
- [Constant expressions](#). Constant expressions are formed entirely of constant data.
- [Expressions with explicit type conversions](#). Explicit type conversions, or "casts," can be used in expressions.
- [Expressions with pointer-to-member operators](#).
- [Casting](#). Type-safe "casts" can be used in expressions.
- [Run-Time Type Information](#). Determine the type of an object during program execution.

See also

[Expressions](#)

# Primary Expressions

---

Primary expressions are the building blocks of more complex expressions. They are literals, names, and names qualified by the scope-resolution operator (`::`). A primary expression may have any of the following forms:

```
literal
this
name
::name ( expression )
```

A *literal* is a constant primary expression. Its type depends on the form of its specification. See [Literals](#) for complete information about specifying literals.

The **this** keyword is a pointer to a class object. It is available within nonstatic member functions and points to the instance of the class for which the function was invoked. The **this** keyword cannot be used outside the body of a class-member function.

The type of the **this** pointer is `type *const` (where `type` is the class name) within functions not specifically modifying the **this** pointer. The following example shows member function declarations and the types of **this**:

```
// expre_Primary_Expressions.cpp
// compile with: /LD
class Example
{
public:
    void Func();           // * const this
    void Func() const;     // const * const this
    void Func() volatile;  // volatile * const this
};
```

See [this Pointer](#) for more information about modifying the type of the **this** pointer.

The scope-resolution operator (`::`) followed by a name constitutes a primary expression. Such names must be names at global scope, not member names. The type of this expression is determined by the declaration of the name. It is an l-value (that is, it can appear on the left hand side of an assignment operator expression) if the declaring name is an l-value. The scope-resolution operator allows a global name to be referred to, even if that name is hidden in the current scope. See [Scope](#) for an example of how to use the scope-resolution operator.

An expression enclosed in parentheses is a primary expression whose type and value are identical to those of the unparenthesized expression. It is an l-value if the unparenthesized expression is an l-value.

Examples of primary expressions include:

```
100 // literal
'c' // literal
this // in a member function, a pointer to the class instance
::func // a global function
::operator + // a global operator function
::A::B // a global qualified name
( i + 1 ) // a parenthesized expression
```

The examples below are all considered *names*, and hence primary expressions, in various forms:

```
MyClass // a identifier
MyClass::f // a qualified name
operator = // an operator function name
operator char* // a conversion operator function name
~MyClass // a destructor name
A::B // a qualified name
A<int> // a template id
```

## See also

[Types of Expressions](#)

# Ellipses and Variadic Templates

---

This article shows how to use the ellipsis (...) with C++ variadic templates. The ellipsis has had many uses in C and C++. These include variable argument lists for functions. The `printf()` function from the C Runtime Library is one of the most well-known examples.

A *variadic template* is a class or function template that supports an arbitrary number of arguments. This mechanism is especially useful to C++ library developers because you can apply it to both class templates and function templates, and thereby provide a wide range of type-safe and non-trivial functionality and flexibility.

## Syntax

An ellipsis is used in two ways by variadic templates. To the left of the parameter name, it signifies a *parameter pack*, and to the right of the parameter name, it expands the parameter packs into separate names.

Here's a basic example of *variadic template class* definition syntax:

```
template<typename... Arguments> class classname;
```

For both parameter packs and expansions, you can add whitespace around the ellipsis, based on your preference, as shown in these examples:

```
template<typename ...Arguments> class classname;
```

Or this:

```
template<typename ... Arguments> class classname;
```

Notice that this article uses the convention that's shown in the first example (the ellipsis is attached to `typename`).

In the preceding examples, *Arguments* is a parameter pack. The class `classname` can accept a variable number of arguments, as in these examples:

```
template<typename... Arguments> class vtclass;

vtclass< > vtinstance1;
vtclass<int> vtinstance2;
vtclass<float, bool> vtinstance3;
vtclass<long, std::vector<int>, std::string> vtinstance4;
```

By using a variadic template class definition, you can also require at least one parameter:

```
template <typename First, typename... Rest> class classname;
```

Here's a basic example of *variadic template function* syntax:

```
template <typename... Arguments> returntype functionname(Arguments... args);
```

The *Arguments* parameter pack is then expanded for use, as shown in the next section, **Understanding variadic templates**.

Other forms of variadic template function syntax are possible—including, but not limited to, these examples:

```
template <typename... Arguments> returntype functionname(Arguments&... args);  
template <typename... Arguments> returntype functionname(Arguments&&... args);  
template <typename... Arguments> returntype functionname(Arguments*... args);
```

Specifiers like **const** are also allowed:

```
template <typename... Arguments> returntype functionname(const Arguments&... args);
```

As with variadic template class definitions, you can make functions that require at least one parameter:

```
template <typename First, typename... Rest> returntype functionname(const First& first, const Rest&... args);
```

Variadic templates use the `sizeof...()` operator (unrelated to the older `sizeof()` operator):

```
template<typename... Arguments>  
void tfunc(const Arguments&... args)  
{  
    constexpr auto numargs{ sizeof...(Arguments) };  
  
    X xobj[numargs]; // array of some previously defined type X  
  
    helper_func(xobj, args...);  
}
```

## More about ellipsis placement

Previously, this article described ellipsis placement that defines parameter packs and expansions as "to the left of the parameter name, it signifies a parameter pack, and to the right of the parameter name, it expands the parameter packs into separate names". This is technically true but can be confusing in translation to code. Consider:

- In a template-parameter-list (`template <parameter-list>`), `typename...` introduces a template parameter pack.
- In a parameter-declaration-clause (`func(parameter-list)`), a "top-level" ellipsis introduces a function parameter pack, and the ellipsis positioning is important:

```
// v1 is NOT a function parameter pack:  
template <typename... Types> void func1(std::vector<Types...> v1);  
  
// v2 IS a function parameter pack:  
template <typename... Types> void func2(std::vector<Types>... v2);
```

- Where the ellipsis appears immediately after a parameter name, you have a parameter pack expansion.

## Example

A good way to illustrate the variadic template function mechanism is to use it in a re-write of some of the functionality of `printf`:

```
#include <iostream>  
  
using namespace std;  
  
void print() {  
    cout << endl;  
}  
  
template <typename T> void print(const T& t) {  
    cout << t << endl;  
}  
  
template <typename First, typename... Rest> void print(const First& first, const  
Rest&... rest) {  
    cout << first << ", ";  
    print(rest...); // recursive call using pack expansion syntax  
}  
  
int main()  
{  
    print(); // calls first overload, outputting only a newline  
    print(1); // calls second overload  
  
    // these call the third overload, the variadic template,  
    // which uses recursion as needed.
```

```
print(10, 20);  
print(100, 200, 300);  
print("first", 2, "third", 3.14159);  
}
```

## Output

```
1  
10, 20  
100, 200, 300  
first, 2, third, 3.14159
```

[!NOTE] Most implementations that incorporate variadic template functions use recursion of some form, but it's slightly different from traditional recursion. Traditional recursion involves a function calling itself by using the same signature. (It may be overloaded or templated, but the same signature is chosen each time.) Variadic recursion involves calling a variadic function template by using differing (almost always decreasing) numbers of arguments, and thereby stamping out a different signature every time. A "base case" is still required, but the nature of the recursion is different.



# Postfix Expressions

Postfix expressions consist of primary expressions or expressions in which postfix operators follow a primary expression. The postfix operators are listed in the following table.

## Postfix Operators

Operator Name	Operator Notation
Subscript operator	[ ]
Function call operator	( )
Explicit type conversion operator	<i>type-name</i> ( )
Member access operator	. or ->
Postfix increment operator	++
Postfix decrement operator	--

The following syntax describes possible postfix expressions:

```
primary-expression
postfix-expression[expression]postfix-expression(expression-list)simple-type-
name(expression-list)postfix-expression.namepostfix-expression->namepostfix-
expression++postfix-expression--cast-keyword < typename > (expression )typeid (
typename )
```

The *postfix-expression* above may be a primary expression or another postfix expression. See **primary expressions**. Postfix expressions group left to right, thus allowing the expressions to be chained together as follows:

```
func(1)->GetValue()++
```

In the above expression, `func` is a primary expression, `func(1)` is a function postfix expression, `func(1)->GetValue` is a postfix expression specifying a member of the class, `func(1)->GetValue()` is another function postfix expression, and the entire expression is a postfix expression incrementing the return value of `GetValue`. The meaning of the expression as a whole is "call `func` passing 1 as an argument and get a pointer to a class as a return value. Then call `GetValue()` on that class, then increment the value returned.

The expressions listed above are assignment expressions, meaning that the result of these expressions must be an r-value.

The postfix expression form

```
simple-type-name ( expression-list )
```

indicates the invocation of the constructor. If the simple-type-name is a fundamental type, the expression list must be a single expression, and this expression indicates a cast of the expression's value to the fundamental type. This type of cast expression mimics a constructor. Because this form allows fundamental types and classes to be constructed using the same syntax, this form is especially useful when defining template classes.

The *cast-keyword* is one of **dynamic\_cast**, **static\_cast** or **reinterpret\_cast**. More information may be found in **dynamic\_cast**, **static\_cast** and **reinterpret\_cast**.

The **typeid** operator is considered a postfix expression. See **typeid operator**.

## Formal and actual arguments

Calling programs pass information to called functions in "actual arguments." The called functions access the information using corresponding "formal arguments."

When a function is called, the following tasks are performed:

- All actual arguments (those supplied by the caller) are evaluated. There is no implied order in which these arguments are evaluated, but all arguments are evaluated and all side effects completed prior to entry to the function.
- Each formal argument is initialized with its corresponding actual argument in the expression list. (A formal argument is an argument that is declared in the function header and used in the body of a function.) Conversions are done as if by initialization — both standard and user-defined conversions are performed in converting an actual argument to the correct type. The initialization performed is illustrated conceptually by the following code:

```
void Func( int i ); // Function prototype
...
Func( 7 );          // Execute function call
```

The conceptual initializations prior to the call are:

```
int Temp_i = 7;
Func( Temp_i );
```

Note that the initialization is performed as if using the equal-sign syntax instead of the parentheses syntax. A copy of `i` is made prior to passing the value to the function. (For more information, see [Initializers](#) and [Conversions](#)).

Therefore, if the function prototype (declaration) calls for an argument of type **long**, and if the calling program supplies an actual argument of type **int**, the actual argument is promoted using a standard type conversion to type **long** (see [Standard Conversions](#)).

It is an error to supply an actual argument for which there is no standard or user-defined conversion to the type of the formal argument.

For actual arguments of class type, the formal argument is initialized by calling the class's constructor. (See [Constructors](#) for more about these special class member functions.)

- The function call is executed.

The following program fragment demonstrates a function call:

```
// expre_Formal_and_Actual_Arguments.cpp
void func( long param1, double param2 );

int main()
{
    long i = 1;
    double j = 2;

    // Call func with actual arguments i and j.
    func( i, j );
}

// Define func with formal parameters param1 and param2.
void func( long param1, double param2 )
{
}
```

When `func` is called from `main`, the formal parameter `param1` is initialized with the value of `i` (`i` is converted to type **long** to correspond to the correct type using a standard conversion), and the formal parameter `param2` is initialized with the value of `j` (`j` is converted to type **double** using a standard conversion).

## Treatment of argument types

Formal arguments declared as `const` types cannot be changed within the body of a function. Functions can change any argument that is not of type **const**. However, the change is local to the function and does not affect the actual argument's value unless the actual argument was a reference to an object not of type **const**.

The following functions illustrate some of these concepts:

```
// expre_Treatment_of_Argument_Types.cpp
int func1( const int i, int j, char *c ) {
    i = 7;    // C3892 i is const.
    j = i;    // value of j is lost at return
    *c = 'a' + j;    // changes value of c in calling function
    return i;
}

double& func2( double& d, const char *c ) {
    d = 14.387;    // changes value of d in calling function.
    *c = 'a';    // C3892 c is a pointer to a const object.
}
```

```
    return d;
}
```

## Ellipses and default arguments

Functions can be declared to accept fewer arguments than specified in the function definition, using one of two methods: ellipsis (...) or default arguments.

Ellipses denote that arguments may be required but that the number and types are not specified in the declaration. This is normally poor C++ programming practice because it defeats one of the benefits of C++: type safety. Different conversions are applied to functions declared with ellipses than to those functions for which the formal and actual argument types are known:

- If the actual argument is of type **float**, it is promoted to type **double** prior to the function call.
- Any signed or unsigned **char**, **short**, enumerated type, or bit field is converted to either a signed or an unsigned **int** using integral promotion.
- Any argument of class type is passed by value as a data structure; the copy is created by binary copying instead of by invoking the class's copy constructor (if one exists).

Ellipses, if used, must be declared last in the argument list. For more information about passing a variable number of arguments, see the discussion of [va\\_arg](#), [va\\_start](#), and [va\\_list](#) in the *Run-Time Library Reference*.

For information on default arguments in CLR programming, see [Variable Argument Lists \(...\) \(C++/CLI\)](#).

Default arguments enable you to specify the value an argument should assume if none is supplied in the function call. The following code fragment shows how default arguments work. For more information about restrictions on specifying default arguments, see [Default Arguments](#).

```
// expre_Ellipses_and_Default_Arguments.cpp
// compile with: /EHsc
#include <iostream>

// Declare the function print that prints a string,
// then a terminator.
void print( const char *string,
           const char *terminator = "\n" );

int main()
{
    print( "hello," );
    print( "world!" );

    print( "good morning", ", " );
    print( "sunshine." );
}

using namespace std;
// Define print.
void print( const char *string, const char *terminator )
```

```
{  
    if( string != NULL )  
        cout << string;  
  
    if( terminator != NULL )  
        cout << terminator;  
}
```

The preceding program declares a function, `print`, that takes two arguments. However, the second argument, *terminator*, has a default value, `"\n"`. In `main`, the first two calls to `print` allow the default second argument to supply a new line to terminate the printed string. The third call specifies an explicit value for the second argument. The output from the program is

```
hello,  
world!  
good morning, sunshine.
```

## See also

[Types of Expressions](#)

# Expressions with Unary Operators

---

Unary operators act on only one operand in an expression. The unary operators are as follows:

- Indirection operator (\*)
- Address-of operator (&)
- Unary plus operator (+)
- Unary negation operator (-)
- Logical negation operator (!)
- One's complement operator (~)
- Prefix increment operator (++)
- Prefix decrement operator (--)
- Cast operator ()
- sizeof operator
- \_\_typeof operator
- \_\_alignof operator
- new operator
- delete operator

These operators have right-to-left associativity. Unary expressions generally involve syntax that precedes a postfix or primary expression.

The following are the possible forms of unary expressions.

- *postfix-expression*
- **++** *unary-expression*
- **--** *unary-expression*
- *unary-operator cast-expression*
- **sizeof** *unary-expression*
- **sizeof**( *type-name* )
- **decltype**( *expression* )
- *allocation-expression*
- *deallocation-expression*

Any *postfix-expression* is considered a *unary-expression*, and because any primary expression is considered a *postfix-expression*, any primary expressions is considered a *unary-expression* also. For more information, see [Postfix Expressions](#) and [Primary Expressions](#).

A *unary-operator* consists of one or more of the following symbols: \* & + - ! ~

The *cast-expression* is a unary expression with an optional cast to change the type. For more information see [Cast Operator: \(\)](#).

An *expression* can be any expression. For more information, see [Expressions](#).

The *allocation-expression* refers to the **new** operator. The *deallocation-expression* refers to the **delete** operator. For more information, see the links earlier in this topic.

## See also

[Types of Expressions](#)

# Expressions with Binary Operators

---

Binary operators act on two operands in an expression. The binary operators are:

- **Multiplicative operators**
  - Multiplication (\*)
  - Division (/)
  - Modulus (%)
- **Additive operators**
  - Addition (+)
  - Subtraction (-)
- **Shift operators**
  - Right shift (>>)
  - Left shift (<<)
- **Relational and equality operators**
  - Less than (<)
  - Greater than (>)
  - Less than or equal to (<=)
  - Greater than or equal to (>=)
  - Equal to (==)
  - Not equal to (!=)
- **Bitwise operators**
  - **Bitwise AND (&)**
  - **Bitwise exclusive OR (^)**
  - **Bitwise inclusive OR (|)**
- **Logical operators**
  - **Logical AND (&&)**
  - **Logical OR (||)**
- **Assignment operators**



- Assignment (=)
  - Addition assignment (+=)
  - Subtraction assignment (-=)
  - Multiplication assignment (\*=)
  - Division assignment (/=)
  - Modulus assignment (%=)
  - Left shift assignment (<<=)
  - Right shift assignment (>>=)
  - Bitwise AND assignment (&=)
  - Bitwise exclusive OR assignment (^=)
  - Bitwise inclusive OR assignment (|=)
- [Comma Operator](#) (,)

See also

[Types of Expressions](#)

# C++ Constant Expressions

---

A *constant* value is one that doesn't change. C++ provides two keywords to enable you to express the intent that an object is not intended to be modified, and to enforce that intent.

C++ requires constant expressions — expressions that evaluate to a constant — for declarations of:

- Array bounds
- Selectors in case statements
- Bit-field length specification
- Enumeration initializers

The only operands that are legal in constant expressions are:

- Literals
- Enumeration constants
- Values declared as `const` that are initialized with constant expressions
- **`sizeof`** expressions

Nonintegral constants must be converted (either explicitly or implicitly) to integral types to be legal in a constant expression. Therefore, the following code is legal:

```
const double Size = 11.0;
char chArray[(int)Size];
```

Explicit conversions to integral types are legal in constant expressions; all other types and derived types are illegal except when used as operands to the **`sizeof`** operator.

The comma operator and assignment operators cannot be used in constant expressions.

## See also

[Types of Expressions](#)

# Semantics of Expressions

---

Expressions are evaluated according to the precedence and grouping of their operators. ([Operator Precedence and Associativity](#) in [Lexical Conventions](#), shows the relationships the C++ operators impose on expressions.)

## Order of evaluation

Consider this example:

```
// Order_of_Evaluation.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;
int main()
{
    int a = 2, b = 4, c = 9;

    cout << a + b * c << "\n";
    cout << a + (b * c) << "\n";
    cout << (a + b) * c << "\n";
}
```

```
38
38
54
```

 Evaluation order in an expression

Expression-evaluation order

The order in which the expression shown in the above figure is evaluated is determined by the precedence and associativity of the operators:

1. Multiplication (\*) has the highest precedence in this expression; hence the subexpression `b * c` is evaluated first.
2. Addition (+) has the next highest precedence, so `a` is added to the product of `b` and `c`.
3. Left shift (<<) has the lowest precedence in the expression, but there are two occurrences. Because the left-shift operator groups left-to-right, the left subexpression is evaluated first and then the right one.

When parentheses are used to group the subexpressions, they alter the precedence and also the order in which the expression is evaluated, as shown in the following figure.

 Evaluation order of expression with parentheses

Expression-evaluation order with parentheses

Expressions such as those in the above figure are evaluated purely for their side effects — in this case, to transfer information to the standard output device.

## Notation in expressions

The C++ language specifies certain compatibilities when specifying operands. The following table shows the types of operands acceptable to operators that require operands of type *type*.

Operand Types Acceptable to Operators

Type expected	Types allowed
<i>type</i>	<code>const type</code> <code>volatile type</code> <code>type&amp;</code> <code>const type&amp;</code> <code>volatile type&amp;</code> <code>volatile const type</code> <code>volatile const type&amp;</code>
<i>type *</i>	<code>type *</code> <code>const type *</code> <code>volatile type *</code> <code>volatile const type *</code>
<code>const type</code>	<code>type</code> <code>const type</code> <code>const type&amp;</code>
<code>volatile type</code>	<code>type</code> <code>volatile type</code> <code>volatile type&amp;</code>

Because the preceding rules can always be used in combination, a `const` pointer to a volatile object can be supplied where a pointer is expected.

## Ambiguous expressions

Certain expressions are ambiguous in their meaning. These expressions occur most frequently when an object's value is modified more than once in the same expression. These expressions rely on a particular order of evaluation where the language does not define one. Consider the following example:

```
int i = 7;

func( i, ++i );
```

The C++ language does not guarantee the order in which arguments to a function call are evaluated. Therefore, in the preceding example, `func` could receive the values 7 and 8, or 8 and 8 for its parameters,

depending on whether the parameters are evaluated from left to right or from right to left.

## C++ sequence points (Microsoft-specific)

An expression can modify an object's value only once between consecutive "sequence points."

The C++ language definition does not currently specify sequence points. Microsoft C++ uses the same sequence points as ANSI C for any expression involving C operators and not involving overloaded operators. When operators are overloaded, the semantics change from operator sequencing to function-call sequencing. Microsoft C++ uses the following sequence points:

- Left operand of the logical AND operator (&&). The left operand of the logical AND operator is completely evaluated and all side effects completed before continuing. There is no guarantee that the right operand of the logical AND operator will be evaluated.
- Left operand of the logical OR operator (||). The left operand of the logical OR operator is completely evaluated and all side effects completed before continuing. There is no guarantee that the right operand of the logical OR operator will be evaluated.
- Left operand of the comma operator. The left operand of the comma operator is completely evaluated and all side effects completed before continuing. Both operands of the comma operator are always evaluated.
- Function-call operator. The function-call expression and all arguments to a function, including default arguments, are evaluated and all side effects completed prior to entry to the function. There is no specified order of evaluation among the arguments or the function-call expression.
- First operand of the conditional operator. The first operand of the conditional operator is completely evaluated and all side effects completed before continuing.
- The end of a full initialization expression, such as the end of an initialization in a declaration statement.
- The expression in an expression statement. Expression statements consist of an optional expression followed by a semicolon (;). The expression is completely evaluated for its side effects.
- The controlling expression in a selection (if or switch) statement. The expression is completely evaluated and all side effects completed before the code dependent on the selection is executed.
- The controlling expression of a while or do statement. The expression is completely evaluated and all side effects completed before any statements in the next iteration of the while or do loop are executed.
- Each of the three expressions of a for statement. Each expression is completely evaluated and all side effects completed before moving to the next expression.
- The expression in a return statement. The expression is completely evaluated and all side effects completed before control returns to the calling function.

See also

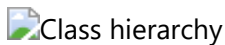
[Expressions](#)

# Casting

---

The C++ language provides that if a class is derived from a base class containing virtual functions, a pointer to that base class type can be used to call the implementations of the virtual functions residing in the derived class object. A class containing virtual functions is sometimes called a "polymorphic class."

Since a derived class completely contains the definitions of all the base classes from which it is derived, it is safe to cast a pointer up the class hierarchy to any of these base classes. Given a pointer to a base class, it might be safe to cast the pointer down the hierarchy. It is safe if the object being pointed to is actually of a type derived from the base class. In this case, the actual object is said to be the "complete object." The pointer to the base class is said to point to a "subobject" of the complete object. For example, consider the class hierarchy shown in the following figure.



Class hierarchy

Class hierarchy

An object of type **C** could be visualized as shown in the following figure.



Class C with sub-objects B and A

Class C with sub-objects B and A

Given an instance of class **C**, there is a **B** subobject and an **A** subobject. The instance of **C**, including the **A** and **B** subobjects, is the "complete object."

Using run-time type information, it is possible to check whether a pointer actually points to a complete object and can be safely cast to point to another object in its hierarchy. The `dynamic_cast` operator can be used to make these types of casts. It also performs the run-time check necessary to make the operation safe.

For conversion of nonpolymorphic types, you can use the `static_cast` operator (this topic explains the difference between static and dynamic casting conversions, and when it is appropriate to use each).

This section covers the following topics:

- [Casting operators](#)
- [Run-time type information](#)

## See also

[Expressions](#)

# Casting Operators

---

There are several casting operators specific to the C++ language. These operators are intended to remove some of the ambiguity and danger inherent in old style C language casts. These operators are:

- [dynamic\\_cast](#) Used for conversion of polymorphic types.
- [static\\_cast](#) Used for conversion of nonpolymorphic types.
- [const\\_cast](#) Used to remove the **const**, **volatile**, and **\_\_unaligned** attributes.
- [reinterpret\\_cast](#) Used for simple reinterpretation of bits.
- [safe\\_cast](#) Used in C++/CLI to produce verifiable MSIL.

Use **const\_cast** and **reinterpret\_cast** as a last resort, since these operators present the same dangers as old style casts. However, they are still necessary in order to completely replace old style casts.

See also

[Casting](#)

# dynamic\_cast Operator

---

Converts the operand `expression` to an object of type `type-id`.

## Syntax

```
dynamic_cast < type-id > ( expression )
```

## Remarks

The `type-id` must be a pointer or a reference to a previously defined class type or a "pointer to void". The type of `expression` must be a pointer if `type-id` is a pointer, or an l-value if `type-id` is a reference.

See [static\\_cast](#) for an explanation of the difference between static and dynamic casting conversions, and when it is appropriate to use each.

There are two breaking changes in the behavior of **dynamic\_cast** in managed code:

- **dynamic\_cast** to a pointer to the underlying type of a boxed enum will fail at runtime, returning 0 instead of the converted pointer.
- **dynamic\_cast** will no longer throw an exception when `type-id` is an interior pointer to a value type, with the cast failing at runtime. The cast will now return the 0 pointer value instead of throwing.

If `type-id` is a pointer to an unambiguous accessible direct or indirect base class of `expression`, a pointer to the unique subobject of type `type-id` is the result. For example:

```
// dynamic_cast_1.cpp
// compile with: /c
class B { };
class C : public B { };
class D : public C { };

void f(D* pd) {
    C* pc = dynamic_cast<C*>(pd);    // ok: C is a direct base class
                                    // pc points to C subobject of pd
    B* pb = dynamic_cast<B*>(pd);    // ok: B is an indirect base class
                                    // pb points to B subobject of pd
}
```

This type of conversion is called an "upcast" because it moves a pointer up a class hierarchy, from a derived class to a class it is derived from. An upcast is an implicit conversion.

If `type-id` is `void*`, a run-time check is made to determine the actual type of `expression`. The result is a pointer to the complete object pointed to by `expression`. For example:



```
// dynamic_cast_2.cpp
// compile with: /c /GR
class A {virtual void f();};
class B {virtual void f();};

void f() {
    A* pa = new A;
    B* pb = new B;
    void* pv = dynamic_cast<void*>(pa);
    // pv now points to an object of type A

    pv = dynamic_cast<void*>(pb);
    // pv now points to an object of type B
}
```

If **type-id** is not `void*`, a run-time check is made to see if the object pointed to by **expression** can be converted to the type pointed to by **type-id**.

If the type of **expression** is a base class of the type of **type-id**, a run-time check is made to see if **expression** actually points to a complete object of the type of **type-id**. If this is true, the result is a pointer to a complete object of the type of **type-id**. For example:

```
// dynamic_cast_3.cpp
// compile with: /c /GR
class B {virtual void f();};
class D : public B {virtual void f();};

void f() {
    B* pb = new D;    // unclear but ok
    B* pb2 = new B;

    D* pd = dynamic_cast<D*>(pb);    // ok: pb actually points to a D
    D* pd2 = dynamic_cast<D*>(pb2);  // pb2 points to a B not a D
}
```

This type of conversion is called a "downcast" because it moves a pointer down a class hierarchy, from a given class to a class derived from it.

In cases of multiple inheritance, possibilities for ambiguity are introduced. Consider the class hierarchy shown in the following figure.

For CLR types, **dynamic\_cast** results in either a no-op if the conversion can be performed implicitly, or an MSIL **isinst** instruction, which performs a dynamic check and returns **nullptr** if the conversion fails.

The following sample uses **dynamic\_cast** to determine if a class is an instance of particular type:

```
// dynamic_cast_clr.cpp
// compile with: /clr
```

```


using namespace System;

void PrintObjectType( Object^o ) {
    if( dynamic_cast<String^>(o) )
        Console::WriteLine("Object is a String");
    else if( dynamic_cast<int^>(o) )
        Console::WriteLine("Object is an int");
}

int main() {
    Object^o1 = "hello";
    Object^o2 = 10;

    PrintObjectType(o1);
    PrintObjectType(o2);
}

```

 Class hierarchy that shows multiple inheritance

Class hierarchy that shows multiple inheritance

A pointer to an object of type **D** can be safely cast to **B** or **C**. However, if **D** is cast to point to an **A** object, which instance of **A** would result? This would result in an ambiguous casting error. To get around this problem, you can perform two unambiguous casts. For example:


```

// dynamic_cast_4.cpp
// compile with: /c /GR
class A {virtual void f();};
class B : public A {virtual void f();};
class C : public A {virtual void f();};
class D : public B, public C {virtual void f();};

void f() {
    D* pd = new D;
    A* pa = dynamic_cast<A*>(pd);    // C4540, ambiguous cast fails at runtime
    B* pb = dynamic_cast<B*>(pd);    // first cast to B
    A* pa2 = dynamic_cast<A*>(pb);   // ok: unambiguous
}

```

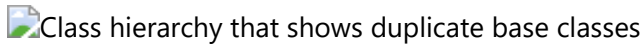
Further ambiguities can be introduced when you use virtual base classes. Consider the class hierarchy shown in the following figure.

 Class hierarchy that shows virtual base classes

Class hierarchy that shows virtual base classes

In this hierarchy, **A** is a virtual base class. Given an instance of class **E** and a pointer to the **A** subobject, a **dynamic\_cast** to a pointer to **B** will fail due to ambiguity. You must first cast back to the complete **E** object, then work your way back up the hierarchy, in an unambiguous manner, to reach the correct **B** object.

Consider the class hierarchy shown in the following figure.



Class hierarchy that shows duplicate base classes

Class hierarchy that shows duplicate base classes

Given an object of type **E** and a pointer to the **D** subobject, to navigate from the **D** subobject to the left-most **A** subobject, three conversions can be made. You can perform a **dynamic\_cast** conversion from the **D** pointer to an **E** pointer, then a conversion (either **dynamic\_cast** or an implicit conversion) from **E** to **B**, and finally an implicit conversion from **B** to **A**. For example:

```
// dynamic_cast_5.cpp
// compile with: /c /GR
class A {virtual void f();};
class B : public A {virtual void f();};
class C : public A { };
class D {virtual void f();};
class E : public B, public C, public D {virtual void f();};

void f(D* pd) {
    E* pe = dynamic_cast<E*>(pd);
    B* pb = pe;    // upcast, implicit conversion
    A* pa = pb;    // upcast, implicit conversion
}
```

The **dynamic\_cast** operator can also be used to perform a "cross cast." Using the same class hierarchy, it is possible to cast a pointer, for example, from the **B** subobject to the **D** subobject, as long as the complete object is of type **E**.

Considering cross casts, it is actually possible to do the conversion from a pointer to **D** to a pointer to the left-most **A** subobject in just two steps. You can perform a cross cast from **D** to **B**, then an implicit conversion from **B** to **A**. For example:

```
// dynamic_cast_6.cpp
// compile with: /c /GR
class A {virtual void f();};
class B : public A {virtual void f();};
class C : public A { };
class D {virtual void f();};
class E : public B, public C, public D {virtual void f();};

void f(D* pd) {
    B* pb = dynamic_cast<B*>(pd);    // cross cast
    A* pa = pb;    // upcast, implicit conversion
}
```

A null pointer value is converted to the null pointer value of the destination type by **dynamic\_cast**.

When you use **dynamic\_cast < type-id > ( expression )**, if **expression** cannot be safely converted to type **type-id**, the run-time check causes the cast to fail. For example:

```

// dynamic_cast_7.cpp
// compile with: /c /GR
class A {virtual void f();};
class B {virtual void f();};

void f() {
    A* pa = new A;
    B* pb = dynamic_cast<B*>(pa);    // fails at runtime, not safe;
    // B not derived from A
}

```

The value of a failed cast to pointer type is the null pointer. A failed cast to reference type throws a [bad\\_cast Exception](#). If `expression` does not point to or reference a valid object, a `__non_rtti_object` exception is thrown.

See [typeid](#) for an explanation of the `__non_rtti_object` exception.

## Example

The following sample creates the base class (struct A) pointer, to an object (struct C). This, plus the fact there are virtual functions, enables runtime polymorphism.

The sample also calls a non-virtual function in the hierarchy.

```

// dynamic_cast_8.cpp
// compile with: /GR /EHsc
#include <stdio.h>
#include <iostream>

struct A {
    virtual void test() {
        printf_s("in A\n");
    }
};

struct B : A {
    virtual void test() {
        printf_s("in B\n");
    }

    void test2() {
        printf_s("test2 in B\n");
    }
};

struct C : B {
    virtual void test() {
        printf_s("in C\n");
    }
}

```

```

    void test2() {
        printf_s("test2 in C\n");
    }
};

void Globaltest(A& a) {
    try {
        C &c = dynamic_cast<C&>(a);
        printf_s("in GlobalTest\n");
    }
    catch(std::bad_cast) {
        printf_s("Can't cast to C\n");
    }
}

int main() {
    A *pa = new C;
    A *pa2 = new B;

    pa->test();

    B * pb = dynamic_cast<B *>(pa);
    if (pb)
        pb->test2();

    C * pc = dynamic_cast<C *>(pa2);
    if (pc)
        pc->test2();

    C ConStack;
    Globaltest(ConStack);

    // will fail because B knows nothing about C
    B BonStack;
    Globaltest(BonStack);
}

```

```

in C
test2 in B
in GlobalTest
Can't cast to C

```

## See also

[Casting Operators](#)

[Keywords](#)

# bad\_cast exception

---

The **bad\_cast** exception is thrown by the **dynamic\_cast** operator as the result of a failed cast to a reference type.

## Syntax

```
catch (bad_cast)
    statement
```

## Remarks

The interface for **bad\_cast** is:

```
class bad_cast : public exception
```

The following code contains an example of a failed **dynamic\_cast** that throws the **bad\_cast** exception.

```
// expre_bad_cast_Exception.cpp
// compile with: /EHsc /GR
#include <typeinfo>
#include <iostream>

class Shape {
public:
    virtual void virtualfunc() const {}
};

class Circle: public Shape {
public:
    virtual void virtualfunc() const {}
};

using namespace std;
int main() {
    Shape shape_instance;
    Shape& ref_shape = shape_instance;
    try {
        Circle& ref_circle = dynamic_cast<Circle&>(ref_shape);
    }
    catch (bad_cast b) {
        cout << "Caught: " << b.what();
    }
}
```

The exception is thrown because the object being cast (a Shape) isn't derived from the specified cast type (Circle). To avoid the exception, add these declarations to `main`:

```
Circle circle_instance;  
Circle& ref_circle = circle_instance;
```

Then reverse the sense of the cast in the **try** block as follows:

```
Shape& ref_shape = dynamic_cast<Shape*>(ref_circle);
```

## Members

### Constructors

Constructor	Description
<code>bad_cast</code>	The constructor for objects of type <code>bad_cast</code> .

### Functions

Function	Description
<code>what</code>	TBD

### Operators

Operator	Description
<code>operator=</code>	An assignment operator that assigns one <code>bad_cast</code> object to another.

## bad\_cast

The constructor for objects of type `bad_cast`.

```
bad_cast(const char * _Message = "bad cast");  
bad_cast(const bad_cast &);
```

## operator=

An assignment operator that assigns one `bad_cast` object to another.

```
bad_cast& operator=(const bad_cast&) noexcept;
```

## what

```
const char* what() const noexcept override;
```

## See also

[dynamic\\_cast Operator](#)

[Keywords](#)

[Modern C++ best practices for exceptions and error handling](#)



# static\_cast Operator

---

Converts an *expression* to the type of *type-id*, based only on the types that are present in the expression.

## Syntax

```
static_cast <type-id> ( expression )
```

## Remarks

In standard C++, no run-time type check is made to help ensure the safety of the conversion. In C++/CX, a compile time and runtime check are performed. For more information, see [Casting](#).

The **static\_cast** operator can be used for operations such as converting a pointer to a base class to a pointer to a derived class. Such conversions are not always safe.

In general you use **static\_cast** when you want to convert numeric data types such as enums to ints or ints to floats, and you are certain of the data types involved in the conversion. **static\_cast** conversions are not as safe as **dynamic\_cast** conversions, because **static\_cast** does no run-time type check, while **dynamic\_cast** does. A **dynamic\_cast** to an ambiguous pointer will fail, while a **static\_cast** returns as if nothing were wrong; this can be dangerous. Although **dynamic\_cast** conversions are safer, **dynamic\_cast** only works on pointers or references, and the run-time type check is an overhead. For more information, see [dynamic\\_cast Operator](#).

In the example that follows, the line `D* pd2 = static_cast<D*>(pb);` is not safe because `D` can have fields and methods that are not in `B`. However, the line `B* pb2 = static_cast<B*>(pd);` is a safe conversion because `D` always contains all of `B`.

```
// static_cast_Operator.cpp
// compile with: /LD
class B {};

class D : public B {};

void f(B* pb, D* pd) {
    D* pd2 = static_cast<D*>(pb);    // Not safe, D can have fields
                                    // and methods that are not in B.

    B* pb2 = static_cast<B*>(pd);    // Safe conversion, D always
                                    // contains all of B.
}
```

In contrast to [dynamic\\_cast](#), no run-time check is made on the **static\_cast** conversion of `pb`. The object pointed to by `pb` may not be an object of type `D`, in which case the use of `*pd2` could be disastrous. For instance, calling a function that is a member of the `D` class, but not the `B` class, could result in an access violation.

The **dynamic\_cast** and **static\_cast** operators move a pointer throughout a class hierarchy. However, **static\_cast** relies exclusively on the information provided in the cast statement and can therefore be unsafe. For example:

```
// static_cast_Operator_2.cpp
// compile with: /LD /GR
class B {
public:
    virtual void Test(){}
};
class D : public B {};

void f(B* pb) {
    D* pd1 = dynamic_cast<D*>(pb);
    D* pd2 = static_cast<D*>(pb);
}
```

If **pb** really points to an object of type **D**, then **pd1** and **pd2** will get the same value. They will also get the same value if **pb == 0**.

If **pb** points to an object of type **B** and not to the complete **D** class, then **dynamic\_cast** will know enough to return zero. However, **static\_cast** relies on the programmer's assertion that **pb** points to an object of type **D** and simply returns a pointer to that supposed **D** object.

Consequently, **static\_cast** can do the inverse of implicit conversions, in which case the results are undefined. It is left to the programmer to verify that the results of a **static\_cast** conversion are safe.

This behavior also applies to types other than class types. For instance, **static\_cast** can be used to convert from an **int** to a **char**. However, the resulting **char** may not have enough bits to hold the entire **int** value. Again, it is left to the programmer to verify that the results of a **static\_cast** conversion are safe.

The **static\_cast** operator can also be used to perform any implicit conversion, including standard conversions and user-defined conversions. For example:

```
// static_cast_Operator_3.cpp
// compile with: /LD /GR
typedef unsigned char BYTE;

void f() {
    char ch;
    int i = 65;
    float f = 2.5;
    double dbl;

    ch = static_cast<char>(i);    // int to char
    dbl = static_cast<double>(f); // float to double
    i = static_cast<BYTE>(ch);
}
```

The **static\_cast** operator can explicitly convert an integral value to an enumeration type. If the value of the integral type does not fall within the range of enumeration values, the resulting enumeration value is undefined.

The **static\_cast** operator converts a null pointer value to the null pointer value of the destination type.

Any expression can be explicitly converted to type void by the **static\_cast** operator. The destination void type can optionally include the **const**, **volatile**, or **\_\_unaligned** attribute.

The **static\_cast** operator cannot cast away the **const**, **volatile**, or **\_\_unaligned** attributes. See [const\\_cast Operator](#) for information on removing these attributes.

**C++/CLI:** Due to the danger of performing unchecked casts on top of a relocating garbage collector, the use of **static\_cast** should only be in performance-critical code when you are certain it will work correctly. If you must use **static\_cast** in release mode, substitute it with [safe\\_cast](#) in your debug builds to ensure success.

## See also

[Casting Operators](#)

[Keywords](#)

# const\_cast Operator

---

Removes the **const**, **volatile**, and **\_\_unaligned** attribute(s) from a class.

## Syntax

```
const_cast <type-id> (expression)
```

## Remarks

A pointer to any object type or a pointer to a data member can be explicitly converted to a type that is identical except for the **const**, **volatile**, and **\_\_unaligned** qualifiers. For pointers and references, the result will refer to the original object. For pointers to data members, the result will refer to the same member as the original (uncast) pointer to data member. Depending on the type of the referenced object, a write operation through the resulting pointer, reference, or pointer to data member might produce undefined behavior.

You cannot use the **const\_cast** operator to directly override a constant variable's constant status.

The **const\_cast** operator converts a null pointer value to the null pointer value of the destination type.

## Example

```
// expre_const_cast_Operator.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;
class CCTest {
public:
    void setNumber( int );
    void printNumber() const;
private:
    int number;
};

void CCTest::setNumber( int num ) { number = num; }

void CCTest::printNumber() const {
    cout << "\nBefore: " << number;
    const_cast< CCTest * >( this )->number--;
    cout << "\nAfter: " << number;
}

int main() {
    CCTest X;
    X.setNumber( 8 );
}
```

```
X.printNumber();  
}
```

On the line containing the **const\_cast**, the data type of the **this** pointer is `const CCTest *`. The **const\_cast** operator changes the data type of the **this** pointer to `CCTest *`, allowing the member `number` to be modified. The cast lasts only for the remainder of the statement in which it appears.

## See also

[Casting Operators](#)

[Keywords](#)

# reinterpret\_cast Operator

---

Allows any pointer to be converted into any other pointer type. Also allows any integral type to be converted into any pointer type and vice versa.

## Syntax

```
reinterpret_cast < type-id > ( expression )
```

## Remarks

Misuse of the **reinterpret\_cast** operator can easily be unsafe. Unless the desired conversion is inherently low-level, you should use one of the other cast operators.

The **reinterpret\_cast** operator can be used for conversions such as `char*` to `int*`, or `One_class*` to `Unrelated_class*`, which are inherently unsafe.

The result of a **reinterpret\_cast** cannot safely be used for anything other than being cast back to its original type. Other uses are, at best, nonportable.

The **reinterpret\_cast** operator cannot cast away the **const**, **volatile**, or **\_\_unaligned** attributes. See [const\\_cast Operator](#) for information on removing these attributes.

The **reinterpret\_cast** operator converts a null pointer value to the null pointer value of the destination type.

One practical use of **reinterpret\_cast** is in a hash function, which maps a value to an index in such a way that two distinct values rarely end up with the same index.

```
#include <iostream>
using namespace std;

// Returns a hash code based on an address
unsigned short Hash( void *p ) {
    unsigned int val = reinterpret_cast<unsigned int>( p );
    return ( unsigned short )( val ^ (val >> 16));
}

using namespace std;
int main() {
    int a[20];
    for ( int i = 0; i < 20; i++ )
        cout << Hash( a + i ) << endl;
}
```

Output:

```
64641
64645
```

```
64889
64893
64881
64885
64873
64877
64865
64869
64857
64861
64849
64853
64841
64845
64833
64837
64825
64829
```

The **`reinterpret_cast`** allows the pointer to be treated as an integral type. The result is then bit-shifted and XORed with itself to produce a unique index (unique to a high degree of probability). The index is then truncated by a standard C-style cast to the return type of the function.

## See also

[Casting Operators](#)

[Keywords](#)

# Run-Time Type Information

---

Run-time type information (RTTI) is a mechanism that allows the type of an object to be determined during program execution. RTTI was added to the C++ language because many vendors of class libraries were implementing this functionality themselves. This caused incompatibilities between libraries. Thus, it became obvious that support for run-time type information was needed at the language level.

For the sake of clarity, this discussion of RTTI is almost completely restricted to pointers. However, the concepts discussed also apply to references.

There are three main C++ language elements to run-time type information:

- The [dynamic\\_cast](#) operator.

Used for conversion of polymorphic types.

- The [typeid](#) operator.

Used for identifying the exact type of an object.

- The [type\\_info](#) class.

Used to hold the type information returned by the **typeid** operator.

## See also

[Casting](#)



# bad\_typeid exception

---

The **bad\_typeid** exception is thrown by the [typeid operator](#) when the operand for **typeid** is a NULL pointer.

## Syntax

```
catch (bad_typeid)
    statement
```

## Remarks

The interface for **bad\_typeid** is:

```
class bad_typeid : public exception
{
public:
    bad_typeid();
    bad_typeid(const char * _Message = "bad typeid");
    bad_typeid(const bad_typeid &);
    virtual ~bad_typeid();

    bad_typeid& operator=(const bad_typeid&);
    const char* what() const;
};
```

The following example shows the **typeid** operator throwing a **bad\_typeid** exception.

```
// expre_bad_typeid.cpp
// compile with: /EHsc /GR
#include <typeinfo>
#include <iostream>

class A{
public:
    // object for class needs vtable
    // for RTTI
    virtual ~A();
};

using namespace std;
int main() {
    A* a = NULL;

    try {
        cout << typeid(*a).name() << endl; // Error condition
```

```
    }  
    catch (bad_typeid){  
        cout << "Object is NULL" << endl;  
    }  
}
```

## Output

```
Object is NULL
```

## See also

[Run-Time Type Information](#)

[Keywords](#)

# type\_info Class

---

The **type\_info** class describes type information generated within the program by the compiler. Objects of this class effectively store a pointer to a name for the type. The **type\_info** class also stores an encoded value suitable for comparing two types for equality or collating order. The encoding rules and collating sequence for types are unspecified and may differ between programs.

The `<typeinfo>` header file must be included in order to use the **type\_info** class. The interface for the **type\_info** class is:

```
class type_info {
public:
    type_info(const type_info& rhs) = delete; // cannot be copied
    virtual ~type_info();
    size_t hash_code() const;
    _CRTIMP_PURE bool operator==(const type_info& rhs) const;
    type_info& operator=(const type_info& rhs) = delete; // cannot be copied
    _CRTIMP_PURE bool operator!=(const type_info& rhs) const;
    _CRTIMP_PURE int before(const type_info& rhs) const;
    size_t hash_code() const noexcept;
    _CRTIMP_PURE const char* name() const;
    _CRTIMP_PURE const char* raw_name() const;
};
```

You cannot instantiate objects of the **type\_info** class directly, because the class has only a private copy constructor. The only way to construct a (temporary) **type\_info** object is to use the `typeid` operator. Since the assignment operator is also private, you cannot copy or assign objects of class **type\_info**.

`type_info::hash_code` defines a hash function suitable for mapping values of type **typeid** to a distribution of index values.

The operators `==` and `!=` can be used to compare for equality and inequality with other **type\_info** objects, respectively.

There is no link between the collating order of types and inheritance relationships. Use the `type_info::before` member function to determine the collating sequence of types. There is no guarantee that `type_info::before` will yield the same result in different programs or even different runs of the same program. In this manner, `type_info::before` is similar to the address-of (`&`) operator.

The `type_info::name` member function returns a `const char*` to a null-terminated string representing the human-readable name of the type. The memory pointed to is cached and should never be directly deallocated.

The `type_info::raw_name` member function returns a `const char*` to a null-terminated string representing the decorated name of the object type. The name is actually stored in its decorated form to save space. Consequently, this function is faster than `type_info::name` because it doesn't need to undecorate the name.

The string returned by the `type_info::raw_name` function is useful in comparison operations but is not readable. If you need a human-readable string, use the `type_info::name` function instead.

Type information is generated for polymorphic classes only if the [/GR \(Enable Run-Time Type Information\)](#) compiler option is specified.

## See also

[Run-Time Type Information](#)

# Statements (C++)

---

C++ statements are the program elements that control how and in what order objects are manipulated. This section includes:

- [Overview](#)
- [Labeled Statements](#)
- Categories of Statements
  - [Expression statements](#). These statements evaluate an expression for its side effects or for its return value.
  - [Null statements](#). These statements can be provided where a statement is required by the C++ syntax but where no action is to be taken.
  - [Compound statements](#). These statements are groups of statements enclosed in curly braces ({ }). They can be used wherever a single statement may be used.
  - [Selection statements](#). These statements perform a test; they then execute one section of code if the test evaluates to true (nonzero). They may execute another section of code if the test evaluates to false.
  - [Iteration statements](#). These statements provide for repeated execution of a block of code until a specified termination criterion is met.
  - [Jump statements](#). These statements either transfer control immediately to another location in the function or return control from the function.
  - [Declaration statements](#). Declarations introduce a name into a program.

For information on exception handling statements see [Exception Handling](#).

See also

[C++ Language Reference](#)

# Overview of C++ Statements

---

C++ statements are executed sequentially, except when an expression statement, a selection statement, an iteration statement, or a jump statement specifically modifies that sequence.

Statements may be of the following types:

```
labeled-statement
expression-statement
compound-statement
selection-statement
iteration-statement
jump-statement
declaration-statement
try-throw-catch
```

In most cases, the C++ statement syntax is identical to that of ANSI C. The primary difference between the two is that in C, declarations are allowed only at the start of a block; C++ adds the *declaration-statement*, which effectively removes this restriction. This enables you to introduce variables at a point in the program where a precomputed initialization value can be calculated.

Declaring variables inside blocks also allows you to exercise precise control over the scope and lifetime of those variables.

The topics on statements describe the following C++ keywords:

<a href="#">break</a>	<a href="#">else</a>	<a href="#">__if_exists</a>	<a href="#">__try</a>
<a href="#">case</a>	<a href="#">__except</a>	<a href="#">__if_not_exists</a>	<a href="#">try</a>
<a href="#">catch</a>	<a href="#">for</a>	<a href="#">__leave</a>	<a href="#">while</a>
<a href="#">continue</a>	<a href="#">goto</a>	<a href="#">return</a>	
<a href="#">default</a>	<a href="#">__finally</a>	<a href="#">switch</a>	
<a href="#">do</a>	<a href="#">if</a>	<a href="#">throw</a>	

## See also

[Statements](#)

# Labeled Statements

---

Labels are used to transfer program control directly to the specified statement.

```
identifier : statement
case constant-expression : statement
default : statement
```

The scope of a label is the entire function in which it is declared.

## Remarks

There are three types of labeled statements. All use a colon to separate some type of label from the statement. The case and default labels are specific to case statements.

```
#include <iostream>
using namespace std;

void test_label(int x) {

    if (x == 1){
        goto label1;
    }
    goto label2;

label1:
    cout << "in label1" << endl;
    return;

label2:
    cout << "in label2" << endl;
    return;
}

int main() {
    test_label(1); // in label1
    test_label(2); // in label2
}
```

### The goto statement

The appearance of an *identifier* label in the source program declares a label. Only a **goto** statement can transfer control to an *identifier* label. The following code fragment illustrates use of the **goto** statement and an *identifier* label:

A label cannot appear by itself but must always be attached to a statement. If a label is needed by itself, place a null statement after the label.

The label has function scope and cannot be redeclared within the function. However, the same name can be used as a label in different functions.

```
// labels_with_goto.cpp
// compile with: /EHsc
#include <iostream>
int main() {
    using namespace std;
    goto Test2;

    cout << "testing" << endl;

    Test2:
        cerr << "At Test2 label." << endl;
}

//Output: At Test2 label.
```

## The case statement

Labels that appear after the **case** keyword cannot also appear outside a **switch** statement. (This restriction also applies to the **default** keyword.) The following code fragment shows the correct use of **case** labels:

```
// Sample Microsoft Windows message processing loop.
switch( msg )
{
    case WM_TIMER:    // Process timer event.
        SetClassWord( hWnd, GCW_HICON, ahIcon[nIcon++] );
        ShowWindow( hWnd, SW_SHOWNA );
        nIcon %= 14;
        Yield();
        break;

    case WM_PAINT:
        memset( &ps, 0x00, sizeof(PAINTSTRUCT) );
        hDC = BeginPaint( hWnd, &ps );
        EndPaint( hWnd, &ps );
        break;

    default:
        // This choice is taken for all messages not specifically
        // covered by a case statement.

        return DefWindowProc( hWnd, Message, wParam, lParam );
        break;
}
```



## Labels in the case statement

Labels that appear after the **case** keyword cannot also appear outside a **switch** statement. (This restriction also applies to the **default** keyword.) The following code fragment shows the correct use of **case** labels:

```
// Sample Microsoft Windows message processing loop.
switch( msg )
{
    case WM_TIMER:      // Process timer event.
        SetClassWord( hWnd, GCW_HICON, ahIcon[nIcon++] );
        ShowWindow( hWnd, SW_SHOWNA );
        nIcon %= 14;
        Yield();
        break;

    case WM_PAINT:
        // Obtain a handle to the device context.
        // BeginPaint will send WM_ERASEBKGND if appropriate.

        memset( &ps, 0x00, sizeof(PAINTSTRUCT) );
        hDC = BeginPaint( hWnd, &ps );

        // Inform Windows that painting is complete.

        EndPaint( hWnd, &ps );
        break;

    case WM_CLOSE:
        // Close this window and all child windows.

        KillTimer( hWnd, TIMER1 );
        DestroyWindow( hWnd );
        if ( hWnd == hWndMain )
            PostQuitMessage( 0 ); // Quit the application.
        break;

    default:
        // This choice is taken for all messages not specifically
        // covered by a case statement.

        return DefWindowProc( hWnd, Message, wParam, lParam );
        break;
}
```

## Labels in the goto statement

The appearance of an *identifier* label in the source program declares a label. Only a **goto** statement can transfer control to an *identifier* label. The following code fragment illustrates use of the **goto** statement and an *identifier* label:

A label cannot appear by itself but must always be attached to a statement. If a label is needed by itself, place a null statement after the label.

The label has function scope and cannot be redeclared within the function. However, the same name can be used as a label in different functions.

```
// labels_with_goto.cpp
// compile with: /EHsc
#include <iostream>
int main() {
    using namespace std;
    goto Test2;

    cout << "testing" << endl;

    Test2:
        cerr << "At Test2 label." << endl;
// At Test2 label.
}
```

## See also

[Overview of C++ Statements](#)

[switch Statement \(C++\)](#)

# Expression Statement

---

Expression statements cause expressions to be evaluated. No transfer of control or iteration takes place as a result of an expression statement.

The syntax for the expression statement is simply

## Syntax

```
[expression ] ;
```

## Remarks

All expressions in an expression statement are evaluated and all side effects are completed before the next statement is executed. The most common expression statements are assignments and function calls. Since the expression is optional, a semicolon alone is considered an empty expression statement, referred to as the [null](#) statement.

## See also

[Overview of C++ Statements](#)

# Null Statement

---

The "null statement" is an expression statement with the *expression* missing. It is useful when the syntax of the language calls for a statement but no expression evaluation. It consists of a semicolon.

Null statements are commonly used as placeholders in iteration statements or as statements on which to place labels at the end of compound statements or functions.

The following code fragment shows how to copy one string to another and incorporates the null statement:

```
// null_statement.cpp
char *myStrCpy( char *Dest, const char *Source )
{
    char *DestStart = Dest;

    // Assign value pointed to by Source to
    // Dest until the end-of-string 0 is
    // encountered.
    while( *Dest++ = *Source++ )
        ; // Null statement.

    return DestStart;
}

int main()
{
}
```

## See also

[Expression Statement](#)

# Compound Statements (Blocks)

---

A compound statement consists of zero or more statements enclosed in curly braces (`{ }`). A compound statement can be used anywhere a statement is expected. Compound statements are commonly called "blocks."

## Syntax

```
{ [ statement-list ] }
```

## Remarks

The following example uses a compound statement as the *statement* part of the **if** statement (see [The if Statement](#) for details about the syntax):

```
if( Amount > 100 )
{
    cout << "Amount was too large to handle\n";
    Alert();
}
else
{
    Balance -= Amount;
}
```

[!NOTE] Because a declaration is a statement, a declaration can be one of the statements in the *statement-list*. As a result, names declared inside a compound statement, but not explicitly declared as static, have local scope and (for objects) lifetime. See [Scope](#) for details about treatment of names with local scope.

## See also

[Overview of C++ Statements](#)

# Selection Statements (C++)

---

The C++ selection statements, [if](#) and [switch](#), provide a means to conditionally execute sections of code.

The [\\_\\_if\\_exists](#) and [\\_\\_if\\_not\\_exists](#) statements allow you to conditionally include code depending on the existence of a symbol.

See the individual topics for the syntax for each statement.

## See also

[Overview of C++ Statements](#)

# if-else Statement (C++)

---

Controls conditional branching. Statements in the *if-block* are executed only if the *if-expression* evaluates to a non-zero value (or TRUE). If the value of *expression* is nonzero, *statement1* and any other statements in the block are executed and the else-block, if present, is skipped. If the value of *expression* is zero, then the if-block is skipped and the else-block, if present, is executed. Expressions that evaluate to non-zero are

- TRUE
- a non-null pointer,
- any non-zero arithmetic value, or
- a class type that defines an unambiguous conversion to an arithmetic, boolean or pointer type. (For information about conversions, see [Standard Conversions](#).)

## Syntax

```
if ( expression )
{
    statement1;
    ...
}
else // optional
{
    statement2;
    ...
}

// C++17 - Visual Studio 2017 version 15.3 and later:
if ( initialization; expression )
{
    statement1;
    ...
}
else // optional
{
    statement2;
    ...
}

// C++17 - Visual Studio 2017 version 15.3 and later:
if constexpr (expression)
{
    statement1;
    ...
}
else // optional
{
    statement2;
    ...
}
```

## Example

```
// if_else_statement.cpp
#include <iostream>

using namespace std;

class C
{
public:
    void do_something(){}
};

void init(C){}
bool is_true() { return true; }
int x = 10;

int main()
{
    if (is_true())
    {
        cout << "b is true!\n"; // executed
    }
    else
    {
        cout << "b is false!\n";
    }

    // no else statement
    if (x == 10)
    {
        x = 0;
    }

    C* c;
    init(c);
    if (c)
    {
        c->do_something();
    }
    else
    {
        cout << "c is null!\n";
    }
}
```

## if statement with an initializer

**Visual Studio 2017 version 15.3 and later** (available with [/std:c++17](#)): An **if** statement may also contain an expression that declares and initializes a named variable. Use this form of the if-statement when the variable is



only needed within the scope of the if-block.

## Example

```
#include <iostream>
#include <mutex>
#include <map>
#include <string>
#include <algorithm>

using namespace std;

map<int, string> m;
mutex mx;
bool shared_flag; // guarded by mx
void unsafe_operation() {}

int main()
{
    if (auto it = m.find(10); it != m.end())
    {
        cout << it->second;
        return 0;
    }

    if (char buf[10]; fgets(buf, 10, stdin))
    {
        m[0] += buf;
    }

    if (lock_guard<mutex> lock(mx); shared_flag)
    {
        unsafe_operation();
        shared_flag = false;
    }

    string s{ "if" };
    if (auto keywords = { "if", "for", "while" }; any_of(keywords.begin(),
keywords.end(), [&s](const char* kw) { return s == kw; })))
    {
        cout << "Error! Token must not be a keyword\n";
    }
}
```

In all forms of the **if** statement, *expression*, which can have any value except a structure, is evaluated, including all side effects. Control passes from the **if** statement to the next statement in the program unless one of the *statements* contains a **break**, **continue**, or **goto**.

The **else** clause of an **if...else** statement is associated with the closest previous **if** statement in the same scope that does not have a corresponding **else** statement.

## if constexpr statements

**Visual Studio 2017 version 15.3 and later** (available with [/std:c++17](#)): In function templates, you can use an **if constexpr** statement to make compile-time branching decisions without having to resort to multiple function overloads. For example, you can write a single function that handles parameter unpacking (no zero-parameter overload is needed):

```
template <class T, class... Rest>
void f(T&& t, Rest&&... r)
{
    // handle t
    do_something(t);

    // handle r conditionally
    if constexpr (sizeof...(r))
    {
        f(r...);
    }
    else
    {
        g(r...);
    }
}
```

## See also

[Selection Statements](#)

[Keywords](#)

[switch Statement \(C++\)](#)

# \_\_if\_exists Statement

---

The **\_\_if\_exists** statement tests whether the specified identifier exists. If the identifier exists, the specified statement block is executed.

## Syntax

```
__if_exists ( identifier ) {  
    statements  
};
```

## Parameters

Parameter	Description
<i>identifier</i>	The identifier whose existence you want to test.
<i>statements</i>	One or more statements to execute if <i>identifier</i> exists.

## Remarks

[!CAUTION] To achieve the most reliable results, use the **\_\_if\_exists** statement under the following constraints.

- Apply the **\_\_if\_exists** statement to only simple types, not templates.
- Apply the **\_\_if\_exists** statement to identifiers both inside or outside a class. Do not apply the **\_\_if\_exists** statement to local variables.
- Use the **\_\_if\_exists** statement only in the body of a function. Outside of the body of a function, the **\_\_if\_exists** statement can test only fully defined types.
- When you test for overloaded functions, you cannot test for a specific form of the overload.

The complement to the **\_\_if\_exists** statement is the **\_\_if\_not\_exists** statement.

## Example

Notice that this example uses templates, which is not advised.

```
// the__if_exists_statement.cpp  
// compile with: /EHsc  
#include <iostream>  
  
template<typename T>  
class X : public T {  
public:
```

```

void Dump() {
    std::cout << "In X<T>::Dump()" << std::endl;

    __if_exists(T::Dump) {
        T::Dump();
    }

    __if_not_exists(T::Dump) {
        std::cout << "T::Dump does not exist" << std::endl;
    }
}

};

class A {
public:
    void Dump() {
        std::cout << "In A::Dump()" << std::endl;
    }
};

class B {};

bool g_bFlag = true;

class C {
public:
    void f(int);
    void f(double);
};

int main() {
    X<A> x1;
    X<B> x2;

    x1.Dump();
    x2.Dump();

    __if_exists(::g_bFlag) {
        std::cout << "g_bFlag = " << g_bFlag << std::endl;
    }

    __if_exists(C::f) {
        std::cout << "C::f exists" << std::endl;
    }

    return 0;
}

```

## Output

```

In X<A>::Dump()
In X<B>::Dump()
g_bFlag = 1
C::f exists

```

```
In X<T>::Dump()  
In A::Dump()  
In X<T>::Dump()  
T::Dump does not exist  
g_bFlag = 1  
C::f exists
```

## See also

[Selection Statements](#)

[Keywords](#)

[\\_if\\_not\\_exists Statement](#)

# \_\_if\_not\_exists Statement

---

The **\_\_if\_not\_exists** statement tests whether the specified identifier exists. If the identifier does not exist, the specified statement block is executed.

## Syntax

```
__if_not_exists ( identifier ) {  
    statements  
};
```

## Parameters

Parameter	Description
<i>identifier</i>	The identifier whose existence you want to test.
<i>statements</i>	One or more statements to execute if <i>identifier</i> does not exist.

## Remarks

[!CAUTION] To achieve the most reliable results, use the **\_\_if\_not\_exists** statement under the following constraints.

- Apply the **\_\_if\_not\_exists** statement to only simple types, not templates.
- Apply the **\_\_if\_not\_exists** statement to identifiers both inside or outside a class. Do not apply the **\_\_if\_not\_exists** statement to local variables.
- Use the **\_\_if\_not\_exists** statement only in the body of a function. Outside of the body of a function, the **\_\_if\_not\_exists** statement can test only fully defined types.
- When you test for overloaded functions, you cannot test for a specific form of the overload.

The complement to the **\_\_if\_not\_exists** statement is the [\\_\\_if\\_exists](#) statement.

## Example

For an example about how to use **\_\_if\_not\_exists**, see [\\_\\_if\\_exists Statement](#).

## See also

[Selection Statements](#)

[Keywords](#)

[\\_\\_if\\_exists Statement](#)

# switch Statement (C++)

---

Allows selection among multiple sections of code, depending on the value of an integral expression.

## Syntax

```
switch ( init; expression )
case constant-expression : statement
[default : statement]
```

## Remarks

The *expression* must be of an integral type or of a class type for which there is an unambiguous conversion to integral type. Integral promotion is performed as described in [Standard Conversions](#).

The **switch** statement body consists of a series of **case** labels and an optional **default** label. No two constant expressions in **case** statements can evaluate to the same value. The **default** label can appear only once. The labeled statements are not syntactic requirements, but the **switch** statement is meaningless without them. The default statement need not come at the end; it can appear anywhere in the body of the switch statement. A case or default label can only appear inside a switch statement.

The *constant-expression* in each **case** label is converted to the type of *expression* and compared with *expression* for equality. Control passes to the statement whose **case constant-expression** matches the value of *expression*. The resulting behavior is shown in the following table.

### Switch Statement Behavior

Condition	Action
Converted value matches that of the promoted controlling expression.	Control is transferred to the statement following that label.
None of the constants match the constants in the <b>case</b> labels; a <b>default</b> label is present.	Control is transferred to the <b>default</b> label.
None of the constants match the constants in the <b>case</b> labels; <b>default</b> label is not present.	Control is transferred to the statement after the <b>switch</b> statement.

If a matching expression is found, control is not impeded by subsequent **case** or **default** labels. The [break](#) statement is used to stop execution and transfer control to the statement after the **switch** statement. Without a **break** statement, every statement from the matched **case** label to the end of the **switch**, including the **default**, is executed. For example:

```
// switch_statement1.cpp
#include <stdio.h>
```

```

int main() {
    char *buffer = "Any character stream";
    int capa, lettera, nota;
    char c;
    capa = lettera = nota = 0;

    while ( c = *buffer++ )    // Walks buffer until NULL
    {
        switch ( c )
        {
            case 'A':
                capa++;
                break;
            case 'a':
                lettera++;
                break;
            default:
                nota++;
        }
    }
    printf_s( "\nUppercase a: %d\nLowercase a: %d\nTotal: %d\n",
        capa, lettera, (capa + lettera + nota) );
}

```

In the above example, `capa` is incremented if `c` is an uppercase `A`. The **break** statement after `capa++` terminates execution of the **switch** statement body and control passes to the **while** loop. Without the **break** statement, execution would "fall through" to the next labeled statement, so that `lettera` and `nota` would also be incremented. A similar purpose is served by the **break** statement for `case 'a'`. If `c` is a lowercase `a`, `lettera` is incremented and the **break** statement terminates the **switch** statement body. If `c` is not an `a` or `A`, the **default** statement is executed.

**Visual Studio 2017 and later:** (available with `/std:c++17`) The `[[fallthrough]]` attribute is specified in the C++17 standard. It can be used in a **switch** statement as a hint to the compiler (or to anyone reading the code) that fall-through behavior is intended. The Microsoft C++ compiler currently does not warn on fallthrough behavior, so this attribute has no effect on compiler behavior. Note that the attribute is applied to an empty statement within the labeled statement; in other words the semicolon is necessary.

```

int main()
{
    int n = 5;
    switch (n)
    {

        case 1:
            a();
            break;
        case 2:
            b();
            d();
            [[fallthrough]]; // I meant to do this!
    }
}

```



```

    case 3:
        c();
        break;
    default:
        d();
        break;
}

return 0;
}

```

**Visual Studio 2017 version 15.3 and later** (available with `/std:c++17`): A switch statement may introduce and initialize a variable whose scope is limited to the block of the switch statement:

```

switch (Gadget gadget(args); auto s = gadget.get_status())
{
    case status::good:
        gadget.zip();
        break;
    case status::bad:
        throw BadGadget();
};

```

An inner block of a **switch** statement can contain definitions with initializations as long as they are reachable — that is, not bypassed by all possible execution paths. Names introduced using these declarations have local scope. For example:

```

// switch_statement2.cpp
// C2360 expected
#include <iostream>
using namespace std;
int main(int argc, char *argv[])
{
    switch( tolower( *argv[1] ) )
    {
        // Error. Unreachable declaration.
        char szChEntered[] = "Character entered was: ";

        case 'a' :
        {
            // Declaration of szChEntered OK. Local scope.
            char szChEntered[] = "Character entered was: ";
            cout << szChEntered << "a\n";
        }
        break;

        case 'b' :
            // Value of szChEntered undefined.
            cout << szChEntered << "b\n";
    }
}

```

```
        break;

    default:
        // Value of szChEntered undefined.
        cout << szChEntered << "neither a nor b\n";
        break;
    }
}
```

A **switch** statement can be nested. In such cases, **case** or **default** labels associate with the closest **switch** statement that encloses them.

### Microsoft Specific

Microsoft C does not limit the number of case values in a **switch** statement. The number is limited only by the available memory. ANSI C requires at least 257 case labels be allowed in a **switch** statement.

The default for Microsoft C is that the Microsoft extensions are enabled. Use the [/Za](#) compiler option to disable these extensions.

### END Microsoft Specific

## See also

[Selection Statements](#)

[Keywords](#)

# Iteration Statements (C++)

---

Iteration statements cause statements (or compound statements) to be executed zero or more times, subject to some loop-termination criteria. When these statements are compound statements, they are executed in order, except when either the [break](#) statement or the [continue](#) statement is encountered.

C++ provides four iteration statements — [while](#), [do](#), [for](#), and [range-based for](#). Each of these iterates until its termination expression evaluates to zero (false), or until loop termination is forced with a **break** statement. The following table summarizes these statements and their actions; each is discussed in detail in the sections that follow.

## Iteration Statements

Statement	Evaluated At	Initialization	Increment
<b>while</b>	Top of loop	No	No
<b>do</b>	Bottom of loop	No	No
<b>for</b>	Top of loop	Yes	Yes
<b>range-based for</b>	Top of loop	Yes	Yes

The statement part of an iteration statement cannot be a declaration. However, it can be a compound statement containing a declaration.

## See also

[Overview of C++ Statements](#)

# while Statement (C++)

---

Executes *statement* repeatedly until *expression* evaluates to zero.

## Syntax

```
while ( expression )
    statement
```

## Remarks

The test of *expression* takes place before each execution of the loop; therefore, a **while** loop executes zero or more times. *expression* must be of an integral type, a pointer type, or a class type with an unambiguous conversion to an integral or pointer type.

A **while** loop can also terminate when a [break](#), [goto](#), or [return](#) within the statement body is executed. Use [continue](#) to terminate the current iteration without exiting the **while** loop. **continue** passes control to the next iteration of the **while** loop.

The following code uses a **while** loop to trim trailing underscores from a string:

```
// while_statement.cpp

#include <string.h>
#include <stdio.h>
char *trim( char *szSource )
{
    char *pszEOS = 0;

    // Set pointer to character before terminating NULL
    pszEOS = szSource + strlen( szSource ) - 1;

    // iterate backwards until non '_' is found
    while( (pszEOS >= szSource) && (*pszEOS == '_') )
        *pszEOS-- = '\0';

    return szSource;
}

int main()
{
    char szbuf[] = "12345_____";

    printf_s("\nBefore trim: %s", szbuf);
    printf_s("\nAfter trim: %s\n", trim(szbuf));
}
```

The termination condition is evaluated at the top of the loop. If there are no trailing underscores, the loop never executes.

## See also

[Iteration Statements](#)

[Keywords](#)

[do-while Statement \(C++\)](#)

[for Statement \(C++\)](#)

[Range-based for Statement \(C++\)](#)

# do-while Statement (C++)

---

Executes a *statement* repeatedly until the specified termination condition (the *expression*) evaluates to zero.

## Syntax

```
do
    statement
while ( expression ) ;
```

## Remarks

The test of the termination condition is made after each execution of the loop; therefore, a **do-while** loop executes one or more times, depending on the value of the termination expression. The **do-while** statement can also terminate when a [break](#), [goto](#), or [return](#) statement is executed within the statement body.

The *expression* must have arithmetic or pointer type. Execution proceeds as follows:

1. The statement body is executed.
2. Next, *expression* is evaluated. If *expression* is false, the **do-while** statement terminates and control passes to the next statement in the program. If *expression* is true (nonzero), the process is repeated, beginning with step 1.

## Example

The following sample demonstrates the **do-while** statement:

```
// do_while_statement.cpp
#include <stdio.h>
int main()
{
    int i = 0;
    do
    {
        printf_s("\n%d", i++);
    } while (i < 3);
}
```

## See also

[Iteration Statements](#)

[Keywords](#)

[while Statement \(C++\)](#)

for Statement (C++)

Range-based for Statement (C++)

# for Statement (C++)

Executes a statement repeatedly until the condition becomes false. For information on the range-based for statement, see [Range-based for Statement \(C++\)](#).

## Syntax

```
for ( init-expression ; cond-expression ; loop-expression )
    statement;
```

## Remarks

Use the **for** statement to construct loops that must execute a specified number of times.

The **for** statement consists of three optional parts, as shown in the following table.

### for Loop Elements

Syntax Name	When Executed	Description
init-expression	Before any other element of the <b>for</b> statement, <b>init-expression</b> is executed only once. Control then passes to <b>cond-expression</b> .	Often used to initialize loop indices. It can contain expressions or declarations.
cond-expression	Before execution of each iteration of <b>statement</b> , including the first iteration. <b>statement</b> is executed only if <b>cond-expression</b> evaluates to true (nonzero).	An expression that evaluates to an integral type or a class type that has an unambiguous conversion to an integral type. Normally used to test for loop-termination criteria.
loop-expression	At the end of each iteration of <b>statement</b> . After <b>loop-expression</b> is executed, <b>cond-expression</b> is evaluated.	Normally used to increment loop indices.

The following examples show different ways to use the **for** statement.

```
#include <iostream>
using namespace std;

int main() {
    // The counter variable can be declared in the init-expression.
    for (int i = 0; i < 2; i++){
        cout << i;
    }
}
```



```

// Output: 01
// The counter variable can be declared outside the for loop.
int i;
for (i = 0; i < 2; i++){
    cout << i;
}
// Output: 01
// These for loops are the equivalent of a while loop.
i = 0;
while (i < 2){
    cout << i++;
}
}
// Output: 012

```

`init-expression` and `loop-expression` can contain multiple statements separated by commas. For example:

```

#include <iostream>
using namespace std;

int main(){
    int i, j;
    for ( i = 5, j = 10 ; i + j < 20; i++, j++ ) {
        cout << "i + j = " << (i + j) << '\n';
    }
}
// Output:
i + j = 15
i + j = 17
i + j = 19

```

`loop-expression` can be incremented or decremented, or modified in other ways.

```

#include <iostream>
using namespace std;

int main(){
    for (int i = 10; i > 0; i--) {
        cout << i << ' ';
    }
    // Output: 10 9 8 7 6 5 4 3 2 1
    for (int i = 10; i < 20; i = i+2) {
        cout << i << ' ';
    }
    // Output: 10 12 14 16 18

```

A **for** loop terminates when a **break**, **return**, or **goto** (to a labeled statement outside the **for** loop) within **statement** is executed. A **continue** statement in a **for** loop terminates only the current iteration.

If **cond-expression** is omitted, it is considered true and the **for** loop will not terminate without a **break**, **return**, or **goto** within **statement**.

Although the three fields of the **for** statement are normally used for initialization, testing for termination, and incrementing, they are not restricted to these uses. For example, the following code prints the numbers 0 through 4. In this case, **statement** is the null statement:

```
#include <iostream>
using namespace std;

int main()
{
    int i;
    for( i = 0; i < 5; cout << i << '\n', i++){
        ;
    }
}
```

## for Loops and the C++ Standard

The C++ standard says that a variable declared in a **for** loop shall go out of scope after the **for** loop ends. For example:

```
for (int i = 0 ; i < 5 ; i++) {
    // do something
}
// i is now out of scope under /Za or /Zc:forScope
```

By default, under **/Ze**, a variable declared in a **for** loop remains in scope until the **for** loop's enclosing scope ends.

**/Zc:forScope** enables standard behavior of variables declared in for loops without needing to specify **/Za**.

It is also possible to use the scoping differences of the **for** loop to redeclare variables under **/Ze** as follows:

```
// for_statement5.cpp
int main(){
    int i = 0;    // hidden by var with same name declared in for loop
    for ( int i = 0 ; i < 3; i++ ) {}

    for ( int i = 0 ; i < 3; i++ ) {}
}
```

This more closely mimics the standard behavior of a variable declared in a **for** loop, which requires variables declared in a **for** loop to go out of scope after the loop is done. When a variable is declared in a **for** loop, the compiler internally promotes it to a local variable in the **for** loop's enclosing scope even if there is already a local variable with the same name.

## See also

[Iteration Statements](#)

[Keywords](#)

[while Statement \(C++\)](#)

[do-while Statement \(C++\)](#)

[Range-based for Statement \(C++\)](#)

# Range-based for Statement (C++)

---

Executes **statement** repeatedly and sequentially for each element in **expression**.

## Syntax

```
for ( for-range-declaration : expression )
    statement
```

## Remarks

Use the range-based **for** statement to construct loops that must execute through a "range", which is defined as anything that you can iterate through—for example, `std::vector`, or any other C++ Standard Library sequence whose range is defined by a `begin()` and `end()`. The name that is declared in the **for-range-declaration** portion is local to the **for** statement and cannot be re-declared in **expression** or **statement**. Note that the `auto` keyword is preferred in the **for-range-declaration** portion of the statement.

**New in Visual Studio 2017:** Range-based for loops no longer require that `begin()` and `end()` return objects of the same type. This enables `end()` to return a sentinel object such as used by ranges as defined in the Ranges-V3 proposal. For more information, see [Generalizing the Range-Based For Loop](#) and the [range-v3 library on GitHub](#).

This code shows how to use range-based **for** loops to iterate through an array and a vector:

```
// range-based-for.cpp
// compile by using: cl /EHsc /nologo /W4
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    // Basic 10-element integer array.
    int x[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

    // Range-based for loop to iterate through the array.
    for( int y : x ) { // Access by value using a copy declared as a specific
type.
                        // Not preferred.
        cout << y << " ";
    }
    cout << endl;

    // The auto keyword causes type inference to be used. Preferred.

    for( auto y : x ) { // Copy of 'x', almost always undesirable
        cout << y << " ";
```

```

    }
    cout << endl;

    for( auto &y : x ) { // Type inference by reference.
        // Observes and/or modifies in-place. Preferred when modify is needed.
        cout << y << " ";
    }
    cout << endl;

    for( const auto &y : x ) { // Type inference by const reference.
        // Observes in-place. Preferred when no modify is needed.
        cout << y << " ";
    }
    cout << endl;
    cout << "end of integer array test" << endl;
    cout << endl;

    // Create a vector object that contains 10 elements.
    vector<double> v;
    for (int i = 0; i < 10; ++i) {
        v.push_back(i + 0.14159);
    }

    // Range-based for loop to iterate through the vector, observing in-place.
    for( const auto &j : v ) {
        cout << j << " ";
    }
    cout << endl;
    cout << "end of vector test" << endl;
}

```

Here is the output:

```

1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
end of integer array test

0.14159 1.14159 2.14159 3.14159 4.14159 5.14159 6.14159 7.14159 8.14159 9.14159
end of vector test

```

A range-based **for** loop terminates when one of these in **statement** is executed: a **break**, **return**, or **goto** to a labeled statement outside the range-based **for** loop. A **continue** statement in a range-based **for** loop terminates only the current iteration.

Keep in mind these facts about range-based **for**:

- Automatically recognizes arrays.

- Recognizes containers that have `.begin()` and `.end()`.
- Uses argument-dependent lookup `begin()` and `end()` for anything else.

## See also

[auto](#)

[Iteration Statements](#)

[Keywords](#)

[while Statement \(C++\)](#)

[do-while Statement \(C++\)](#)

[for Statement \(C++\)](#)

# Jump Statements (C++)

---

A C++ jump statement performs an immediate local transfer of control.

## Syntax

```
break;  
continue;  
return [expression];  
goto identifier;
```

## Remarks

See the following topics for a description of the C++ jump statements.

- [break Statement](#)
- [continue Statement](#)
- [return Statement](#)
- [goto Statement](#)

## See also

[Overview of C++ Statements](#)

# break Statement (C++)

---

The **break** statement ends execution of the nearest enclosing loop or conditional statement in which it appears. Control passes to the statement that follows the end of the statement, if any.

## Syntax

```
break;
```

## Remarks

The **break** statement is used with the conditional **switch** statement and with the **do**, **for**, and **while** loop statements.

In a **switch** statement, the **break** statement causes the program to execute the next statement outside the **switch** statement. Without a **break** statement, every statement from the matched **case** label to the end of the **switch** statement, including the **default** clause, is executed.

In loops, the **break** statement ends execution of the nearest enclosing **do**, **for**, or **while** statement. Control passes to the statement that follows the ended statement, if any.

Within nested statements, the **break** statement ends only the **do**, **for**, **switch**, or **while** statement that immediately encloses it. You can use a **return** or **goto** statement to transfer control from more deeply nested structures.

## Example

The following code shows how to use the **break** statement in a **for** loop.

```
#include <iostream>
using namespace std;

int main()
{
    // An example of a standard for loop
    for (int i = 1; i < 10; i++)
    {
        if (i == 4) {
            break;
        }
        cout << i << '\n';
    }

    // An example of a range-based for loop
    int nums []{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    for (int i : nums) {
```



```

        if (i == 4) {
            break;
        }
        cout << i << '\n';
    }
}

```

In each case:

```

1
2
3

```

The following code shows how to use **break** in a **while** loop and a **do** loop.

```

#include <iostream>
using namespace std;

int main() {
    int i = 0;

    while (i < 10) {
        if (i == 4) {
            break;
        }
        cout << i << '\n';
        i++;
    }

    i = 0;
    do {
        if (i == 4) {
            break;
        }
        cout << i << '\n';
        i++;
    } while (i < 10);
}

```

In each case:

```

0123

```

The following code shows how to use **break** in a switch statement. You must use **break** in every case if you want to handle each case separately; if you do not use **break**, the code execution falls through to the next case.

```

#include <iostream>
using namespace std;

enum Suit{ Diamonds, Hearts, Clubs, Spades };

int main() {

    Suit hand;
    . . .
    // Assume that some enum value is set for hand
    // In this example, each case is handled separately
    switch (hand)
    {
    case Diamonds:
        cout << "got Diamonds \n";
        break;
    case Hearts:
        cout << "got Hearts \n";
        break;
    case Clubs:
        cout << "got Clubs \n";
        break;
    case Spades:
        cout << "got Spades \n";
        break;
    default:
        cout << "didn't get card \n";
    }
    // In this example, Diamonds and Hearts are handled one way, and
    // Clubs, Spades, and the default value are handled another way
    switch (hand)
    {
    case Diamonds:
    case Hearts:
        cout << "got a red card \n";
        break;
    case Clubs:
    case Spades:
    default:
        cout << "didn't get a red card \n";
    }
}

```

## See also

[Jump Statements](#)

[Keywords](#)

[continue Statement](#)

# continue Statement (C++)

---

Forces transfer of control to the controlling expression of the smallest enclosing [do](#), [for](#), or [while](#) loop.

## Syntax

```
continue;
```

## Remarks

Any remaining statements in the current iteration are not executed. The next iteration of the loop is determined as follows:

- In a **do** or **while** loop, the next iteration starts by reevaluating the controlling expression of the **do** or **while** statement.
- In a **for** loop (using the syntax `for(init-expr; cond-expr; loop-expr)`), the `loop-expr` clause is executed. Then the `cond-expr` clause is reevaluated and, depending on the result, the loop either ends or another iteration occurs.

The following example shows how the **continue** statement can be used to bypass sections of code and begin the next iteration of a loop.

## Example

```
// continue_statement.cpp
#include <stdio.h>
int main()
{
    int i = 0;
    do
    {
        i++;
        printf_s("before the continue\n");
        continue;
        printf("after the continue, should never print\n");
    } while (i < 3);

    printf_s("after the do loop\n");
}
```

```
before the continue
before the continue
before the continue
after the do loop
```

---

## See also

[Jump Statements](#)

[Keywords](#)

# return Statement (C++)

---

Terminates the execution of a function and returns control to the calling function (or to the operating system if you transfer control from the `main` function). Execution resumes in the calling function at the point immediately following the call.

## Syntax

```
return [expression];
```

## Remarks

The `expression` clause, if present, is converted to the type specified in the function declaration, as if an initialization were being performed. Conversion from the type of the expression to the **return** type of the function can create temporary objects. For more information about how and when temporaries are created, see [Temporary Objects](#).

The value of the `expression` clause is returned to the calling function. If the expression is omitted, the return value of the function is undefined. Constructors and destructors, and functions of type **void**, cannot specify an expression in the **return** statement. Functions of all other types must specify an expression in the **return** statement.

When the flow of control exits the block enclosing the function definition, the result is the same as it would be if a **return** statement without an expression had been executed. This is invalid for functions that are declared as returning a value.

A function can have any number of **return** statements.

The following example uses an expression with a **return** statement to obtain the largest of two integers.

## Example

```
// return_statement2.cpp
#include <stdio.h>

int max ( int a, int b )
{
    return ( a > b ? a : b );
}

int main()
{
    int nOne = 5;
    int nTwo = 7;
```

```
    printf_s("\n%d is bigger\n", max( nOne, nTwo ));  
}
```

## See also

[Jump Statements](#)

[Keywords](#)

# goto Statement (C++)

---

The **goto** statement unconditionally transfers control to the statement labeled by the specified identifier.

## Syntax

```
goto identifier;
```

## Remarks

The labeled statement designated by **identifier** must be in the current function. All **identifier** names are members of an internal namespace and therefore do not interfere with other identifiers.

A statement label is meaningful only to a **goto** statement; otherwise, statement labels are ignored. Labels cannot be redeclared.

A **goto** statement is not allowed to transfer control to a location that skips over the initialization of any variable that is in scope in that location. The following example raises C2362:

```
int goto_fn(bool b)
{
    if (!b)
    {
        goto exit; // C2362
    }
    else
    { /*...*/ }

    int error_code = 42;

exit:
    return error_code;
}
```

It is good programming style to use the **break**, **continue**, and **return** statements instead of the **goto** statement whenever possible. However, because the **break** statement exits from only one level of a loop, you might have to use a **goto** statement to exit a deeply nested loop.

For more information about labels and the **goto** statement, see [Labeled Statements](#).

## Example

In this example, a **goto** statement transfers control to the point labeled **stop** when **i** equals 3.

```

// goto_statement.cpp
#include <stdio.h>
int main()
{
    int i, j;

    for ( i = 0; i < 10; i++ )
    {
        printf_s( "Outer loop executing. i = %d\n", i );
        for ( j = 0; j < 2; j++ )
        {
            printf_s( " Inner loop executing. j = %d\n", j );
            if ( i == 3 )
                goto stop;
        }
    }

    // This message does not print:
    printf_s( "Loop exited. i = %d\n", i );

    stop:
    printf_s( "Jumped to stop. i = %d\n", i );
}

```

```

Outer loop executing. i = 0
Inner loop executing. j = 0
Inner loop executing. j = 1
Outer loop executing. i = 1
Inner loop executing. j = 0
Inner loop executing. j = 1
Outer loop executing. i = 2
Inner loop executing. j = 0
Inner loop executing. j = 1
Outer loop executing. i = 3
Inner loop executing. j = 0
Jumped to stop. i = 3

```

## See also

[Jump Statements](#)

[Keywords](#)



# Transfers of Control

---

You can use the **goto** statement or a **case** label in a **switch** statement to specify a program that branches past an initializer. Such code is illegal unless the declaration that contains the initializer is in a block enclosed by the block in which the jump statement occurs.

The following example shows a loop that declares and initializes the objects **total**, **ch**, and **i**. There is also an erroneous **goto** statement that transfers control past an initializer.

```
// transfers_of_control.cpp
// compile with: /W1
// Read input until a nonnumeric character is entered.
int main()
{
    char MyArray[5] = {'2','2','a','c'};
    int i = 0;
    while( 1 )
    {
        int total = 0;

        char ch = MyArray[i++];

        if ( ch >= '0' && ch <= '9' )
        {
            goto Label1;

            int i = ch - '0';
        Label1:
            total += i;    // C4700: transfers past initialization of i.
        } // i would be destroyed here if goto error were not present
    else
        // Break statement transfers control out of loop,
        // destroying total and ch.
        break;
    }
}
```

In the preceding example, the **goto** statement tries to transfer control past the initialization of **i**. However, if **i** were declared but not initialized, the transfer would be legal.

The objects **total** and **ch**, declared in the block that serves as the *statement* of the **while** statement, are destroyed when that block is exited using the **break** statement.

# Namespaces (C++)

---

A namespace is a declarative region that provides a scope to the identifiers (the names of types, functions, variables, etc) inside it. Namespaces are used to organize code into logical groups and to prevent name collisions that can occur especially when your code base includes multiple libraries. All identifiers at namespace scope are visible to one another without qualification. Identifiers outside the namespace can access the members by using the fully qualified name for each identifier, for example `std::vector<std::string> vec;`, or else by a [using Declaration](#) for a single identifier (`using std::string;`), or a [using Directive](#) for all the identifiers in the namespace (`using namespace std;`). Code in header files should always use the fully qualified namespace name.

The following example shows a namespace declaration and three ways that code outside the namespace can access their members.

```
namespace ContosoData
{
    class ObjectManager
    {
    public:
        void DoSomething() {}
    };
    void Func(ObjectManager) {}
}
```

Use the fully qualified name:

```
ContosoData::ObjectManager mgr;
mgr.DoSomething();
ContosoData::Func(mgr);
```

Use a using declaration to bring one identifier into scope:

```
using ContosoData::ObjectManager;
ObjectManager mgr;
mgr.DoSomething();
```

Use a using directive to bring everything in the namespace into scope:

```
using namespace ContosoData;

ObjectManager mgr;
mgr.DoSomething();
Func(mgr);
```

## using directives

The **using** directive allows all the names in a **namespace** to be used without the *namespace-name* as an explicit qualifier. Use a using directive in an implementation file (i.e. \*.cpp) if you are using several different identifiers in a namespace; if you are just using one or two identifiers, then consider a using declaration to only bring those identifiers into scope and not all the identifiers in the namespace. If a local variable has the same name as a namespace variable, the namespace variable is hidden. It is an error to have a namespace variable with the same name as a global variable.

[!NOTE] A using directive can be placed at the top of a .cpp file (at file scope), or inside a class or function definition.

In general, avoid putting using directives in header files (\*.h) because any file that includes that header will bring everything in the namespace into scope, which can cause name hiding and name collision problems that are very difficult to debug. Always use fully qualified names in a header file. If those names get too long, you can use a namespace alias to shorten them. (See below.)

## Declaring namespaces and namespace members

Typically, you declare a namespace in a header file. If your function implementations are in a separate file, then qualify the function names, as in this example.

```
//contosoData.h
#pragma once
namespace ContosoDataServer
{
    void Foo();
    int Bar();
}
```

Function implementations in contosodata.cpp should use the fully qualified name, even if you place a **using** directive at the top of the file:

```
#include "contosodata.h"
using namespace ContosoDataServer;

void ContosoDataServer::Foo() // use fully-qualified name here
{
    // no qualification needed for Bar()
    Bar();
}

int ContosoDataServer::Bar(){return 0;}
```

A namespace can be declared in multiple blocks in a single file, and in multiple files. The compiler joins the parts together during preprocessing and the resulting namespace contains all the members declared in all the parts. An example of this is the `std` namespace which is declared in each of the header files in the standard library.

Members of a named namespace can be defined outside the namespace in which they are declared by explicit qualification of the name being defined. However, the definition must appear after the point of declaration in a namespace that encloses the declaration's namespace. For example:

```
// defining_namespace_members.cpp
// C2039 expected
namespace V {
    void f();
}

void V::f() { }           // ok
void V::g() { }           // C2039, g() is not yet a member of V

namespace V {
    void g();
}
```

This error can occur when namespace members are declared across multiple header files, and you have not included those headers in the correct order.

## The global namespace

If an identifier is not declared in an explicit namespace, it is part of the implicit global namespace. In general, try to avoid making declarations at global scope when possible, except for the entry point [main Function](#), which is required to be in the global namespace. To explicitly qualify a global identifier, use the scope resolution operator with no name, as in `::SomeFunction(x)`; This will differentiate the identifier from anything with the same name in any other namespace, and it will also help to make your code easier for others to understand.

## The std namespace

All C++ standard library types and functions are declared in the `std` namespace or namespaces nested inside `std`.

## Nested namespaces

Namespaces may be nested. An ordinary nested namespace has unqualified access to its parent's members, but the parent members do not have unqualified access to the nested namespace (unless it is declared as inline), as shown in the following example:

```
namespace ContosoDataServer
{
    void Foo();
}
```

```

namespace Details
{
    int CountImpl;
    void Ban() { return Foo(); }
}

int Bar(){...};
int Baz(int i) { return Details::CountImpl; }
}

```

Ordinary nested namespaces can be used to encapsulate internal implementation details that are not part of the public interface of the parent namespace.

## Inline namespaces (C++ 11)

In contrast to an ordinary nested namespace, members of an inline namespace are treated as members of the parent namespace. This characteristic enables argument dependent lookup on overloaded functions to work on functions that have overloads in a parent and a nested inline namespace. It also enables you to declare a specialization in a parent namespace for a template that is declared in the inline namespace. The following example shows how external code binds to the inline namespace by default:

```

//Header.h
#include <string>

namespace Test
{
    namespace old_ns
    {
        std::string Func() { return std::string("Hello from old"); }
    }

    inline namespace new_ns
    {
        std::string Func() { return std::string("Hello from new"); }
    }
}

#include "header.h"
#include <string>
#include <iostream>

int main()
{
    using namespace Test;
    using namespace std;

    string s = Func();
    std::cout << s << std::endl; // "Hello from new"
    return 0;
}

```

The following example shows how you can declare a specialization in a parent of a template that is declared in an inline namespace:

```
namespace Parent
{
    inline namespace new_ns
    {
        template <typename T>
        struct C
        {
            T member;
        };
    }
    template<>
    class C<int> {};
}
```

You can use inline namespaces as a versioning mechanism to manage changes to the public interface of a library. For example, you can create a single parent namespace, and encapsulate each version of the interface in its own namespace nested inside the parent. The namespace that holds the most recent or preferred version is qualified as inline, and is therefore exposed as if it were a direct member of the parent namespace. Client code that invokes the `Parent::Class` will automatically bind to the new code. Clients that prefer to use the older version can still access it by using the fully qualified path to the nested namespace that has that code.

The inline keyword must be applied to the first declaration of the namespace in a compilation unit.

The following example shows two versions of an interface, each in a nested namespace. The `v_20` namespace has some modification from the `v_10` interface and is marked as inline. Client code that uses the new library and calls `Contoso::Funcs::Add` will invoke the `v_20` version. Code that attempts to call `Contoso::Funcs::Divide` will now get a compile time error. If they really need that function, they can still access the `v_10` version by explicitly calling `Contoso::v_10::Funcs::Divide`.

```
namespace Contoso
{
    namespace v_10
    {
        template <typename T>
        class Funcs
        {
        public:
            Funcs(void);
            T Add(T a, T b);
            T Subtract(T a, T b);
            T Multiply(T a, T b);
            T Divide(T a, T b);
        };
    }
}
```

```

    }

    inline namespace v_20
    {
        template <typename T>
        class Funcs
        {
        public:
            Funcs(void);
            T Add(T a, T b);
            T Subtract(T a, T b);
            T Multiply(T a, T b);
            std::vector<double> Log(double);
            T Accumulate(std::vector<T> nums);
        };
    }
}

```

## Namespace aliases

Namespace names need to be unique, which means that often they should not be too short. If the length of a name makes code difficult to read, or is tedious to type in a header file where using directives can't be used, then you can make a namespace alias which serves as an abbreviation for the actual name. For example:

```

namespace a_very_long_namespace_name { class Foo {}; }
namespace AVLNN = a_very_long_namespace_name;
void Bar(AVLNN::Foo foo){ }

```

## anonymous or unnamed namespaces

You can create an explicit namespace but not give it a name:

```

namespace
{
    int MyFunc(){}
}

```

This is called an unnamed or anonymous namespace and it is useful when you want to make variable declarations invisible to code in other files (i.e. give them internal linkage) without having to create a named namespace. All code in the same file can see the identifiers in an unnamed namespace but the identifiers, along with the namespace itself, are not visible outside that file—or more precisely outside the translation unit.

## See also

[Declarations and Definitions](#)

# Enumerations (C++)

---

An enumeration is a user-defined type that consists of a set of named integral constants that are known as enumerators.

[!NOTE] This article covers the ISO Standard C++ Language **enum** type and the scoped (or strongly-typed) **enum class** type which is introduced in C++11. For information about the **public enum class** or **private enum class** types in C++/CLI and C++/CX, see [enum class](#).

## Syntax

```
// unscoped enum:  
enum [identifier] [: type]  
{enum-list};
```

```
// scoped enum:  
enum [class|struct]  
[identifier] [: type]  
{enum-list};
```

```
// Forward declaration of enumerations (C++11):  
enum A : int; // non-scoped enum must have type specified  
enum class B; // scoped enum defaults to int but ...  
enum class C : short; // ... may have any integral underlying type
```

## Parameters

*identifier*

The type name given to the enumeration.

*type*

The underlying type of the enumerators; all enumerators have the same underlying type. May be any integral type.

*enum-list*

Comma-separated list of the enumerators in the enumeration. Every enumerator or variable name in the scope must be unique. However, the values can be duplicated. In a unscoped enum, the scope is the surrounding scope; in a scoped enum, the scope is the *enum-list* itself. In a scoped enum, the list may be empty which in effect defines a new integral type.

*class*

By using this keyword in the declaration, you specify the enum is scoped, and an *identifier* must be provided. You can also use the **struct** keyword in place of **class**, as they are semantically equivalent in this context.

## Enumerator scope



An enumeration provides context to describe a range of values which are represented as named constants and are also called enumerators. In the original C and C++ enum types, the unqualified enumerators are visible throughout the scope in which the enum is declared. In scoped enums, the enumerator name must be qualified by the enum type name. The following example demonstrates this basic difference between the two kinds of enums:

```
namespace CardGame_Scoped
{
    enum class Suit { Diamonds, Hearts, Clubs, Spades };

    void PlayCard(Suit suit)
    {
        if (suit == Suit::Clubs) // Enumerator must be qualified by enum type
        { /*...*/ }
    }
}

namespace CardGame_NonScoped
{
    enum Suit { Diamonds, Hearts, Clubs, Spades };

    void PlayCard(Suit suit)
    {
        if (suit == Clubs) // Enumerator is visible without qualification
        { /*...*/ }
    }
}
```

Every name in an enumeration is assigned an integral value that corresponds to its place in the order of the values in the enumeration. By default, the first value is assigned 0, the next one is assigned 1, and so on, but you can explicitly set the value of an enumerator, as shown here:

```
enum Suit { Diamonds = 1, Hearts, Clubs, Spades };
```

The enumerator **Diamonds** is assigned the value 1. Subsequent enumerators, if they are not given an explicit value, receive the value of the previous enumerator plus one. In the previous example, **Hearts** would have the value 2, **Clubs** would have 3, and so on.

Every enumerator is treated as a constant and must have a unique name within the scope where the **enum** is defined (for unscoped enums) or within the **enum** itself (for scoped enums). The values given to the names do not have to be unique. For example, if the declaration of a unscoped enum **Suit** is this:

```
enum Suit { Diamonds = 5, Hearts, Clubs = 4, Spades };
```

Then the values of **Diamonds**, **Hearts**, **Clubs**, and **Spades** are 5, 6, 4, and 5, respectively. Notice that 5 is used more than once; this is allowed even though it may not be intended. These rules are the same for scoped enums.

## Casting rules

Unscoped enum constants can be implicitly converted to **int**, but an **int** is never implicitly convertible to an enum value. The following example shows what happens if you try to assign **hand** a value that is not a **Suit**:

```
int account_num = 135692;
Suit hand;
hand = account_num; // error C2440: '=' : cannot convert from 'int' to 'Suit'
```

A cast is required to convert an **int** to a scoped or unscoped enumerator. However, you can promote a unscoped enumerator to an integer value without a cast.

```
int account_num = Hearts; //OK if Hearts is in a unscoped enum
```

Using implicit conversions in this way can lead to unintended side-effects. To help eliminate programming errors associated with unscoped enums, scoped enum values are strongly typed. Scoped enumerators must be qualified by the enum type name (identifier) and cannot be implicitly converted, as shown in the following example:

```
namespace ScopedEnumConversions
{
    enum class Suit { Diamonds, Hearts, Clubs, Spades };

    void AttemptConversions()
    {
        Suit hand;
        hand = Clubs; // error C2065: 'Clubs' : undeclared identifier
        hand = Suit::Clubs; //Correct.
        int account_num = 135692;
        hand = account_num; // error C2440: '=' : cannot convert from 'int' to
'Suit'
        hand = static_cast<Suit>(account_num); // OK, but probably a bug!!!

        account_num = Suit::Hearts; // error C2440: '=' : cannot convert from
'Suit' to 'int'
        account_num = static_cast<int>(Suit::Hearts); // OK
    }
}
```

Notice that the line **hand = account\_num;** still causes the error that occurs with unscoped enums, as shown earlier. It is allowed with an explicit cast. However, with scoped enums, the attempted conversion in the next

statement, `account_num = Suit::Hearts;`, is no longer allowed without an explicit cast.

## Enums with no enumerators

**Visual Studio 2017 version 15.3 and later** (available with [/std:c++17](#)): By defining an enum (regular or scoped) with an explicit underlying type and no enumerators, you can in effect introduce a new integral type that has no implicit conversion to any other type. By using this type instead of its built-in underlying type, you can eliminate the potential for subtle errors caused by inadvertent implicit conversions.

```
enum class byte : unsigned char { };
```

The new type is an exact copy of the underlying type, and therefore has the same calling convention, which means it can be used across ABIs without any performance penalty. No cast is required when variables of the type are initialized by using direct-list initialization. The following example shows how to initialize enums with no enumerators in various contexts:

```
enum class byte : unsigned char { };

enum class E : int { };
E e1{ 0 };
E e2 = E{ 0 };

struct X
{
    E e{ 0 };
    X() : e{ 0 } { }
};

E* p = new E{ 0 };

void f(E e) {};

int main()
{
    f(E{ 0 });
    byte i{ 42 };
    byte j = byte{ 42 };

    // unsigned char c = j; // C2440: 'initializing': cannot convert from 'byte'
    // to 'unsigned char'
    return 0;
}
```

## See also

[C Enumeration Declarations](#)

[Keywords](#)

# Unions

---

[!NOTE] In C++17 and later, the **std::variant** class is a type-safe alternative for unions.

A **union** is a user-defined type in which all members share the same memory location. This means that at any given time a union can contain no more than one object from its list of members. It also means that no matter how many members a union has, it always uses only enough memory to store the largest member.

Unions can be useful for conserving memory when you have lots of objects and/or limited memory. However they require extra care to use correctly because you are responsible for ensuring that you always access the last member that was written to. If any member types have a non-trivial constructor, then you must write additional code to explicitly construct and destroy that member. Before using a union, consider whether the problem you are trying to solve could be better expressed by using a base class and derived classes.

## Syntax

```
union [name] { member-list };
```

### Parameters

*name*

The type name given to the union.

*member-list*

Members that the union can contain. See Remarks.

### Remarks

## Declaring a Union

Begin the declaration of a union with the **union** keyword, and enclose the member list in curly braces:

```
// declaring_a_union.cpp
union RecordType    // Declare a simple union type
{
    char   ch;
    int    i;
    long   l;
    float  f;
    double d;
    int *int_ptr;
};
int main()
{
    RecordType t;
    t.i = 5; // t holds an int
```

```
t.f = 7.25; // t now holds a float  
}
```

## Using unions

In the previous example, any code that accesses the union needs to know which member is holding the data. The most common solution to this problem is to enclose the union in a struct along with an additional enum member that indicates the type of the data currently being stored in the union. This is called a *discriminated union* and the following example shows the basic pattern.

```
#include <queue>

using namespace std;

enum class WeatherDataType
{
    Temperature, Wind
};

struct TempData
{
    int StationId;
    time_t time;
    double current;
    double max;
    double min;
};

struct WindData
{
    int StationId;
    time_t time;
    int speed;
    short direction;
};

struct Input
{
    WeatherDataType type;
    union
    {
        TempData temp;
        WindData wind;
    };
};

// Functions that are specific to data types
void Process_Temp(TempData t) {}
void Process_Wind(WindData w) {}

// Container for all the data records
```

```

queue<Input> inputs;
void Initialize();

int main(int argc, char* argv[])
{
    Initialize();
    while (!inputs.empty())
    {
        Input i = inputs.front();
        switch (i.type)
        {
            case WeatherDataType::Temperature:
                Process_Temp(i.temp);
                break;
            case WeatherDataType::Wind:
                Process_Wind(i.wind);
                break;
            default:
                break;
        }
        inputs.pop();
    }
    return 0;
}

void Initialize()
{
    Input first, second;
    first.type = WeatherDataType::Temperature;
    first.temp = { 101, 1418855664, 91.8, 108.5, 67.2 };
    inputs.push(first);

    second.type = WeatherDataType::Wind;
    second.wind = { 204, 1418859354, 14, 27 };
    inputs.push(second);
}

```

In the previous example, note that the union in the `Input` struct has no name. This is an anonymous union and its members can be accessed as if they were direct members of the struct. For more information about anonymous unions, see the section below.

Of course, the previous example shows a problem that could also be solved by using classes that derive from a common base class, and branching your code based on the runtime type of each object in the container. This may result in code that is easier to maintain and understand, but it might also be slower than using unions. Also, with a union, you can store completely unrelated types, and dynamically change the type of the value that is stored without changing the type of the union variable itself. Thus you can create a heterogeneous array of `MyUnionType` whose elements store different values of different types.

Note that the `Input` struct in the preceding example can be easily misused. It is completely up to the user to use the discriminator correctly to access the member that holds the data. You can protect against misuse by

making the union private and providing special access functions, as shown in the next example.

## Unrestricted Unions (C++11)

In C++03 and earlier a union can contain non-static data members with class type as long as the type has no user provided constructors, destructors or assignment operators. In C++11, these restrictions are removed. If you include such a member in your union then the compiler will automatically mark any special member functions that are not user provided as deleted. If the union is an anonymous union inside a class or struct, then any special member functions of the class or struct that are not user provided are marked as deleted. The following example shows how to handle the case where one of the members of the union has a member that requires this special treatment:

```
// for MyVariant
#include <crtdbg.h>
#include <new>
#include <utility>

// for sample objects and output
#include <string>
#include <vector>
#include <iostream>

using namespace std;

struct A
{
    A() = default;
    A(int i, const string& str) : num(i), name(str) {}

    int num;
    string name;
    //...
};

struct B
{
    B() = default;
    B(int i, const string& str) : num(i), name(str) {}

    int num;
    string name;
    vector<int> vec;
    // ...
};

enum class Kind { None, A, B, Integer };

#pragma warning (push)
#pragma warning(disable:4624)
class MyVariant
{
```

```

public:
    MyVariant()
        : kind_(Kind::None)
    {
    }

    MyVariant(Kind kind)
        : kind_(kind)
    {
        switch (kind_)
        {
            case Kind::None:
                break;
            case Kind::A:
                new (&a_) A();
                break;
            case Kind::B:
                new (&b_) B();
                break;
            case Kind::Integer:
                i_ = 0;
                break;
            default:
                _ASSERT(false);
                break;
        }
    }

    ~MyVariant()
    {
        switch (kind_)
        {
            case Kind::None:
                break;
            case Kind::A:
                a_.~A();
                break;
            case Kind::B:
                b_.~B();
                break;
            case Kind::Integer:
                break;
            default:
                _ASSERT(false);
                break;
        }
        kind_ = Kind::None;
    }

    MyVariant(const MyVariant& other)
        : kind_(other.kind_)
    {
        switch (kind_)
        {

```



```

        case Kind::None:
            break;
        case Kind::A:
            new (&a_) A(other.a_);
            break;
        case Kind::B:
            new (&b_) B(other.b_);
            break;
        case Kind::Integer:
            i_ = other.i_;
            break;
        default:
            _ASSERT(false);
            break;
    }
}

MyVariant(MyVariant&& other)
    : kind_(other.kind_)
{
    switch (kind_)
    {
        case Kind::None:
            break;
        case Kind::A:
            new (&a_) A(move(other.a_));
            break;
        case Kind::B:
            new (&b_) B(move(other.b_));
            break;
        case Kind::Integer:
            i_ = other.i_;
            break;
        default:
            _ASSERT(false);
            break;
    }
    other.kind_ = Kind::None;
}

MyVariant& operator=(const MyVariant& other)
{
    if (&other != this)
    {
        switch (other.kind_)
        {
            case Kind::None:
                this->~MyVariant();
                break;
            case Kind::A:
                *this = other.a_;
                break;
            case Kind::B:
                *this = other.b_;

```

```

        break;
    case Kind::Integer:
        *this = other.i_;
        break;
    default:
        _ASSERT(false);
        break;
    }
}
return *this;
}

```

MyVariant& operator=(MyVariant&& other)

```

{
    _ASSERT(this != &other);
    switch (other.kind_)
    {
    case Kind::None:
        this->~MyVariant();
        break;
    case Kind::A:
        *this = move(other.a_);
        break;
    case Kind::B:
        *this = move(other.b_);
        break;
    case Kind::Integer:
        *this = other.i_;
        break;
    default:
        _ASSERT(false);
        break;
    }
    other.kind_ = Kind::None;
    return *this;
}

```

```

MyVariant(const A& a)
    : kind_(Kind::A), a_(a)
{
}

```

```

MyVariant(A&& a)
    : kind_(Kind::A), a_(move(a))
{
}

```

```

MyVariant& operator=(const A& a)
{
    if (kind_ != Kind::A)
    {
        this->~MyVariant();
        new (this) MyVariant(a);
    }
}

```

```

        else
        {
            a_ = a;
        }
        return *this;
    }

MyVariant& operator=(A&& a)
{
    if (kind_ != Kind::A)
    {
        this->~MyVariant();
        new (this) MyVariant(move(a));
    }
    else
    {
        a_ = move(a);
    }
    return *this;
}

MyVariant(const B& b)
    : kind_(Kind::B), b_(b)
{
}

MyVariant(B&& b)
    : kind_(Kind::B), b_(move(b))
{
}

MyVariant& operator=(const B& b)
{
    if (kind_ != Kind::B)
    {
        this->~MyVariant();
        new (this) MyVariant(b);
    }
    else
    {
        b_ = b;
    }
    return *this;
}

MyVariant& operator=(B&& b)
{
    if (kind_ != Kind::B)
    {
        this->~MyVariant();
        new (this) MyVariant(move(b));
    }
    else
    {

```

```

        b_ = move(b);
    }
    return *this;
}

MyVariant(int i)
    : kind_(Kind::Integer), i_(i)
{
}

MyVariant& operator=(int i)
{
    if (kind_ != Kind::Integer)
    {
        this->~MyVariant();
        new (this) MyVariant(i);
    }
    else
    {
        i_ = i;
    }
    return *this;
}

Kind GetKind() const
{
    return kind_;
}

A& GetA()
{
    _ASSERT(kind_ == Kind::A);
    return a_;
}

const A& GetA() const
{
    _ASSERT(kind_ == Kind::A);
    return a_;
}

B& GetB()
{
    _ASSERT(kind_ == Kind::B);
    return b_;
}

const B& GetB() const
{
    _ASSERT(kind_ == Kind::B);
    return b_;
}

int& GetInteger()

```

```

{
    _ASSERT(kind_ == Kind::Integer);
    return i_;
}

const int& GetInteger() const
{
    _ASSERT(kind_ == Kind::Integer);
    return i_;
}

private:
    Kind kind_;
    union
    {
        A a_;
        B b_;
        int i_;
    };
};

#pragma warning (pop)

int main()
{
    A a(1, "Hello from A");
    B b(2, "Hello from B");

    MyVariant mv_1 = a;

    cout << "mv_1 = a: " << mv_1.GetA().name << endl;
    mv_1 = b;
    cout << "mv_1 = b: " << mv_1.GetB().name << endl;
    mv_1 = A(3, "hello again from A");
    cout << R"aaa(mv_1 = A(3, "hello again from A"): )aaa" << mv_1.GetA().name <<
endl;
    mv_1 = 42;
    cout << "mv_1 = 42: " << mv_1.GetInteger() << endl;

    b.vec = { 10,20,30,40,50 };

    mv_1 = move(b);
    cout << "After move, mv_1 = b: vec.size = " << mv_1.GetB().vec.size() << endl;

    cout << endl << "Press a letter" << endl;
    char c;
    cin >> c;
}

#include <queue>
#include <iostream>
using namespace std;

enum class WeatherDataType
{
    Temperature, Wind

```

```

};

struct TempData
{
    TempData() : StationId(""), time(0), current(0), maxTemp(0), minTemp(0) {}
    TempData(string id, time_t t, double cur, double max, double min)
        : StationId(id), time(t), current(cur), maxTemp(max), minTemp(0) {}
    string StationId;
    time_t time = 0;
    double current;
    double maxTemp;
    double minTemp;
};

struct WindData
{
    int StationId;
    time_t time;
    int speed;
    short direction;
};

struct Input
{
    Input() {}
    Input(const Input&) {}

    ~Input()
    {
        if (type == WeatherDataType::Temperature)
        {
            temp.StationId.~string();
        }
    }

    WeatherDataType type;
    void SetTemp(const TempData& td)
    {
        type = WeatherDataType::Temperature;

        // must use placement new because of string member!
        new(&temp) TempData(td);
    }

    TempData GetTemp()
    {
        if (type == WeatherDataType::Temperature)
            return temp;
        else
            throw logic_error("Can't return TempData when Input holds a
WindData");
    }
    void SetWind(WindData wd)
    {

```

```

        // Explicitly delete struct member that has a
        // non-trivial constructor
        if (type == WeatherDataType::Temperature)
        {
            temp.StationId.~string();
        }
        wind = wd; //placement new not required.
    }
    WindData GetWind()
    {
        if (type == WeatherDataType::Wind)
        {
            return wind;
        }
        else
            throw logic_error("Can't return WindData when Input holds a
TempData");
    }

private:
    union
    {
        TempData temp;
        WindData wind;
    };
};

```

Unions cannot store references. Unions don't support inheritance, therefore a union itself cannot be used as a base class, or inherit from another class, or have virtual functions.

## Initializing unions

You can declare and initialize a union in the same statement by assigning an expression enclosed in braces. The expression is evaluated and assigned to the first field of the union.

```

#include <iostream>
using namespace std;

union NumericType
{
    short    iValue;
    long     lValue;
    double   dValue;
};

int main()
{
    union NumericType Values = { 10 };    // iValue = 10
    cout << Values.iValue << endl;
    Values.dValue = 3.1416;
}

```

```
    cout << Values.dValue) << endl;
}
/* Output:
10
3.141600
*/
```

The `NumericType` union is arranged in memory (conceptually) as shown in the following figure.



Storage of data in a numeric type union

Storage of Data in NumericType Union

## Anonymous unions

Anonymous unions are unions that are declared without a *class-name* or *declarator-list*.

```
union { member-list }
```

Names declared in an anonymous union are used directly, like nonmember variables. Therefore, the names declared in an anonymous union must be unique in the surrounding scope.

In addition to the restrictions for named unions, anonymous unions are subject to these additional restrictions:

- They must also be declared as **static** if declared in file or namespace scope.
- They can have only **public** members; **private** and **protected** members in anonymous unions generate errors.
- They cannot have member functions.

## See also

[Classes and Structs](#)

[Keywords](#)

[class](#)

[struct](#)



# Functions (C++)

---

A function is a block of code that performs some operation. A function can optionally define input parameters that enable callers to pass arguments into the function. A function can optionally return a value as output. Functions are useful for encapsulating common operations in a single reusable block, ideally with a name that clearly describes what the function does. The following function accepts two integers from a caller and returns their sum; *a* and *b* are *parameters* of type **int**.

```
int sum(int a, int b)
{
    return a + b;
}
```

The function can be invoked, or *called*, from any number of places in the program. The values that are passed to the function are the *arguments*, whose types must be compatible with the parameter types in the function definition.

```
int main()
{
    int i = sum(10, 32);
    int j = sum(i, 66);
    cout << "The value of j is" << j << endl; // 108
}
```

There is no practical limit to function length, but good design aims for functions that perform a single well-defined task. Complex algorithms should be broken up into easy-to-understand simpler functions whenever possible.

Functions that are defined at class scope are called member functions. In C++, unlike other languages, a function can also be defined at namespace scope (including the implicit global namespace). Such functions are called *free functions* or *non-member functions*; they are used extensively in the Standard Library.

Functions may be *overloaded*, which means different versions of a function may share the same name if they differ by the number and/or type of formal parameters. For more information, see [Function Overloading](#).

## Parts of a function declaration

A minimal function *declaration* consists of the return type, function name, and parameter list (which may be empty), along with optional keywords that provide additional instructions to the compiler. The following example is a function declaration:

```
int sum(int a, int b);
```

A function definition consists of a declaration, plus the *body*, which is all the code between the curly braces:

```
int sum(int a, int b)
{
    return a + b;
}
```

A function declaration followed by a semicolon may appear in multiple places in a program. It must appear prior to any calls to that function in each translation unit. The function definition must appear only once in the program, according to the One Definition Rule (ODR).

The required parts of a function declaration are:

1. The return type, which specifies the type of the value that the function returns, or **void** if no value is returned. In C++11, **auto** is a valid return type that instructs the compiler to infer the type from the return statement. In C++14, `decltype(auto)` is also allowed. For more information, see Type Deduction in Return Types below.
2. The function name, which must begin with a letter or underscore and cannot contain spaces. In general, leading underscores in the Standard Library function names indicate private member functions, or non-member functions that are not intended for use by your code.
3. The parameter list, a brace delimited, comma-separated set of zero or more parameters that specify the type and optionally a local name by which the values may be accessed inside the function body.

Optional parts of a function declaration are:

1. **constexpr**, which indicates that the return value of the function is a constant value can be computed at compile time.

```
constexpr float exp(float x, int n)
{
    return n == 0 ? 1 :
           n % 2 == 0 ? exp(x * x, n / 2) :
           exp(x * x, (n - 1) / 2) * x;
};
```

2. Its linkage specification, **extern** or **static**.

```
//Declare printf with C linkage.
extern "C" int printf( const char *fmt, ... );
```

For more information, see [Translation units and linkage](#).

3. **inline**, which instructs the compiler to replace every call to the function with the function code itself. inlining can help performance in scenarios where a function executes quickly and is invoked repeatedly in a performance-critical section of code.

```
inline double Account::GetBalance()
{
    return balance;
}
```

For more information, see [Inline Functions](#).

4. A **noexcept** expression, which specifies whether or not the function can throw an exception. In the following example, the function does not throw an exception if the **is\_pod** expression evaluates to **true**.

```
#include <type_traits>

template <typename T>
T copy_object(T& obj) noexcept(std::is_pod<T>) {...}
```

For more information, see [noexcept](#).

5. (Member functions only) The cv-qualifiers, which specify whether the function is **const** or **volatile**.
6. (Member functions only) **virtual**, **override**, or **final**. **virtual** specifies that a function can be overridden in a derived class. **override** means that a function in a derived class is overriding a virtual function. **final** means a function cannot be overridden in any further derived class. For more information, see [Virtual Functions](#).
7. (member functions only) **static** applied to a member function means that the function is not associated with any object instances of the class.
8. (Non-static member functions only) The ref-qualifier, which specifies to the compiler which overload of a function to choose when the implicit object parameter (\*this) is an rvalue reference vs. an lvalue reference. For more information, see [Function Overloading](#).

The following figure shows the parts of a function definition. The shaded area is the function body.

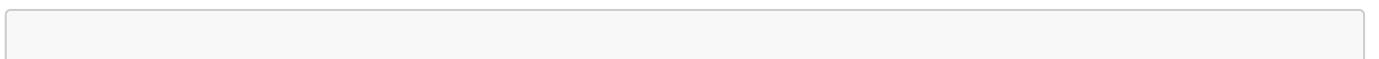


Parts of a function definition

Parts of a function definition

## Function definitions

A *function definition* consists of the declaration and the function body, enclosed in curly braces, which contains variable declarations, statements and expressions. The following example shows a complete function definition:



```
int foo(int i, std::string s)
{
    int value {i};
    MyClass mc;
    if(strcmp(s, "default") != 0)
    {
        value = mc.do_something(i);
    }
    return value;
}
```

Variables declared inside the body are called local variables or locals. They go out of scope when the function exits; therefore, a function should never return a reference to a local!

```
MyClass& boom(int i, std::string s)
{
    int value {i};
    MyClass mc;
    mc.Initialize(i,s);
    return mc;
}
```

## const and constexpr functions

You can declare a member function as **const** to specify that the function is not allowed to change the values of any data members in the class. By declaring a member function as **const**, you help the compiler to enforce *const-correctness*. If someone mistakenly tries to modify the object by using a function declared as **const**, a compiler error is raised. For more information, see [const](#).

Declare a function as **constexpr** when the value it produces can possibly be determined at compile time. A constexpr function generally executes faster than a regular function. For more information, see [constexpr](#).

## Function Templates

A function template is similar to a class template; it generates concrete functions based on the template arguments. In many cases, the template is able to infer the type arguments and therefore it isn't necessary to explicitly specify them.

```
template<typename Lhs, typename Rh>
auto Add2(const Lhs& lhs, const Rh& rhs)
{
    return lhs + rhs;
}

auto a = Add2(3.13, 2.895); // a is a double
auto b = Add2(string{ "Hello" }, string{ " World" }); // b is a std::string
```

For more information, see [Function Templates](#)

## Function parameters and arguments

A function has a comma-separated parameter list of zero or more types, each of which has a name by which it can be accessed inside the function body. A function template may specify additional type or value parameters. The caller passes arguments, which are concrete values whose types are compatible with the parameter list.

By default, arguments are passed to the function by value, which means the function receives a copy of the object being passed. For large objects, making a copy can be expensive and is not always necessary. To cause arguments to be passed by reference (specifically lvalue reference), add a reference quantifier to the parameter:

```
void DoSomething(std::string& input){...}
```

When a function modifies an argument that is passed by reference, it modifies the original object, not a local copy. To prevent a function from modifying such an argument, qualify the parameter as `const&`:

```
void DoSomething(const std::string& input){...}
```

**C++ 11:** To explicitly handle arguments that are passed by rvalue-reference or lvalue-reference, use a double-ampersand on the parameter to indicate a universal reference:

```
void DoSomething(const std::string&& input){...}
```

A function declared with the single keyword **void** in the parameter declaration list takes no arguments, as long as the keyword **void** is the first and only member of the argument declaration list. Arguments of type **void** elsewhere in the list produce errors. For example:

```
// OK same as GetTickCount()  
long GetTickCount( void );
```

Note that, while it is illegal to specify a **void** argument except as outlined here, types derived from type **void** (such as pointers to **void** and arrays of **void**) can appear anywhere the argument declaration list.

### Default Arguments

The last parameter or parameters in a function signature may be assigned a default argument, which means that the caller may leave out the argument when calling the function unless they want to specify some other value.

```

int DoSomething(int num,
               string str,
               Allocator& alloc = defaultAllocator)
{ ... }

// OK both parameters are at end
int DoSomethingElse(int num,
                   string str = string{ "Working" },
                   Allocator& alloc = defaultAllocator)
{ ... }

// C2548: 'DoMore': missing default parameter for parameter 2
int DoMore(int num = 5, // Not a trailing parameter!
           string str,
           Allocator& = defaultAllocator)
{...}

```

For more information, see [Default Arguments](#).

## Function return types

A function may not return another function, or a built-in array; however it can return pointers to these types, or a *lambda*, which produces a function object. Except for these cases, a function may return a value of any type that is in scope, or it may return no value, in which case the return type is **void**.

### Trailing return types

An "ordinary" return type is located on the left side of the function signature. A *trailing return type* is located on the right most side of the signature and is preceded by the `->` operator. Trailing return types are especially useful in function templates when the type of the return value depends on template parameters.

```

template<typename Lhs, typename Rhs>
auto Add(const Lhs& lhs, const Rhs& rhs) -> decltype(lhs + rhs)
{
    return lhs + rhs;
}

```

When **auto** is used in conjunction with a trailing return type, it just serves as a placeholder for whatever the `decltype` expression produces, and does not itself perform type deduction.

## Function local variables

A variable that is declared inside a function body is called a *local variable* or simply a *local*. Non-static locals are only visible inside the function body and, if they are declared on the stack go out of scope when the function exits. When you construct a local variable and return it by value, the compiler can usually perform the *named return value optimization* to avoid unnecessary copy operations. If you return a local variable by reference, the compiler will issue a warning because any attempt by the caller to use that reference will occur after the local has been destroyed.

In C++ a local variable may be declared as static. The variable is only visible inside the function body, but a single copy of the variable exists for all instances of the function. Local static objects are destroyed during termination specified by `atexit`. If a static object was not constructed because the program's flow of control bypassed its declaration, no attempt is made to destroy that object.

## Type deduction in return types (C++14)

In C++14, you can use **auto** to instruct the compiler to infer the return type from the function body without having to provide a trailing return type. Note that **auto** always deduces to a return-by-value. Use `auto&&` to instruct the compiler to deduce a reference.

In this example, **auto** will be deduced as a non-const value copy of the sum of lhs and rhs.

```
template<typename Lhs, typename Rhs>
auto Add2(const Lhs& lhs, const Rhs& rhs)
{
    return lhs + rhs; //returns a non-const object by value
}
```

Note that **auto** does not preserve the const-ness of the type it deduces. For forwarding functions whose return value needs to preserve the const-ness or ref-ness of its arguments, you can use the **decltype(auto)** keyword, which uses the **decltype** type inference rules and preserves all the type information. **decltype(auto)** may be used as an ordinary return value on the left side, or as a trailing return type.

The following example (based on code from [N3493](#)), shows **decltype(auto)** being used to enable perfect forwarding of function arguments in a return type that isn't known until the template is instantiated.

```
template<typename F, typename Tuple = tuple<T...>, int... I>
decltype(auto) apply_(F&& f, Tuple&& args, index_sequence<I...>)
{
    return std::forward<F>(f)(std::get<I>(std::forward<Tuple>(args))...);
}

template<typename F, typename Tuple = tuple<T...>,
         typename Indices = make_index_sequence<tuple_size<Tuple>::value >>
decltype(auto)
apply(F&& f, Tuple&& args)
{
    return apply_(std::forward<F>(f), std::forward<Tuple>(args), Indices());
}
```

## Returning multiple values from a function

There are various ways to return more than one value from a function:

1. Encapsulate the values in a named class or struct object. Requires the class or struct definition to be visible to the caller:

```

#include <string>
#include <iostream>

using namespace std;

struct S
{
    string name;
    int num;
};

S g()
{
    string t{ "hello" };
    int u{ 42 };
    return { t, u };
}

int main()
{
    S s = g();
    cout << s.name << " " << s.num << endl;
    return 0;
}

```

2. Return a std::tuple or std::pair object:

```

#include <tuple>
#include <string>
#include <iostream>

using namespace std;

tuple<int, string, double> f()
{
    int i{ 108 };
    string s{ "Some text" };
    double d{ .01 };
    return { i,s,d };
}

int main()
{
    auto t = f();
    cout << get<0>(t) << " " << get<1>(t) << " " << get<2>(t) << endl;

    // --or--

    int myval;
    string myname;
    double mydecimal;
}

```



```

    tie(myval, myname, mydecimal) = f();
    cout << myval << " " << myname << " " << mydecimal << endl;

    return 0;
}

```

3. **Visual Studio 2017 version 15.3 and later** (available with [/std:c++17](#)): Use structured bindings. The advantage of structured bindings is that the variables that store the return values are initialized at the same time they are declared, which in some cases can be significantly more efficient. In this statement - `auto[x, y, z] = f();` -- the brackets introduce and initialize names that are in scope for the entire function block.

```

#include <tuple>
#include <string>
#include <iostream>

using namespace std;

tuple<int, string, double> f()
{
    int i{ 108 };
    string s{ "Some text" };
    double d{ .01 };
    return { i,s,d };
}

struct S
{
    string name;
    int num;
};

S g()
{
    string t{ "hello" };
    int u{ 42 };
    return { t, u };
}

int main()
{
    auto[x, y, z] = f(); // init from tuple
    cout << x << " " << y << " " << z << endl;

    auto[a, b] = g(); // init from POD struct
    cout << a << " " << b << endl;
    return 0;
}

```

4. In addition to using the return value itself, you can "return" values by defining any number of parameters to use pass-by-reference so that the function can modify or initialize the values of objects

that the caller provides. For more information, see [Reference-Type Function Arguments](#).

## Function pointers

C++ supports function pointers in the same manner as the C language. However a more type-safe alternative is usually to use a function object.

It is recommended that **typedef** be used to declare an alias for the function pointer type if declaring a function that returns a function pointer type. For example

```
typedef int (*fp)(int);  
fp myFunction(char* s); // function returning function pointer
```

If this is not done, the proper syntax for the function declaration may be deduced from the declarator syntax for the function pointer by replacing the identifier (`fp` in the above example) with the functions name and argument list, as follows:

```
int (*myFunction(char* s))(int);
```

The preceding declaration is equivalent to the declaration using typedef above.

## See also

[Function Overloading](#)

[Functions with Variable Argument Lists](#)

[Explicitly Defaulted and Deleted Functions](#)

[Argument-Dependent Name \(Koenig\) Lookup on Functions](#)

[Default Arguments](#)

[Inline Functions](#)

# Functions with Variable Argument Lists (C++)

---

Function declarations in which the last member of is the ellipsis (...) can take a variable number of arguments. In these cases, C++ provides type checking only for the explicitly declared arguments. You can use variable argument lists when you need to make a function so general that even the number and types of arguments can vary. The family of functions is an example of functions that use variable argument lists.[printfargument-declaration-list](#)

## Functions with variable arguments

To access arguments after those declared, use the macros contained in the standard include file <stdarg.h> as described below.

### Microsoft Specific

Microsoft C++ allows the ellipsis to be specified as an argument if the ellipsis is the last argument and the ellipsis is preceded by a comma. Therefore, the declaration `int Func( int i, ... );` is legal, but `int Func( int i ... );` is not.

### END Microsoft Specific

Declaration of a function that takes a variable number of arguments requires at least one placeholder argument, even if it is not used. If this placeholder argument is not supplied, there is no way to access the remaining arguments.

When arguments of type **char** are passed as variable arguments, they are converted to type **int**. Similarly, when arguments of type **float** are passed as variable arguments, they are converted to type **double**. Arguments of other types are subject to the usual integral and floating-point promotions. See [Standard Conversions](#) for more information.

Functions that require variable lists are declared by using the ellipsis (...) in the argument list. Use the types and macros that are described in the <stdarg.h> include file to access arguments that are passed by a variable list. For more information about these macros, see [va\\_arg](#), [va\\_copy](#), [va\\_end](#), [va\\_start](#). in the documentation for the C Run-Time Library.

The following example shows how the macros work together with the type (declared in <stdarg.h>):

```
// variable_argument_lists.cpp
#include <stdio.h>
#include <stdarg.h>

// Declaration, but not definition, of ShowVar.
void ShowVar( char *szTypes, ... );
int main() {
    ShowVar( "fcsi", 32.4f, 'a', "Test string", 4 );
}

// ShowVar takes a format string of the form
// "ifcs", where each character specifies the
```

```

//  type of the argument in that position.
//
//  i = int
//  f = float
//  c = char
//  s = string (char *)
//
//  Following the format specification is a variable
//  list of arguments. Each argument corresponds to
//  a format character in the format string to which
//  the szTypes parameter points
void ShowVar( char *szTypes, ... ) {
    va_list vl;
    int i;

    //  szTypes is the last argument specified; you must access
    //  all others using the variable-argument macros.
    va_start( vl, szTypes );

    // Step through the list.
    for( i = 0; szTypes[i] != '\0'; ++i ) {
        union Printable_t {
            int      i;
            float    f;
            char     c;
            char     *s;
        } Printable;

        switch( szTypes[i] ) {    // Type to expect.
            case 'i':
                Printable.i = va_arg( vl, int );
                printf_s( "%i\n", Printable.i );
                break;

            case 'f':
                Printable.f = va_arg( vl, double );
                printf_s( "%f\n", Printable.f );
                break;

            case 'c':
                Printable.c = va_arg( vl, char );
                printf_s( "%c\n", Printable.c );
                break;

            case 's':
                Printable.s = va_arg( vl, char * );
                printf_s( "%s\n", Printable.s );
                break;

            default:
                break;
        }
    }
    va_end( vl );
}

```

```
}  
//Output:  
// 32.400002  
// a  
// Test string
```

The previous example illustrates these important concepts:

1. You must establish a list marker as a variable of type `va_list` before any variable arguments are accessed. In the previous example, the marker is called `v1`.
2. The individual arguments are accessed by using the `va_arg` macro. You must tell the `va_arg` macro the type of argument to retrieve so that it can transfer the correct number of bytes from the stack. If you specify an incorrect type of a size different from that supplied by the calling program to `va_arg`, the results are unpredictable.
3. You should explicitly cast the result obtained by using the `va_arg` macro to the type that you want.

You must call the macro to terminate variable-argument processing. `va_end`

# Function Overloading

---

C++ allows specification of more than one function of the same name in the same scope. These functions are called *overloaded* functions. Overloaded functions enable you to supply different semantics for a function, depending on the types and number of arguments.

For example, a `print` function that takes a `std::string` argument might perform very different tasks than one that takes an argument of type `double`. Overloading saves you from having to use names such as `print_string` or `print_double`. At compile time, the compiler chooses which overload to use based on the type of arguments passed in by the caller. If you call `print(42.0)`, then the `void print(double d)` function will be invoked. If you call `print("hello world")`, then the `void print(std::string)` overload will be invoked.

You can overload both member functions and non-member functions. The following table shows what parts of a function declaration C++ uses to differentiate between groups of functions with the same name in the same scope.

## Overloading Considerations

Function Declaration Element	Used for Overloading?
Function return type	No
Number of arguments	Yes
Type of arguments	Yes
Presence or absence of ellipsis	Yes
Use of <b>typedef</b> names	No
Unspecified array bounds	No
<b>const</b> or <b>volatile</b>	Yes, when applied to entire function
Ref-qualifiers	Yes

## Example

The following example illustrates how overloading can be used.

```
// function_overloading.cpp
// compile with: /EHsc
#include <iostream>
#include <math.h>
#include <string>

// Prototype three print functions.
int print(std::string s);           // Print a string.
int print(double dvalue);          // Print a double.
```

```

int print(double dvalue, int prec); // Print a double with a
                                   // given precision.

using namespace std;
int main(int argc, char *argv[])
{
    const double d = 893094.2987;
    if (argc < 2)
    {
        // These calls to print invoke print( char *s ).
        print("This program requires one argument.");
        print("The argument specifies the number of");
        print("digits precision for the second number");
        print("printed.");
        exit(0);
    }

    // Invoke print( double dvalue ).
    print(d);

    // Invoke print( double dvalue, int prec ).
    print(d, atoi(argv[1]));
}

// Print a string.
int print(string s)
{
    cout << s << endl;
    return cout.good();
}

// Print a double in default precision.
int print(double dvalue)
{
    cout << dvalue << endl;
    return cout.good();
}

// Print a double in specified precision.
// Positive numbers for precision indicate how many digits
// precision after the decimal point to show. Negative
// numbers for precision indicate where to round the number
// to the left of the decimal point.
int print(double dvalue, int prec)
{
    // Use table-lookup for rounding/truncation.
    static const double rgPow10[] = {
        10E-7, 10E-6, 10E-5, 10E-4, 10E-3, 10E-2, 10E-1,
        10E0, 10E1, 10E2, 10E3, 10E4, 10E5, 10E6 };
    const int iPowZero = 6;

    // If precision out of range, just print the number.
    if (prec < -6 || prec > 7)
    {
        return print(dvalue);
    }
}

```

```

    }
    // Scale, truncate, then rescale.
    dvalue = floor(dvalue / rgPow10[iPowZero - prec]) *
        rgPow10[iPowZero - prec];
    cout << dvalue << endl;
    return cout.good();
}

```

The preceding code shows overloading of the `print` function in file scope.

The default argument isn't considered part of the function type. Therefore, it's not used in selecting overloaded functions. Two functions that differ only in their default arguments are considered multiple definitions rather than overloaded functions.

Default arguments can't be supplied for overloaded operators.

## Argument Matching

Overloaded functions are selected for the best match of function declarations in the current scope to the arguments supplied in the function call. If a suitable function is found, that function is called. "Suitable" in this context means either:

- An exact match was found.
- A trivial conversion was performed.
- An integral promotion was performed.
- A standard conversion to the desired argument type exists.
- A user-defined conversion (either conversion operator or constructor) to the desired argument type exists.
- Arguments represented by an ellipsis were found.

The compiler creates a set of candidate functions for each argument. Candidate functions are functions in which the actual argument in that position can be converted to the type of the formal argument.

A set of "best matching functions" is built for each argument, and the selected function is the intersection of all the sets. If the intersection contains more than one function, the overloading is ambiguous and generates an error. The function that is eventually selected is always a better match than every other function in the group for at least one argument. If there's no clear winner, the function call generates an error.

Consider the following declarations (the functions are marked `Variant 1`, `Variant 2`, and `Variant 3`, for identification in the following discussion):

```

Fraction &Add( Fraction &f, long l );           // Variant 1
Fraction &Add( long l, Fraction &f );           // Variant 2
Fraction &Add( Fraction &f, Fraction &f );      // Variant 3

Fraction F1, F2;

```



Consider the following statement:

```
F1 = Add( F2, 23 );
```

The preceding statement builds two sets:

Set 1: Candidate Functions That Have First Argument of Type Fraction	Set 2: Candidate Functions Whose Second Argument Can Be Converted to Type int
Variant 1	Variant 1 ( <b>int</b> can be converted to <b>long</b> using a standard conversion)
Variant 3	

Functions in Set 2 are functions for which there are implicit conversions from actual parameter type to formal parameter type, and among such functions there's a function for which the "cost" of converting the actual parameter type to its formal parameter type is the smallest.

The intersection of these two sets is Variant 1. An example of an ambiguous function call is:

```
F1 = Add( 3, 6 );
```

The preceding function call builds the following sets:

Set 1: Candidate Functions That Have First Argument of Type int	Set 2: Candidate Functions That Have Second Argument of Type int
Variant 2 ( <b>int</b> can be converted to <b>long</b> using a standard conversion)	Variant 1 ( <b>int</b> can be converted to <b>long</b> using a standard conversion)

Because the intersection of these two sets is empty, the compiler generates an error message.

For argument matching, a function with *n* default arguments is treated as *n*+1 separate functions, each with a different number of arguments.

The ellipsis (...) acts as a wildcard; it matches any actual argument. It can lead to many ambiguous sets, if you don't design your overloaded function sets with extreme care.

[!NOTE] Ambiguity of overloaded functions can't be determined until a function call is encountered. At that point, the sets are built for each argument in the function call, and you can determine whether an unambiguous overload exists. This means that ambiguities can remain in your code until they are evoked by a particular function call.

## Argument Type Differences

Overloaded functions differentiate between argument types that take different initializers. Therefore, an argument of a given type and a reference to that type are considered the same for the purposes of overloading. They are considered the same because they take the same initializers. For example, `max( double, double )` is considered the same as `max( double &, double & )`. Declaring two such functions causes an error.

For the same reason, function arguments of a type modified by **const** or **volatile** are not treated differently than the base type for the purposes of overloading.

However, the function overloading mechanism can distinguish between references that are qualified by **const** and **volatile** and references to the base type. It makes code such as the following possible:

```
// argument_type_differences.cpp
// compile with: /EHsc /W3
// C4521 expected
#include <iostream>

using namespace std;
class Over {
public:
    Over() { cout << "Over default constructor\n"; }
    Over( Over &o ) { cout << "Over&\n"; }
    Over( const Over &co ) { cout << "const Over&\n"; }
    Over( volatile Over &vo ) { cout << "volatile Over&\n"; }
};

int main() {
    Over o1;           // Calls default constructor.
    Over o2( o1 );     // Calls Over( Over& ).
    const Over o3;     // Calls default constructor.
    Over o4( o3 );     // Calls Over( const Over& ).
    volatile Over o5;  // Calls default constructor.
    Over o6( o5 );     // Calls Over( volatile Over& ).
}
```

## Output

```
Over default constructor
Over&
Over default constructor
const Over&
Over default constructor
volatile Over&
```

Pointers to **const** and **volatile** objects are also considered different from pointers to the base type for the purposes of overloading.

## Argument matching and conversions

When the compiler tries to match actual arguments against the arguments in function declarations, it can supply standard or user-defined conversions to obtain the correct type if no exact match can be found. The application of conversions is subject to these rules:

- Sequences of conversions that contain more than one user-defined conversion are not considered.
- Sequences of conversions that can be shortened by removing intermediate conversions are not considered.

The resultant sequence of conversions, if any, is called the best matching sequence. There are several ways to convert an object of type **int** to type **unsigned long** using standard conversions (described in [Standard Conversions](#)):

- Convert from **int** to **long** and then from **long** to **unsigned long**.
- Convert from **int** to **unsigned long**.

The first sequence, although it achieves the desired goal, isn't the best matching sequence — a shorter sequence exists.

The following table shows a group of conversions, called trivial conversions, that have a limited effect on determining which sequence is the best matching. The instances in which trivial conversions affect choice of sequence are discussed in the list following the table.


### Trivial Conversions

Convert from Type	Convert to Type
<i>type-name</i>	<i>type-name</i> &
<i>type-name</i> &	<i>type-name</i>
<i>type-name</i> [ ]	<i>type-name</i> *
<i>type-name</i> ( <i>argument-list</i> )	( * <i>type-name</i> ) ( <i>argument-list</i> )
<i>type-name</i>	<b>const</b> <i>type-name</i>
<i>type-name</i>	<b>volatile</b> <i>type-name</i>
<i>type-name</i> *	<b>const</b> <i>type-name</i> *
<i>type-name</i> *	<b>volatile</b> <i>type-name</i> *

The sequence in which conversions are attempted is as follows:

1. Exact match. An exact match between the types with which the function is called and the types declared in the function prototype is always the best match. Sequences of trivial conversions are classified as exact matches. However, sequences that don't make any of these conversions are considered better than sequences that convert:
  - From pointer, to pointer to **const** (**type** \* to **const type** \*).
  - From pointer, to pointer to **volatile** (**type** \* to **volatile type** \*).

- From reference, to reference to **const** (**type &** to **const type &**).
  - From reference, to reference to **volatile** (**type &** to **volatile type &**).
2. Match using promotions. Any sequence not classified as an exact match that contains only integral promotions, conversions from **float** to **double**, and trivial conversions is classified as a match using promotions. Although not as good a match as any exact match, a match using promotions is better than a match using standard conversions.
  3. Match using standard conversions. Any sequence not classified as an exact match or a match using promotions that contains only standard conversions and trivial conversions is classified as a match using standard conversions. Within this category, the following rules are applied:
    - Conversion from a pointer to a derived class, to a pointer to a direct or indirect base class is preferable to converting to **void \*** or **const void \***.
    - Conversion from a pointer to a derived class, to a pointer to a base class produces a better match the closer the base class is to a direct base class. Suppose the class hierarchy is as shown in the following figure.

 Graph of preferred conversions


Graph showing preferred conversions

Conversion from type **D\*** to type **C\*** is preferable to conversion from type **D\*** to type **B\***. Similarly, conversion from type **D\*** to type **B\*** is preferable to conversion from type **D\*** to type **A\***.

This same rule applies to reference conversions. Conversion from type **D&** to type **C&** is preferable to conversion from type **D&** to type **B&**, and so on.

This same rule applies to pointer-to-member conversions. Conversion from type **T D::\*** to type **T C::\*** is preferable to conversion from type **T D::\*** to type **T B::\***, and so on (where **T** is the type of the member).

The preceding rule applies only along a given path of derivation. Consider the graph shown in the following figure.

 Multiple-inheritance that shows preferred conversions

Multiple-inheritance graph that shows preferred conversions

Conversion from type **C\*** to type **B\*** is preferable to conversion from type **C\*** to type **A\***. The reason is that they are on the same path, and **B\*** is closer. However, conversion from type **C\*** to type **D\*** isn't preferable to conversion to type **A\***; there's no preference because the conversions follow different paths.

1. Match with user-defined conversions. This sequence can't be classified as an exact match, a match using promotions, or a match using standard conversions. The sequence must contain only user-defined conversions, standard conversions, or trivial conversions to be classified as a match with user-defined conversions. A match with user-defined conversions is considered a better match than a match with an ellipsis but not as good a match as a match with standard conversions.
2. Match with an ellipsis. Any sequence that matches an ellipsis in the declaration is classified as a match with an ellipsis. It's considered the weakest match.

User-defined conversions are applied if no built-in promotion or conversion exists. These conversions are selected on the basis of the type of the argument being matched. Consider the following code:

```
// argument_matching1.cpp
class UDC
{
public:
    operator int()
    {
        return 0;
    }
    operator long();
};

void Print( int i )
{
};

UDC udc;

int main()
{
    Print( udc );
}
```

The available user-defined conversions for class **UDC** are from type **int** and type **long**. Therefore, the compiler considers conversions for the type of the object being matched: **UDC**. A conversion to **int** exists, and it is selected.

During the process of matching arguments, standard conversions can be applied to both the argument and the result of a user-defined conversion. Therefore, the following code works:

```
void LogToFile( long l );
...
UDC udc;
LogToFile( udc );
```

In the preceding example, the user-defined conversion, **operator long**, is invoked to convert **udc** to type **long**. If no user-defined conversion to type **long** had been defined, the conversion would have proceeded as follows: Type **UDC** would have been converted to type **int** using the user-defined conversion. Then the standard conversion from type **int** to type **long** would have been applied to match the argument in the declaration.

If any user-defined conversions are required to match an argument, the standard conversions aren't used when evaluating the best match. Even if more than one candidate function requires a user-defined conversion, the functions are considered equal. For example:

```

// argument_matching2.cpp
// C2668 expected
class UDC1
{
public:
    UDC1( int ); // User-defined conversion from int.
};

class UDC2
{
public:
    UDC2( long ); // User-defined conversion from long.
};

void Func( UDC1 );
void Func( UDC2 );

int main()
{
    Func( 1 );
}

```

Both versions of `Func` require a user-defined conversion to convert type `int` to the class type argument. The possible conversions are:

- Convert from type `int` to type `UDC1` (a user-defined conversion).
- Convert from type `int` to type `long`; then convert to type `UDC2` (a two-step conversion).

Even though the second one requires both a standard conversion and the user-defined conversion, the two conversions are still considered equal.

[!NOTE] User-defined conversions are considered conversion by construction or conversion by initialization (conversion function). Both methods are considered equal when considering the best match.

## Argument matching and the `this` pointer

Class member functions are treated differently, depending on whether they are declared as **static**. Because nonstatic functions have an implicit argument that supplies the **this** pointer, nonstatic functions are considered to have one more argument than static functions; otherwise, they are declared identically.

These nonstatic member functions require that the implied **this** pointer match the object type through which the function is being called, or, for overloaded operators, they require that the first argument match the object on which the operator is being applied. (For more information about overloaded operators, see [Overloaded Operators](#).)

Unlike other arguments in overloaded functions, no temporary objects are introduced and no conversions are attempted when trying to match the **this** pointer argument.

When the `->` member-selection operator is used to access a member function of class `class_name`, the **this** pointer argument has a type of `class_name * const`. If the members are declared as **const** or **volatile**, the types are `const class_name * const` and `volatile class_name * const`, respectively.

The `.` member-selection operator works exactly the same way, except that an implicit `&` (address-of) operator is prefixed to the object name. The following example shows how this works:

```
// Expression encountered in code
obj.name

// How the compiler treats it
(&obj)->name
```

The left operand of the `->*` and `.*` (pointer to member) operators are treated the same way as the `.` and `->` (member-selection) operators with respect to argument matching.

## Ref-qualifiers on member functions

Ref qualifiers make it possible to overload a member function on the basis of whether the object pointed to by **this** is an rvalue or an lvalue. This feature can be used to avoid unnecessary copy operations in scenarios where you choose not to provide pointer access to the data. For example, assume class `C` initializes some data in its constructor, and returns a copy of that data in member function `get_data()`. If an object of type `C` is an rvalue that is about to be destroyed, then the compiler will choose the `get_data() &&` overload, which moves the data rather than copy it.

```
#include <iostream>
#include <vector>

using namespace std;

class C
{
public:
    C() { /*expensive initialization*/ }
    vector<unsigned> get_data() &
    {
        cout << "lvalue\n";
        return _data;
    }
    vector<unsigned> get_data() &&
    {
        cout << "rvalue\n";
        return std::move(_data);
    }

private:
    vector<unsigned> _data;
};
```

```
int main()
{
    C c;
    auto v = c.get_data(); // get a copy. prints "lvalue".
    auto v2 = C().get_data(); // get the original. prints "rvalue"
    return 0;
}
```

## Restrictions on overloading

Several restrictions govern an acceptable set of overloaded functions:

- Any two functions in a set of overloaded functions must have different argument lists.
- Overloading functions with argument lists of the same types, based on return type alone, is an error.

### Microsoft Specific

You can overload **operator new** solely on the basis of return type — specifically, on the basis of the memory-model modifier specified.

### END Microsoft Specific

- Member functions can't be overloaded solely on the basis of one being static and the other nonstatic.
- **typedef** declarations do not define new types; they introduce synonyms for existing types. They don't affect the overloading mechanism. Consider the following code:

```
typedef char * PSTR;

void Print( char *szToPrint );
void Print( PSTR szToPrint );
```

The preceding two functions have identical argument lists. **PSTR** is a synonym for type **char \***. In member scope, this code generates an error.

- Enumerated types are distinct types and can be used to distinguish between overloaded functions.
- The types "array of " and "pointer to" are considered identical for the purposes of distinguishing between overloaded functions, but only for singly dimensioned arrays. That's why these overloaded functions conflict and generate an error message:

```
void Print( char *szToPrint );
void Print( char szToPrint[] );
```

For multiply dimensioned arrays, the second and all succeeding dimensions are considered part of the type. Therefore, they are used in distinguishing between overloaded functions:



```
void Print( char szToPrint[] );
void Print( char szToPrint[][7] );
void Print( char szToPrint[][9][42] );
```

## Overloading, overriding, and hiding

Any two function declarations of the same name in the same scope can refer to the same function, or to two discrete functions that are overloaded. If the argument lists of the declarations contain arguments of equivalent types (as described in the previous section), the function declarations refer to the same function. Otherwise, they refer to two different functions that are selected using overloading.

Class scope is strictly observed; therefore, a function declared in a base class isn't in the same scope as a function declared in a derived class. If a function in a derived class is declared with the same name as a virtual function in the base class, the derived-class function *overrides* the base-class function. For more information, see [Virtual Functions](#).

If the base class function isn't declared as 'virtual', then the derived class function is said to *hide* it. Both overriding and hiding are distinct from overloading.

Block scope is strictly observed; therefore, a function declared in file scope isn't in the same scope as a function declared locally. If a locally declared function has the same name as a function declared in file scope, the locally declared function hides the file-scoped function instead of causing overloading. For example:

```
// declaration_matching1.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;
void func( int i )
{
    cout << "Called file-scoped func : " << i << endl;
}

void func( char *sz )
{
    cout << "Called locally declared func : " << sz << endl;
}

int main()
{
    // Declare func local to main.
    extern void func( char *sz );

    func( 3 );    // C2664 Error. func( int ) is hidden.
    func( "s" );
}
```

The preceding code shows two definitions from the function `func`. The definition that takes an argument of type `char *` is local to `main` because of the `extern` statement. Therefore, the definition that takes an argument of type `int` is hidden, and the first call to `func` is in error.

For overloaded member functions, different versions of the function can be given different access privileges. They are still considered to be in the scope of the enclosing class and thus are overloaded functions. Consider the following code, in which the member function `Deposit` is overloaded; one version is public, the other, private.

The intent of this sample is to provide an `Account` class in which a correct password is required to perform deposits. It's done by using overloading.

The call to `Deposit` in `Account::Deposit` calls the private member function. This call is correct because `Account::Deposit` is a member function, and has access to the private members of the class.

```
// declaration_matching2.cpp
class Account
{
public:
    Account()
    {
    }
    double Deposit( double dAmount, char *szPassword );

private:
    double Deposit( double dAmount )
    {
        return 0.0;
    }
    int Validate( char *szPassword )
    {
        return 0;
    }
};

int main()
{
    // Allocate a new object of type Account.
    Account *pAcct = new Account;

    // Deposit $57.22. Error: calls a private function.
    // pAcct->Deposit( 57.22 );

    // Deposit $57.22 and supply a password. OK: calls a
    // public function.
    pAcct->Deposit( 52.77, "pswd" );
}

double Account::Deposit( double dAmount, char *szPassword )
{
    if ( Validate( szPassword ) )
```

```
        return Deposit( dAmount );  
    else  
        return 0.0;  
}
```

## See also

[Functions \(C++\)](#)

# Explicitly Defaulted and Deleted Functions

---

In C++11, defaulted and deleted functions give you explicit control over whether the special member functions are automatically generated. Deleted functions also give you simple language to prevent problematic type promotions from occurring in arguments to functions of all types—special member functions, as well as normal member functions and non-member functions—which would otherwise cause an unwanted function call.

## Benefits of explicitly defaulted and deleted functions

In C++, the compiler automatically generates the default constructor, copy constructor, copy-assignment operator, and destructor for a type if it does not declare its own. These functions are known as the *special member functions*, and they are what make simple user-defined types in C++ behave like structures do in C. That is, you can create, copy, and destroy them without any additional coding effort. C++11 brings move semantics to the language and adds the move constructor and move-assignment operator to the list of special member functions that the compiler can automatically generate.

This is convenient for simple types, but complex types often define one or more of the special member functions themselves, and this can prevent other special member functions from being automatically generated. In practice:

- If any constructor is explicitly declared, then no default constructor is automatically generated.
- If a virtual destructor is explicitly declared, then no default destructor is automatically generated.
- If a move constructor or move-assignment operator is explicitly declared, then:
  - No copy constructor is automatically generated.
  - No copy-assignment operator is automatically generated.
- If a copy constructor, copy-assignment operator, move constructor, move-assignment operator, or destructor is explicitly declared, then:
  - No move constructor is automatically generated.
  - No move-assignment operator is automatically generated.

[!NOTE] Additionally, the C++11 standard specifies the following additional rules:

- If a copy constructor or destructor is explicitly declared, then automatic generation of the copy-assignment operator is deprecated.
- If a copy-assignment operator or destructor is explicitly declared, then automatic generation of the copy constructor is deprecated.

In both cases, Visual Studio continues to automatically generate the necessary functions implicitly, and does not emit a warning.

The consequences of these rules can also leak into object hierarchies. For example, if for any reason a base class fails to have a default constructor that's callable from a deriving class—that is, a **public** or **protected**

constructor that takes no parameters—then a class that derives from it cannot automatically generate its own default constructor.

These rules can complicate the implementation of what should be straight-forward, user-defined types and common C++ idioms—for example, making a user-defined type non-copyable by declaring the copy constructor and copy-assignment operator privately and not defining them.

```
struct noncopyable
{
    noncopyable() {};
```

private:

```
    noncopyable(const noncopyable&);
    noncopyable& operator=(const noncopyable&);
};
```

Before C++11, this code snippet was the idiomatic form of non-copyable types. However, it has several problems:

- The copy constructor has to be declared privately to hide it, but because it's declared at all, automatic generation of the default constructor is prevented. You have to explicitly define the default constructor if you want one, even if it does nothing.
- Even if the explicitly-defined default constructor does nothing, it's considered non-trivial by the compiler. It's less efficient than an automatically generated default constructor and prevents `noncopyable` from being a true POD type.
- Even though the copy constructor and copy-assignment operator are hidden from outside code, the member functions and friends of `noncopyable` can still see and call them. If they are declared but not defined, calling them causes a linker error.
- Although this is a commonly accepted idiom, the intent is not clear unless you understand all of the rules for automatic generation of the special member functions.

In C++11, the non-copyable idiom can be implemented in a way that is more straightforward.

```
struct noncopyable
{
    noncopyable() =default;
    noncopyable(const noncopyable&) =delete;
    noncopyable& operator=(const noncopyable&) =delete;
};
```

Notice how the problems with the pre-C++11 idiom are resolved:

- Generation of the default constructor is still prevented by declaring the copy constructor, but you can bring it back by explicitly defaulting it.

- Explicitly defaulted special member functions are still considered trivial, so there is no performance penalty, and `noncopyable` is not prevented from being a true POD type.
- The copy constructor and copy-assignment operator are public but deleted. It is a compile-time error to define or call a deleted function.
- The intent is clear to anyone who understands `=default` and `=delete`. You don't have to understand the rules for automatic generation of special member functions.

Similar idioms exist for making user-defined types that are non-movable, that can only be dynamically allocated, or that cannot be dynamically allocated. Each of these idioms have pre-C++11 implementations that suffer similar problems, and that are similarly resolved in C++11 by implementing them in terms of defaulted and deleted special member functions.

## Explicitly defaulted functions

You can default any of the special member functions—to explicitly state that the special member function uses the default implementation, to define the special member function with a non-public access qualifier, or to reinstate a special member function whose automatic generation was prevented by other circumstances.

You default a special member function by declaring it as in this example:

```
struct widget
{
    widget()=default;

    inline widget& operator=(const widget&);
};

inline widget& widget::operator=(const widget&) =default;
```

Notice that you can default a special member function outside the body of a class as long as it's inlinable.

Because of the performance benefits of trivial special member functions, we recommend that you prefer automatically generated special member functions over empty function bodies when you want the default behavior. You can do this either by explicitly defaulting the special member function, or by not declaring it (and also not declaring other special member functions that would prevent it from being automatically generated.)

## Deleted functions

You can delete special member functions as well as normal member functions and non-member functions to prevent them from being defined or called. Deleting of special member functions provides a cleaner way of preventing the compiler from generating special member functions that you don't want. The function must be deleted as it is declared; it cannot be deleted afterwards in the way that a function can be declared and then later defaulted.

```

struct widget
{
    // deleted operator new prevents widget from being dynamically allocated.
    void* operator new(std::size_t) = delete;
};

```

Deleting of normal member function or non-member functions prevents problematic type promotions from causing an unintended function to be called. This works because deleted functions still participate in overload resolution and provide a better match than the function that could be called after the types are promoted. The function call resolves to the more-specific—but deleted—function and causes a compiler error.

```

// deleted overload prevents call through type promotion of float to double from
succeeding.
void call_with_true_double_only(float) =delete;
void call_with_true_double_only(double param) { return; }

```

Notice in the preceding sample that calling `call_with_true_double_only` by using a **float** argument would cause a compiler error, but calling `call_with_true_double_only` by using an **int** argument would not; in the **int** case, the argument will be promoted from **int** to **double** and successfully call the **double** version of the function, even though that might not be what's intended. To ensure that any call to this function by using a non-double argument causes a compiler error, you can declare a template version of the function that's deleted.

```

template < typename T >
void call_with_true_double_only(T) =delete; //prevent call through type promotion
of any T to double from succeeding.

void call_with_true_double_only(double param) { return; } // also define for const
double, double&, etc. as needed.

```

# Argument-Dependent Name (Koenig) Lookup on Functions

---

The compiler can use argument-dependent name lookup to find the definition of an unqualified function call. Argument-dependent name lookup is also called Koenig lookup. The type of every argument in a function call is defined within a hierarchy of namespaces, classes, structures, unions, or templates. When you specify an unqualified [postfix](#) function call, the compiler searches for the function definition in the hierarchy associated with each argument type.

## Example

In the sample, the compiler notes that function `f()` takes an argument `x`. Argument `x` is of type `A::X`, which is defined in namespace `A`. The compiler searches namespace `A` and finds a definition for function `f()` that takes an argument of type `A::X`.

```
// argument_dependent_name_koenig_lookup_on_functions.cpp
namespace A
{
    struct X
    {
    };
    void f(const X&)
    {
    }
}
int main()
{
    // The compiler finds A::f() in namespace A, which is where
    // the type of argument x is defined. The type of x is A::X.
    A::X x;
    f(x);
}
```



# Default Arguments

---

In many cases, functions have arguments that are used so infrequently that a default value would suffice. To address this, the default-argument facility allows for specifying only those arguments to a function that are meaningful in a given call. To illustrate this concept, consider the example presented in [Function Overloading](#).

```
// Prototype three print functions.
int print( char *s );           // Print a string.
int print( double dvalue );     // Print a double.
int print( double dvalue, int prec ); // Print a double with a
// given precision.
```

In many applications, a reasonable default can be supplied for `prec`, eliminating the need for two functions:

```
// Prototype two print functions.
int print( char *s );           // Print a string.
int print( double dvalue, int prec=2 ); // Print a double with a
// given precision.
```

The implementation of the `print` function is changed slightly to reflect the fact that only one such function exists for type **double**:

```
// default_arguments.cpp
// compile with: /EHsc /c

// Print a double in specified precision.
// Positive numbers for precision indicate how many digits
// precision after the decimal point to show. Negative
// numbers for precision indicate where to round the number
// to the left of the decimal point.

#include <iostream>
#include <math.h>
using namespace std;

int print( double dvalue, int prec ) {
    // Use table-lookup for rounding/truncation.
    static const double rgPow10[] = {
        10E-7, 10E-6, 10E-5, 10E-4, 10E-3, 10E-2, 10E-1, 10E0,
        10E1, 10E2, 10E3, 10E4, 10E5, 10E6
    };
    const int iPowZero = 6;
    // If precision out of range, just print the number.
    if( prec >= -6 && prec <= 7 )
        // Scale, truncate, then rescale.
        dvalue = floor( dvalue / rgPow10[iPowZero - prec] ) *

```

```

    rgPow10[iPowZero - prec];
    cout << dvalue << endl;
    return cout.good();
}

```

To invoke the new `print` function, use code such as the following:

```

print( d );    // Precision of 2 supplied by default argument.
print( d, 0 ); // Override default argument to achieve other
// results.

```

Note these points when using default arguments:

- Default arguments are used only in function calls where trailing arguments are omitted — they must be the last argument(s). Therefore, the following code is illegal:

```

int print( double dvalue = 0.0, int prec );

```

- A default argument cannot be redefined in later declarations even if the redefinition is identical to the original. Therefore, the following code produces an error:

```

// Prototype for print function.
int print( double dvalue, int prec = 2 );

...

// Definition for print function.
int print( double dvalue, int prec = 2 )
{
    ...
}

```

The problem with this code is that the function declaration in the definition redefines the default argument for `prec`.

- Additional default arguments can be added by later declarations.
- Default arguments can be provided for pointers to functions. For example:

```

int (*pShowIntVal)( int i = 0 );

```

# Inline Functions (C++)

---

A function defined in the body of a class declaration is an inline function.

## Example

In the following class declaration, the `Account` constructor is an inline function. The member functions `GetBalance`, `Deposit`, and `Withdraw` are not specified as **inline** but can be implemented as inline functions.

```
// Inline_Member_Functions.cpp
class Account
{
public:
    Account(double initial_balance) { balance = initial_balance; }
    double GetBalance();
    double Deposit( double Amount );
    double Withdraw( double Amount );
private:
    double balance;
};

inline double Account::GetBalance()
{
    return balance;
}

inline double Account::Deposit( double Amount )
{
    return ( balance += Amount );
}

inline double Account::Withdraw( double Amount )
{
    return ( balance -= Amount );
}

int main()
{
}
```

[!NOTE] In the class declaration, the functions were declared without the **inline** keyword. The **inline** keyword can be specified in the class declaration; the result is the same.

A given inline member function must be declared the same way in every compilation unit. This constraint causes inline functions to behave as if they were instantiated functions. Additionally, there must be exactly one definition of an inline function.

A class member function defaults to external linkage unless a definition for that function contains the **inline** specifier. The preceding example shows that these functions need not be explicitly declared with the **inline**

specifier; using **inline** in the function definition causes it to be an inline function. However, it is illegal to redeclare a function as **inline** after a call to that function.

## Inline, `__inline`, and `__forceinline`

The **inline** and `__inline` specifiers instruct the compiler to insert a copy of the function body into each place the function is called.

The insertion (called inline expansion or inlining) occurs only if the compiler's cost/benefit analysis show it to be profitable. Inline expansion alleviates the function-call overhead at the potential cost of larger code size.

The `__forceinline` keyword overrides the cost/benefit analysis and relies on the judgment of the programmer instead. Exercise caution when using `__forceinline`. Indiscriminate use of `__forceinline` can result in larger code with only marginal performance gains or, in some cases, even performance losses (due to increased paging of a larger executable, for example).

Using inline functions can make your program faster because they eliminate the overhead associated with function calls. Functions expanded inline are subject to code optimizations not available to normal functions.

The compiler treats the inline expansion options and keywords as suggestions. There is no guarantee that functions will be inlined. You cannot force the compiler to inline a particular function, even with the `__forceinline` keyword. When compiling with `/clr`, the compiler will not inline a function if there are security attributes applied to the function.

The **inline** keyword is available only in C++. The `__inline` and `__forceinline` keywords are available in both C and C++. For compatibility with previous versions, `__inline` and `__forceinline` are synonyms for `__inline`, and `__forceinline` unless compiler option `/Za (Disable language extensions)` is specified.

The **inline** keyword tells the compiler that inline expansion is preferred. However, the compiler can create a separate instance of the function (instantiate) and create standard calling linkages instead of inserting the code inline. Two cases where this can happen are:

- Recursive functions.
- Functions that are referred to through a pointer elsewhere in the translation unit.

These reasons may interfere with inlining, *as may others*, at the discretion of the compiler; you should not depend on the **inline** specifier to cause a function to be inlined.

As with normal functions, there is no defined order of evaluation of the arguments to an inline function. In fact, it could be different from the order in which the arguments are evaluated when passed using normal function call protocol.

The `/Ob` compiler optimization option helps to determine whether inline function expansion actually occurs.

`/LTCG` performs cross-module inlining regardless of whether it was requested in source code.

### Example 1

```
// inline_keyword1.cpp
// compile with: /c
```

```
inline int max( int a , int b ) {
    if( a > b )
        return a;
    return b;
}
```

A class's member functions can be declared inline either by using the **inline** keyword or by placing the function definition within the class definition.

## Example 2

```
// inline_keyword2.cpp
// compile with: /EHsc /c
#include <iostream>
using namespace std;

class MyClass {
public:
    void print() { cout << i << ' '; } // Implicitly inline
private:
    int i;
};
```

## Microsoft Specific

The **\_\_inline** keyword is equivalent to **inline**.

Even with **\_\_forceinline**, the compiler cannot inline code in all circumstances. The compiler cannot inline a function if:

- The function or its caller is compiled with /Ob0 (the default option for debug builds).
- The function and the caller use different types of exception handling (C++ exception handling in one, structured exception handling in the other).
- The function has a variable argument list.
- The function uses inline assembly, unless compiled with /Og, /Ox, /O1, or /O2.
- The function is recursive and not accompanied by **#pragma inline\_recursion(on)**. With the pragma, recursive functions are inlined to a default depth of 16 calls. To reduce the inlining depth, use [inline\\_depth](#) pragma.
- The function is virtual and is called virtually. Direct calls to virtual functions can be inlined.
- The program takes the address of the function and the call is made via the pointer to the function. Direct calls to functions that have had their address taken can be inlined.
- The function is also marked with the [naked \\_\\_declspec](#) modifier.

If the compiler cannot inline a function declared with **\_\_forceinline**, it generates a level 1 warning, except when:

- The function is compiled by using /Od or /Ob0. No inlining is expected in these cases.
- The function is defined externally, in an included library or another translation unit, or is a virtual call target or indirect call target. The compiler can't identify non-inlined code that it can't find in the current translation unit.

Recursive functions can be substituted inline to a depth specified by the [inline\\_depth](#) pragma, up to a maximum of 16 calls. After that depth, recursive function calls are treated as calls to an instance of the function. The depth to which recursive functions are examined by the inline heuristic cannot exceed 16. The [inline\\_recursion](#) pragma controls the inline expansion of a function currently under expansion. See the [Inline-Function Expansion \(/Ob\)](#) compiler option for related information.

### END Microsoft Specific

For more information on using the **inline** specifier, see:

- [Inline Class Member Functions](#)
- [Defining Inline C++ Functions with dllexport and dllimport](#)

## When to use inline functions

Inline functions are best used for small functions such as accessing private data members. The main purpose of these one- or two-line "accessor" functions is to return state information about objects; short functions are sensitive to the overhead of function calls. Longer functions spend proportionately less time in the calling/returning sequence and benefit less from inlining.

A [Point](#) class can be defined as follows:

```
// when_to_use_inline_functions.cpp
class Point
{
public:
    // Define "accessor" functions as
    // reference types.
    unsigned& x();
    unsigned& y();
private:
    unsigned _x;
    unsigned _y;
};

inline unsigned& Point::x()
{
    return _x;
}
inline unsigned& Point::y()
{
    return _y;
}
```

```

}
int main()
{
}

```

Assuming coordinate manipulation is a relatively common operation in a client of such a class, specifying the two accessor functions (`x` and `y` in the preceding example) as **inline** typically saves the overhead on:

- Function calls (including parameter passing and placing the object's address on the stack)
- Preservation of caller's stack frame
- New stack-frame setup
- Return-value communication
- Old stack-frame restore
- Return

## Inline functions vs. macros

Although inline functions are similar to macros (because the function code is expanded at the point of the call at compile time), inline functions are parsed by the compiler, whereas macros are expanded by the preprocessor. As a result, there are several important differences:

- Inline functions follow all the protocols of type safety enforced on normal functions.
- Inline functions are specified using the same syntax as any other function except that they include the **inline** keyword in the function declaration.
- Expressions passed as arguments to inline functions are evaluated once. In some cases, expressions passed as arguments to macros can be evaluated more than once.

The following example shows a macro that converts lowercase letters to uppercase:

```

// inline_functions_macro.c
#include <stdio.h>
#include <conio.h>

#define toupper(a) ((a) >= 'a' && ((a) <= 'z') ? ((a)-('a'-'A')):(a))

int main() {
    char ch;
    printf_s("Enter a character: ");
    ch = toupper( getc(stdin) );
    printf_s( "%c", ch );
}
// Sample Input:  xyz
// Sample Output:  Z

```

The intent of the expression `toupper(getc(stdin))` is that a character should be read from the console device (`stdin`) and, if necessary, converted to uppercase.

Because of the implementation of the macro, `getc` is executed once to determine whether the character is greater than or equal to "a," and once to determine whether it is less than or equal to "z." If it is in that range, `getc` is executed again to convert the character to uppercase. This means the program waits for two or three characters when, ideally, it should wait for only one.

Inline functions remedy the problem previously described:

```
// inline_functions_inline.cpp
#include <stdio.h>
#include <conio.h>

inline char toupper( char a ) {
    return ((a >= 'a' && a <= 'z') ? a - ('a' - 'A') : a );
}

int main() {
    printf_s("Enter a character: ");
    char ch = toupper( getc(stdin) );
    printf_s( "%c", ch );
}
```

Sample Input: a  
Sample Output: A

## See also

[noinline](#)

[auto\\_inline](#)



# Operator overloading

---

The **operator** keyword declares a function specifying what *operator-symbol* means when applied to instances of a class. This gives the operator more than one meaning, or "overloads" it. The compiler distinguishes between the different meanings of an operator by examining the types of its operands.

## Syntax

```
type operator operator-symbol ( parameter-list )
```

## Remarks

You can redefine the function of most built-in operators globally or on a class-by-class basis. Overloaded operators are implemented as functions.

The name of an overloaded operator is **operator** *x*, where *x* is the operator as it appears in the following table. For example, to overload the addition operator, you define a function called **operator+**. Similarly, to overload the addition/assignment operator, **+=**, define a function called **operator+=**.

## Redefinable Operators

Operator	Name	Type
,	Comma	Binary
!	Logical NOT	Unary
!=	Inequality	Binary
%	Modulus	Binary
%=	Modulus assignment	Binary
&	Bitwise AND	Binary
&	Address-of	Unary
&&	Logical AND	Binary
&=	Bitwise AND assignment	Binary
()	Function call	—
()	Cast Operator	Unary
*	Multiplication	Binary
*	Pointer dereference	Unary
*=	Multiplication assignment	Binary
+	Addition	Binary
+	Unary Plus	Unary

Operator	Name	Type
++	Increment <sup>1</sup>	Unary
+=	Addition assignment	Binary
-	Subtraction	Binary
-	Unary negation	Unary
--	Decrement <sup>1</sup>	Unary
-=	Subtraction assignment	Binary
->	Member selection	Binary
->*	Pointer-to-member selection	Binary
/	Division	Binary
/=	Division assignment	Binary
<	Less than	Binary
<<	Left shift	Binary
<<=	Left shift assignment	Binary
<=	Less than or equal to	Binary
=	Assignment	Binary
==	Equality	Binary
>	Greater than	Binary
>=	Greater than or equal to	Binary
>>	Right shift	Binary
>>=	Right shift assignment	Binary
[ ]	Array subscript	—
^	Exclusive OR	Binary
^=	Exclusive OR assignment	Binary
	Bitwise inclusive OR	Binary
=	Bitwise inclusive OR assignment	Binary
	Logical OR	Binary
~	One's complement	Unary
<b>delete</b>	Delete	—
<b>new</b>	New	—

Operator	Name	Type
conversion operators	conversion operators	Unary

<sup>1</sup> Two versions of the unary increment and decrement operators exist: preincrement and postincrement.

See [General Rules for Operator Overloading](#) for more information. The constraints on the various categories of overloaded operators are described in the following topics:

- [Unary Operators](#)
- [Binary Operators](#)
- [Assignment](#)
- [Function Call](#)
- [Subscripting](#)
- [Class-Member Access](#)
- [Increment and Decrement](#).
- [User-Defined Type Conversions](#)

The operators shown in the following table cannot be overloaded. The table includes the preprocessor symbols `#` and `##`.

## Nonredefinable Operators

Operator	Name
.	Member selection
.*	Pointer-to-member selection
::	Scope resolution
? :	Conditional
#	Preprocessor convert to string
##	Preprocessor concatenate

Although overloaded operators are usually called implicitly by the compiler when they are encountered in code, they can be invoked explicitly the same way as any member or nonmember function is called:

```
Point pt;
pt.operator+( 3 ); // Call addition operator to add 3 to pt.
```

## Example

The following example overloads the + operator to add two complex numbers and returns the result.

```
// operator_overloading.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

struct Complex {
    Complex( double r, double i ) : re(r), im(i) {}
    Complex operator+( Complex &other );
    void Display( ) { cout << re << ", " << im << endl; }
private:
    double re, im;
};

// Operator overloaded using a member function
Complex Complex::operator+( Complex &other ) {
    return Complex( re + other.re, im + other.im );
}

int main() {
    Complex a = Complex( 1.2, 3.4 );
    Complex b = Complex( 5.6, 7.8 );
    Complex c = Complex( 0.0, 0.0 );

    c = a + b;
    c.Display();
}
```

6.8, 11.2

## In this section

- [General Rules for Operator Overloading](#)
- [Overloading Unary Operators](#)
- [Binary Operators](#)
- [Assignment](#)
- [Function Call](#)
- [Subscripting](#)
- [Member Access](#)

## See also

## C++ Built-in Operators, Precedence and Associativity

### Keywords

# General Rules for Operator Overloading

---

The following rules constrain how overloaded operators are implemented. However, they do not apply to the `new` and `delete` operators, which are covered separately.

- You cannot define new operators, such as `..`.
- You cannot redefine the meaning of operators when applied to built-in data types.
- Overloaded operators must either be a nonstatic class member function or a global function. A global function that needs access to private or protected class members must be declared as a friend of that class. A global function must take at least one argument that is of class or enumerated type or that is a reference to a class or enumerated type. For example:

```
// rules_for_operator_overloading.cpp
class Point
{
public:
    Point operator<( Point & ); // Declare a member operator
                                // overload.

    // Declare addition operators.
    friend Point operator+( Point&, int );
    friend Point operator+( int, Point& );
};

int main()
{
}
```

The preceding code sample declares the less-than operator as a member function; however, the addition operators are declared as global functions that have friend access. Note that more than one implementation can be provided for a given operator. In the case of the preceding addition operator, the two implementations are provided to facilitate commutativity. It is just as likely that operators that add a `Point` to a `Point`, `int` to a `Point`, and so on, might be implemented.

- Operators obey the precedence, grouping, and number of operands dictated by their typical use with built-in types. Therefore, there is no way to express the concept "add 2 and 3 to an object of type `Point`," expecting 2 to be added to the x coordinate and 3 to be added to the y coordinate.
- Unary operators declared as member functions take no arguments; if declared as global functions, they take one argument.
- Binary operators declared as member functions take one argument; if declared as global functions, they take two arguments.
- If an operator can be used as either a unary or a binary operator (`&`, `*`, `+`, and `-`), you can overload each use separately.

- Overloaded operators cannot have default arguments.
- All overloaded operators except assignment (**operator=**) are inherited by derived classes.
- The first argument for member-function overloaded operators is always of the class type of the object for which the operator is invoked (the class in which the operator is declared, or a class derived from that class). No conversions are supplied for the first argument.

Note that the meaning of any of the operators can be changed completely. That includes the meaning of the address-of (&), assignment (=), and function-call operators. Also, identities that can be relied upon for built-in types can be changed using operator overloading. For example, the following four statements are usually equivalent when completely evaluated:

```
var = var + 1;  
var += 1;  
var++;  
++var;
```

This identity cannot be relied upon for class types that overload operators. Moreover, some of the requirements implicit in the use of these operators for basic types are relaxed for overloaded operators. For example, the addition/assignment operator, `+=`, requires the left operand to be an l-value when applied to basic types; there is no such requirement when the operator is overloaded.

[!NOTE] For consistency, it is often best to follow the model of the built-in types when defining overloaded operators. If the semantics of an overloaded operator differ significantly from its meaning in other contexts, it can be more confusing than useful.

## See also

[Operator Overloading](#)

# Overloading Unary Operators

---

The unary operators that can be overloaded are the following:

1. `!` ([logical NOT](#))
2. `&` ([address-of](#))
3. `~` ([one's complement](#))
4. `*` ([pointer dereference](#))
5. `+` ([unary plus](#))
6. `-` ([unary negation](#))
7. `++` ([increment](#))
8. `--` ([decrement](#))
9. conversion operators

The postfix increment and decrement operators (`++` and `--`) are treated separately in [Increment and Decrement](#).

Conversion operators are also discussed in a separate topic; see [User-Defined Type Conversions](#).

The following rules are true of all other unary operators. To declare a unary operator function as a nonstatic member, you must declare it in the form:

```
ret-type operator op ()
```

where *ret-type* is the return type and *op* is one of the operators listed in the preceding table.

To declare a unary operator function as a global function, you must declare it in the form:

```
ret-type operator op ( arg )
```

where *ret-type* and *op* are as described for member operator functions and the *arg* is an argument of class type on which to operate.

[!NOTE] There is no restriction on the return types of the unary operators. For example, it makes sense for logical NOT (`!`) to return an integral value, but this is not enforced.

## See also

[Operator Overloading](#)



# Increment and Decrement Operator Overloading (C++)

---

The increment and decrement operators fall into a special category because there are two variants of each:

- Preincrement and postincrement
- Predecrement and postdecrement

When you write overloaded operator functions, it can be useful to implement separate versions for the prefix and postfix versions of these operators. To distinguish between the two, the following rule is observed: The prefix form of the operator is declared exactly the same way as any other unary operator; the postfix form accepts an additional argument of type **int**.

[!NOTE] When specifying an overloaded operator for the postfix form of the increment or decrement operator, the additional argument must be of type **int**; specifying any other type generates an error.

The following example shows how to define prefix and postfix increment and decrement operators for the **Point** class:

```
// increment_and_decrement1.cpp
class Point
{
public:
    // Declare prefix and postfix increment operators.
    Point& operator++();      // Prefix increment operator.
    Point operator++(int);    // Postfix increment operator.

    // Declare prefix and postfix decrement operators.
    Point& operator--();      // Prefix decrement operator.
    Point operator--(int);    // Postfix decrement operator.

    // Define default constructor.
    Point() { _x = _y = 0; }

    // Define accessor functions.
    int x() { return _x; }
    int y() { return _y; }
private:
    int _x, _y;
};

// Define prefix increment operator.
Point& Point::operator++()
{
    _x++;
    _y++;
    return *this;
}
```

```

// Define postfix increment operator.
Point Point::operator++(int)
{
    Point temp = *this;
    ++*this;
    return temp;
}

// Define prefix decrement operator.
Point& Point::operator--()
{
    _x--;
    _y--;
    return *this;
}

// Define postfix decrement operator.
Point Point::operator--(int)
{
    Point temp = *this;
    --*this;
    return temp;
}
int main()
{
}

```

The same operators can be defined in file scope (globally) using the following function heads:

```

friend Point& operator++( Point& )      // Prefix increment
friend Point& operator++( Point&, int ) // Postfix increment
friend Point& operator--( Point& )      // Prefix decrement
friend Point& operator--( Point&, int ) // Postfix decrement

```

The argument of type **int** that denotes the postfix form of the increment or decrement operator is not commonly used to pass arguments. It usually contains the value 0. However, it can be used as follows:

```

// increment_and_decrement2.cpp
class Int
{
public:
    Int &operator++( int n );
private:
    int _i;
};

Int& Int::operator++( int n )
{

```

```
    if( n != 0 )    // Handle case where an argument is passed.
        _i += n;
    else
        _i++;      // Handle case where no argument is passed.
    return *this;
}
int main()
{
    Int i;
    i.operator++( 25 ); // Increment by 25.
}
```

There is no syntax for using the increment or decrement operators to pass these values other than explicit invocation, as shown in the preceding code. A more straightforward way to implement this functionality is to overload the addition/assignment operator (`+=`).

## See also

[Operator Overloading](#)

# Binary Operators

---

The following table shows a list of operators that can be overloaded.

## Redefinable Binary Operators

Operator	Name
,	Comma
!=	Inequality
%	Modulus
%=	Modulus/assignment
&	Bitwise AND
&&	Logical AND
&=	Bitwise AND/assignment
*	Multiplication
*=	Multiplication/assignment
+	Addition
+=	Addition/assignment
-	Subtraction
-=	Subtraction/assignment
->	Member selection
->*	Pointer-to-member selection
/	Division
/=	Division/assignment
<	Less than
<<	Left shift
<<=	Left shift/assignment
<=	Less than or equal to
=	Assignment
==	Equality
>	Greater than
>=	Greater than or equal to

Operator	Name
>>	Right shift
>>=	Right shift/assignment
^	Exclusive OR
^=	Exclusive OR/assignment
	Bitwise inclusive OR
=	Bitwise inclusive OR/assignment
	Logical OR

To declare a binary operator function as a nonstatic member, you must declare it in the form:

```
ret-type operator op ( arg )
```

where *ret-type* is the return type, *op* is one of the operators listed in the preceding table, and *arg* is an argument of any type.

To declare a binary operator function as a global function, you must declare it in the form:

```
ret-type operator op ( arg1, arg2 )
```

where *ret-type* and *op* are as described for member operator functions and *arg1* and *arg2* are arguments. At least one of the arguments must be of class type.

[!NOTE] There is no restriction on the return types of the binary operators; however, most user-defined binary operators return either a class type or a reference to a class type.

## See also

[Operator Overloading](#)

# Assignment

---

The assignment operator (=) is, strictly speaking, a binary operator. Its declaration is identical to any other binary operator, with the following exceptions:

- It must be a nonstatic member function. No **operator=** can be declared as a nonmember function.
- It is not inherited by derived classes.
- A default **operator=** function can be generated by the compiler for class types, if none exists.

The following example illustrates how to declare an assignment operator:

```
class Point
{
public:
    int _x, _y;

    // Right side of copy assignment is the argument.
    Point& operator=(const Point&);
};

// Define copy assignment operator.
Point& Point::operator=(const Point& otherPoint)
{
    _x = otherPoint._x;
    _y = otherPoint._y;

    // Assignment operator returns left side of assignment.
    return *this;
}

int main()
{
    Point pt1, pt2;
    pt1 = pt2;
}
```

The supplied argument is the right side of the expression. The operator returns the object to preserve the behavior of the assignment operator, which returns the value of the left side after the assignment is complete. This allows chaining of assignments, such as:

```
pt1 = pt2 = pt3;
```

The copy assignment operator is not to be confused with the copy constructor. The latter is called during the construction of a new object from an existing one:

```
// Copy constructor is called--not overloaded copy assignment operator!  
Point pt3 = pt1;  
  
// The previous initialization is similar to the following:  
Point pt4(pt1); // Copy constructor call.
```

[!NOTE] It is advisable to follow the [rule of three](#) that a class which defines a copy assignment operator should also explicitly define copy constructor, destructor, and, starting with C++11, move constructor and move assignment operator.

## See also

- [Operator Overloading](#)
- [Copy Constructors and Copy Assignment Operators \(C++\)](#)

# Function Call (C++)

---

The function-call operator, invoked using parentheses, is a binary operator.

## Syntax

```
primary-expression ( expression-list )
```

## Remarks

In this context, `primary-expression` is the first operand, and `expression-list`, a possibly empty list of arguments, is the second operand. The function-call operator is used for operations that require a number of parameters. This works because `expression-list` is a list instead of a single operand. The function-call operator must be a nonstatic member function.

The function-call operator, when overloaded, does not modify how functions are called; rather, it modifies how the operator is to be interpreted when applied to objects of a given class type. For example, the following code would usually be meaningless:

```
Point pt;  
pt( 3, 2 );
```

Given an appropriate overloaded function-call operator, however, this syntax can be used to offset the `x` coordinate 3 units and the `y` coordinate 2 units. The following code shows such a definition:

```
// function_call.cpp  
class Point  
{  
public:  
    Point() { _x = _y = 0; }  
    Point &operator()( int dx, int dy )  
        { _x += dx; _y += dy; return *this; }  
private:  
    int _x, _y;  
};  
  
int main()  
{  
    Point pt;  
    pt( 3, 2 );  
}
```

Note that the function-call operator is applied to the name of an object, not the name of a function.



You can also overload the function call operator using a pointer to a function (rather than the function itself).

```
typedef void(*ptf)();
void func()
{
}
struct S
{
    operator ptf()
    {
        return func;
    }
};

int main()
{
    S s;
    s();//operates as s.operator ptf>()
}
```

See also

[Operator Overloading](#)

# Subscripting

---

The subscript operator (`[]`), like the function-call operator, is considered a binary operator. The subscript operator must be a nonstatic member function that takes a single argument. This argument can be of any type and designates the desired array subscript.

## Example

The following example demonstrates how to create a vector of type **int** that implements bounds checking:

```
// subscripting.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;
class IntVector {
public:
    IntVector( int cElements );
    ~IntVector() { delete [] _iElements; }
    int& operator[](int nSubscript);
private:
    int *_iElements;
    int _iUpperBound;
};

// Construct an IntVector.
IntVector::IntVector( int cElements ) {
    _iElements = new int[cElements];
    _iUpperBound = cElements;
}

// Subscript operator for IntVector.
int& IntVector::operator[](int nSubscript) {
    static int iErr = -1;

    if( nSubscript >= 0 && nSubscript < _iUpperBound )
        return _iElements[nSubscript];
    else {
        clog << "Array bounds violation." << endl;
        return iErr;
    }
}

// Test the IntVector class.
int main() {
    IntVector v( 10 );
    int i;

    for( i = 0; i <= 10; ++i )
        v[i] = i;
```

```
v[3] = v[9];

for ( i = 0; i <= 10; ++i )
    cout << "Element: [" << i << "] = " << v[i] << endl;
}
```

```
Array bounds violation.
Element: [0] = 0
Element: [1] = 1
Element: [2] = 2
Element: [3] = 9
Element: [4] = 4
Element: [5] = 5
Element: [6] = 6
Element: [7] = 7
Element: [8] = 8
Element: [9] = 9
Array bounds violation.
Element: [10] = 10
```

## Comments

When `i` reaches 10 in the preceding program, **`operator[]`** detects that an out-of-bounds subscript is being used and issues an error message.

Note that the function **`operator[]`** returns a reference type. This causes it to be an l-value, allowing you to use subscripted expressions on either side of assignment operators.

## See also

[Operator Overloading](#)

# Member Access

---

Class member access can be controlled by overloading the member access operator (->). This operator is considered a unary operator in this usage, and the overloaded operator function must be a class member function. Therefore, the declaration for such a function is:

## Syntax

```
class-type *operator->()
```

## Remarks

where *class-type* is the name of the class to which this operator belongs. The member access operator function must be a nonstatic member function.

This operator is used (often in conjunction with the pointer-dereference operator) to implement "smart pointers" that validate pointers prior to dereference or count usage.

The . member access operator cannot be overloaded.

## See also

[Operator Overloading](#)

# Classes and Structs (C++)

---

This section introduces C++ classes and structs. The two constructs are identical in C++ except that in structs the default accessibility is public, whereas in classes the default is private.

Classes and structs are the constructs whereby you define your own types. Classes and structs can both contain data members and member functions, which enable you to describe the type's state and behavior.

The following topics are included:

- [class](#)
- [struct](#)
- [Class Member Overview](#)
- [Member Access Control](#)
- [Inheritance](#)
- [Static Members](#)
- [User-Defined Type Conversions](#)
- [Mutable Data Members \(mutable specifier\)](#)
- [Nested Class Declarations](#)
- [Anonymous Class Types](#)
- [Pointers to Members](#)
- [this Pointer](#)
- [C++ Bit Fields](#)

The three class types are structure, class, and union. They are declared using the [struct](#), [class](#), and [union](#) keywords. The following table shows the differences among the three class types.

For more information on unions, see [Unions](#). For information on classes and structs in C++/CLI and C++/CX, see [Classes and Structs](#).

## Access Control and Constraints of Structures, Classes and Unions

Structures	Classes	Unions
class key is <b>struct</b>	class key is <b>class</b>	class key is <b>union</b>
Default access is public	Default access is private	Default access is public
No usage constraints	No usage constraints	Use only one member at a time

See also



# class (C++)

---

The **class** keyword declares a class type or defines an object of a class type.

## Syntax

```
[template-spec]
class [ms-decl-spec] [tag [: base-list ]]
{
    member-list
} [declarators];
[ class ] tag declarators;
```

### Parameters

*template-spec*

Optional template specifications. For more information, refer to [Templates](#).

*class*

The **class** keyword.

*ms-decl-spec*

Optional storage-class specification. For more information, refer to the [\\_\\_declspec](#) keyword.

*tag*

The type name given to the class. The tag becomes a reserved word within the scope of the class. The tag is optional. If omitted, an anonymous class is defined. For more information, see [Anonymous Class Types](#).

*base-list*

Optional list of classes or structures this class will derive its members from. See [Base Classes](#) for more information. Each base class or structure name can be preceded by an access specifier ([public](#), [private](#), [protected](#)) and the [virtual](#) keyword. See the member-access table in [Controlling Access to Class Members](#) for more information.

*member-list*

List of class members. Refer to [Class Member Overview](#) for more information.

*declarators*

Declarator list specifying the names of one or more instances of the class type. Declarators may include initializer lists if all data members of the class are **public**. This is more common in structures, whose data members are **public** by default, than in classes. See [Overview of Declarators](#) for more information.

## Remarks

For more information on classes in general, refer to one of the following topics:

- [struct](#)

- [union](#)
- [\\_\\_multiple\\_inheritance](#)
- [\\_\\_single\\_inheritance](#)
- [\\_\\_virtual\\_inheritance](#)

For information on managed classes and structs in C++/CLI and C++/CX, see [Classes and Structs](#)

## Example

```
// class.cpp
// compile with: /EHsc
// Example of the class keyword
// Exhibits polymorphism/virtual functions.

#include <iostream>
#include <string>
#define TRUE = 1
using namespace std;

class dog
{
public:
    dog()
    {
        _legs = 4;
        _bark = true;
    }

    void setDogSize(string dogSize)
    {
        _dogSize = dogSize;
    }
    virtual void setEars(string type)    // virtual function
    {
        _earType = type;
    }

private:
    string _dogSize, _earType;
    int _legs;
    bool _bark;
};

class breed : public dog
{
public:
    breed( string color, string size)
    {
        _color = color;
    }
};
```



```

        setDogSize(size);
    }

    string getColor()
    {
        return _color;
    }

    // virtual function redefined
    void setEars(string length, string type)
    {
        _earLength = length;
        _earType = type;
    }

protected:
    string _color, _earLength, _earType;
};

int main()
{
    dog mongrel;
    breed labrador("yellow", "large");
    mongrel.setEars("pointy");
    labrador.setEars("long", "floppy");
    cout << "Cody is a " << labrador.getColor() << " labrador" << endl;
}

```

## See also

[Keywords](#)

[Classes and Structs](#)

# struct (C++)

---

The **struct** keyword defines a structure type and/or a variable of a structure type.

## Syntax

```
[template-spec] struct [ms-decl-spec] [tag [: base-list ]]  
{  
    member-list  
} [declarators];  
[struct] tag declarators;
```

### Parameters

#### *template-spec*

Optional template specifications. For more information, refer to [Template Specifications](#).

#### *struct*

The **struct** keyword.

#### *ms-decl-spec*

Optional storage-class specification. For more information, refer to the [\\_\\_declspec](#) keyword.

#### *tag*

The type name given to the structure. The tag becomes a reserved word within the scope of the structure. The tag is optional. If omitted, an anonymous structure is defined. For more information, see [Anonymous Class Types](#).

#### *base-list*

Optional list of classes or structures this structure will derive its members from. See [Base Classes](#) for more information. Each base class or structure name can be preceded by an access specifier ([public](#), [private](#), [protected](#)) and the [virtual](#) keyword. See the member-access table in [Controlling Access to Class Members](#) for more information.

#### *member-list*

List of structure members. Refer to [Class Member Overview](#) for more information. The only difference here is that **struct** is used in place of **class**.

#### *declarators*

Declarator list specifying the names of the structure. Declarator lists declare one or more instances of the structure type. Declarators may include initializer lists if all data members of the structure are **public**. Initializer lists are common in structures because data members are **public** by default. See [Overview of Declarators](#) for more information.

## Remarks

A structure type is a user-defined composite type. It is composed of fields or members that can have different types.

In C++, a structure is the same as a class except that its members are **public** by default.

For information on managed classes and structs in C++/CLI, see [Classes and Structs](#).

## Using a Structure

In C, you must explicitly use the **struct** keyword to declare a structure. In C++, you do not need to use the **struct** keyword after the type has been defined.

You have the option of declaring variables when the structure type is defined by placing one or more comma-separated variable names between the closing brace and the semicolon.

Structure variables can be initialized. The initialization for each variable must be enclosed in braces.

For related information, see [class](#), [union](#), and [enum](#).

## Example

```
#include <iostream>
using namespace std;

struct PERSON {    // Declare PERSON struct type
    int age;        // Declare member types
    long ss;
    float weight;
    char name[25];
} family_member;    // Define object of type PERSON

struct CELL {    // Declare CELL bit field
    unsigned short character : 8;    // 00000000 ????????
    unsigned short foreground : 3;    // 00000??? 00000000
    unsigned short intensity : 1;    // 0000?000 00000000
    unsigned short background : 3;    // 0???0000 00000000
    unsigned short blink : 1;    // ?0000000 00000000
} screen[25][80];    // Array of bit fields

int main() {
    struct PERSON sister;    // C style structure declaration
    PERSON brother;    // C++ style structure declaration
    sister.age = 13;    // assign values to members
    brother.age = 7;
    cout << "sister.age = " << sister.age << '\n';
    cout << "brother.age = " << brother.age << '\n';

    CELL my_cell;
    my_cell.character = 1;
    cout << "my_cell.character = " << my_cell.character;
}
// Output:
```

```
// sister.age = 13  
// brother.age = 7  
// my_cell.character = 1
```

# Class Member Overview

---

A class or struct consists of its members. The work that a class does is performed by its member functions. The state that it maintains is stored in its data members. Initialization of members is done by constructors, and cleanup work such as freeing of memory and releasing of resources is done by destructors. In C++11 and later, data members can (and usually should) be initialized at the point of declaration.

## Kinds of class members

The full list of member categories is as follows:

- [Special Member Functions](#).
- [Overview of Member Functions](#).
- [Data members](#) including built-in types and other user defined types.
- Operators
- [Nested Class Declarations](#) and.)
- [Unions](#)
- [Enumerations](#).
- [Bit fields](#).
- [Friends](#).
- [Aliases and typedefs](#).

[!NOTE] Friends are included in the preceding list because they are contained in the class declaration. However, they are not true class members, because they are not in the scope of the class.

## Example class declaration

The following example shows a simple class declaration:

```
// TestRun.h

class TestRun
{
    // Start member list.

    //The class interface accessible to all callers.
public:
    // Use compiler-generated default constructor:
    TestRun() = default;
    // Don't generate a copy constructor:
```

```

TestRun(const TestRun&) = delete;
TestRun(std::string name);
void DoSomething();
int Calculate(int a, double d);
virtual ~TestRun();
enum class State { Active, Suspended };

// Accessible to this class and derived classes only.
protected:
    virtual void Initialize();
    virtual void Suspend();
    State GetState();

// Accessible to this class only.
private:
    // Default brace-initialization of instance members:
    State _state{ State::Suspended };
    std::string _testName{ "" };
    int _index{ 0 };

    // Non-const static member:
    static int _instances;
    // End member list.
};

// Define and initialize static member.
int TestRun::_instances{ 0 };

```

## Member accessibility

The members of a class are declared in the member list. The member list of a class may be divided into any number of **private**, **protected** and **public** sections using keywords known as access specifiers. A colon **:** must follow the access specifier. These sections need not be contiguous, that is, any of these keywords may appear several times in the member list. The keyword designates the access of all members up until the next access specifier or the closing brace. For more information, see [Member Access Control \(C++\)](#).

## Static members

A data member may be declared as static, which means all objects of the class have access to the same copy of it. A member function may be declared as static, in which case it can only access static data members of the class (and has no *this* pointer). For more information, see [Static Data Members](#).

## Special member functions

Special member functions are functions that are automatically provided by the compiler if you do not specify them in your source code.

1. Default constructor
2. Copy constructor

3. **(C++11)** Move constructor
4. Copy assignment operator
5. **(C++11)** Move assignment operator
6. Destructor

For more information, see [Special Member Functions](#).

## Memberwise initialization

In C++11 and later, non-static member declarators can contain initializers.

```
class CanInit
{
public:
    long num {7};           // OK in C++11
    int k = 9;              // OK in C++11
    static int i = 9;       // Error: must be defined and initialized
                           // outside of class declaration.

    // initializes num to 7 and k to 9
    CanInit(){}

    // overwrites original initialized value of num:
    CanInit(int val) : num(val) {}
};

int main()
{
}
```

If a member is assigned a value in a constructor, that value overwrites the value with which the member was initialized at the point of declaration.

There is only one shared copy of static data members for all objects of a given class type. Static data members must be defined and can be initialized at file scope. (For more information about static data members, see [Static Data Members](#).) The following example shows how to perform these initializations:

```
// class_members2.cpp
class CanInit2
{
public:
    CanInit2() {} // Initializes num to 7 when new objects of type
                  // CanInit are created.

    long    num {7};
    static int i;
    static int j;
};
```

```
// At file scope:  
  
// i is defined at file scope and initialized to 15.  
// The initializer is evaluated in the scope of CanInit.  
int CanInit2::i = 15;  
  
// The right side of the initializer is in the scope  
// of the object being initialized  
int CanInit2::j = i;
```

[!NOTE] The class name, `CanInit2`, must precede `i` to specify that the `i` being defined is a member of class `CanInit2`.

## See also

[Classes and Structs](#)



# Member Access Control (C++)

Access controls enable you to separate the **public** interface of a class from the **private** implementation details and the **protected** members that are only for use by derived classes. The access specifier applies to all members declared after it until the next access specifier is encountered.

```
class Point
{
public:
    Point( int, int ) // Declare public constructor.;
    Point(); // Declare public default constructor.
    int &x( int ); // Declare public accessor.
    int &y( int ); // Declare public accessor.

private:
    // Declare private state variables.
    int _x;
    int _y;

protected:
    // Declare protected function for derived classes only.
    Point ToWindowCoords();
};
```

The default access is **private** in a class, and **public** in a struct or union. Access specifiers in a class can be used any number of times in any order. The allocation of storage for objects of class types is implementation dependent, but members are guaranteed to be assigned successively higher memory addresses between access specifiers.

## Member-Access Control

Type of Access	Meaning
private	Class members declared as <b>private</b> can be used only by member functions and friends (classes or functions) of the class.
protected	Class members declared as <b>protected</b> can be used by member functions and friends (classes or functions) of the class. Additionally, they can be used by classes derived from the class.
public	Class members declared as <b>public</b> can be used by any function.

Access control helps prevent you from using objects in ways they were not intended to be used. This protection is lost when explicit type conversions (casts) are performed.

[!NOTE] Access control is equally applicable to all names: member functions, member data, nested classes, and enumerators.

## Access Control in Derived Classes

Two factors control which members of a base class are accessible in a derived class; these same factors control access to the inherited members in the derived class:

- Whether the derived class declares the base class using the **public** access specifier.
- What the access to the member is in the base class.

The following table shows the interaction between these factors and how to determine base-class member access.

Member Access in Base Class

private	protected	Public
Always inaccessible regardless of derivation access	Private in derived class if you use private derivation	Private in derived class if you use private derivation
	Protected in derived class if you use protected derivation	Protected in derived class if you use protected derivation
	Protected in derived class if you use public derivation	Public in derived class if you use public derivation

The following example illustrates this:

```
// access_specifiers_for_base_classes.cpp
class BaseClass
{
public:
    int PublicFunc(); // Declare a public member.
protected:
    int ProtectedFunc(); // Declare a protected member.
private:
    int PrivateFunc(); // Declare a private member.
};

// Declare two classes derived from BaseClass.
class DerivedClass1 : public BaseClass
{
    void foo()
    {
        PublicFunc();
        ProtectedFunc();
        PrivateFunc(); // function is inaccessible
    }
};

class DerivedClass2 : private BaseClass
{
    void foo()
    {
        PublicFunc();
    }
};
```

```

        ProtectedFunc();
        PrivateFunc(); // function is inaccessible
    }
};

int main()
{
    DerivedClass1 derived_class1;
    DerivedClass2 derived_class2;
    derived_class1.PublicFunc();
    derived_class2.PublicFunc(); // function is inaccessible
}

```

In **DerivedClass1**, the member function **PublicFunc** is a public member and **ProtectedFunc** is a protected member because **BaseClass** is a public base class. **PrivateFunc** is private to **BaseClass**, and it is inaccessible to any derived classes.

In **DerivedClass2**, the functions **PublicFunc** and **ProtectedFunc** are considered private members because **BaseClass** is a private base class. Again, **PrivateFunc** is private to **BaseClass**, and it is inaccessible to any derived classes.

You can declare a derived class without a base-class access specifier. In such a case, the derivation is considered private if the derived class declaration uses the **class** keyword. The derivation is considered public if the derived class declaration uses the **struct** keyword. For example, the following code:

```

class Derived : Base
...

```

is equivalent to:

```

class Derived : private Base
...

```

Similarly, the following code:

```

struct Derived : Base
...

```

is equivalent to:

```

struct Derived : public Base
...

```

Note that members declared as having private access are not accessible to functions or derived classes unless those functions or classes are declared using the **friend** declaration in the base class.

A **union** type cannot have a base class.

[!NOTE] When specifying a private base class, it is advisable to explicitly use the **private** keyword so users of the derived class understand the member access.

## Access control and static members

When you specify a base class as **private**, it affects only nonstatic members. Public static members are still accessible in the derived classes. However, accessing members of the base class using pointers, references, or objects can require a conversion, at which time access control is again applied. Consider the following example:

```
// access_control.cpp
class Base
{
public:
    int Print();           // Nonstatic member.
    static int CountOf();  // Static member.
};

// Derived1 declares Base as a private base class.
class Derived1 : private Base
{
};

// Derived2 declares Derived1 as a public base class.
class Derived2 : public Derived1
{
    int ShowCount();      // Nonstatic member.
};

// Define ShowCount function for Derived2.
int Derived2::ShowCount()
{
    // Call static member function CountOf explicitly.
    int cCount = Base::CountOf();    // OK.

    // Call static member function CountOf using pointer.
    cCount = this->CountOf(); // C2247. Conversion of
                             // Derived2 * to Base * not
                             // permitted.

    return cCount;
}
```

In the preceding code, access control prohibits conversion from a pointer to **Derived2** to a pointer to **Base**. The **this** pointer is implicitly of type **Derived2 \***. To select the **CountOf** function, **this** must be converted to type **Base \***. Such a conversion is not permitted because **Base** is a private indirect base class to **Derived2**. Conversion to a private base class type is acceptable only for pointers to immediate derived classes. Therefore, pointers of type **Derived1 \*** can be converted to type **Base \***.

Note that calling the `CountOf` function explicitly, without using a pointer, reference, or object to select it, implies no conversion. Therefore, the call is allowed.

Members and friends of a derived class, `T`, can convert a pointer to `T` to a pointer to a private direct base class of `T`.

## Access to virtual functions

The access control applied to `virtual` functions is determined by the type used to make the function call. Overriding declarations of the function do not affect the access control for a given type. For example:

```
// access_to_virtual_functions.cpp
class VFuncBase
{
public:
    virtual int GetState() { return _state; }
protected:
    int _state;
};

class VFuncDerived : public VFuncBase
{
private:
    int GetState() { return _state; }
};

int main()
{
    VFuncDerived vfd;           // Object of derived type.
    VFuncBase *pvfb = &vfd;    // Pointer to base type.
    VFuncDerived *pvfd = &vfd; // Pointer to derived type.
    int State;


    State = pvfb->GetState();    // GetState is public.
    State = pvfd->GetState();    // C2248 error expected; GetState is private;
}
```

In the preceding example, calling the virtual function `GetState` using a pointer to type `VFuncBase` calls `VFuncDerived::GetState`, and `GetState` is treated as public. However, calling `GetState` using a pointer to type `VFuncDerived` is an access-control violation because `GetState` is declared private in class `VFuncDerived`.

[!CAUTION] The virtual function `GetState` can be called using a pointer to the base class `VFuncBase`. This does not mean that the function called is the base-class version of that function.

## Access control with multiple inheritance

In multiple-inheritance lattices involving virtual base classes, a given name can be reached through more than one path. Because different access control can be applied along these different paths, the compiler chooses the path that gives the most access. See the following figure.

 Access along paths of an inheritance graph

Access along paths of an inheritance graph

In the figure, a name declared in class `VBase` is always reached through class `RightPath`. The right path is more accessible because `RightPath` declares `VBase` as a public base class, whereas `LeftPath` declares `VBase` as private.

## See also

[C++ Language Reference](#)

# friend (C++)

---

In some circumstances, it is more convenient to grant member-level access to functions that are not members of a class or to all members in a separate class. Only the class implementer can declare who its friends are. A function or class cannot declare itself as a friend of any class. In a class definition, use the **friend** keyword and the name of a non-member function or other class to grant it access to the private and protected members of your class. In a template definition, a type parameter can be declared as a friend.

## Syntax

```
class friend F
friend F;
```

## Friend declarations

If you declare a friend function that was not previously declared, that function is exported to the enclosing nonclass scope.

Functions declared in a friend declaration are treated as if they had been declared using the **extern** keyword. For more information, see [extern](#).

Although functions with global scope can be declared as friends prior to their prototypes, member functions cannot be declared as friends before the appearance of their complete class declaration. The following code shows why this fails:

```
class ForwardDeclared;    // Class name is known.
class HasFriends
{
    friend int ForwardDeclared::IsAFriend();    // C2039 error expected
};
```

The preceding example enters the class name `ForwardDeclared` into scope, but the complete declaration — specifically, the portion that declares the function `IsAFriend` — is not known. Therefore, the **friend** declaration in class `HasFriends` generates an error.

Starting in C++11, there are two forms of friend declarations for a class:

```
friend class F;
friend F;
```

The first form introduces a new class `F` if no existing class by that name was found in the innermost namespace. **C++11**: The second form does not introduce a new class; it can be used when the class has

already been declared, and it must be used when declaring a template type parameter or a typedef as a friend.

Use `class friend F` when the referenced type has not yet been declared:

```
namespace NS
{
    class M
    {
        class friend F; // Introduces F but doesn't define it
    };
}
```

```
namespace NS
{
    class M
    {
        friend F; // error C2433: 'NS::F': 'friend' not permitted on data
declarations
    };
}
```

In the following example, `friend F` refers to the `F` class that is declared outside the scope of `NS`.

```
class F {};
namespace NS
{
    class M
    {
        friend F; // OK
    };
}
```

Use `friend F` to declare a template parameter as a friend:

```
template <typename T>
class my_class
{
    friend T;
    //...
};
```

Use `friend F` to declare a typedef as friend:



```

class Foo {};
typedef Foo F;

class G
{
    friend F; // OK
    friend class F // Error C2371 -- redefinition
};

```

To declare two classes that are friends of one another, the entire second class must be specified as a friend of the first class. The reason for this restriction is that the compiler has enough information to declare individual friend functions only at the point where the second class is declared.

[!NOTE] Although the entire second class must be a friend to the first class, you can select which functions in the first class will be friends of the second class.

## friend functions

A **friend** function is a function that is not a member of a class but has access to the class's private and protected members. Friend functions are not considered class members; they are normal external functions that are given special access privileges. Friends are not in the class's scope, and they are not called using the member-selection operators ( . and -> ) unless they are members of another class. A **friend** function is declared by the class that is granting access. The **friend** declaration can be placed anywhere in the class declaration. It is not affected by the access control keywords.

The following example shows a **Point** class and a friend function, **ChangePrivate**. The **friend** function has access to the private data member of the **Point** object it receives as a parameter.

```

// friend_functions.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;
class Point
{
    friend void ChangePrivate( Point & );
public:
    Point( void ) : m_i(0) {}
    void PrintPrivate( void ){cout << m_i << endl; }

private:
    int m_i;
};

void ChangePrivate ( Point &i ) { i.m_i++; }

int main()
{
    Point sPoint;
}

```

```

    sPoint.PrintPrivate();
    ChangePrivate(sPoint);
    sPoint.PrintPrivate();
// Output: 0
           1
}

```

## Class members as friends

Class member functions can be declared as friends in other classes. Consider the following example:

```

// classes_as_friends1.cpp
// compile with: /c
class B;

class A {
public:
    int Func1( B& b );

private:
    int Func2( B& b );
};

class B {
private:
    int _b;

    // A::Func1 is a friend function to class B
    // so A::Func1 has access to all members of B
    friend int A::Func1( B& );
};

int A::Func1( B& b ) { return b._b; } // OK
int A::Func2( B& b ) { return b._b; } // C2248

```

In the preceding example, only the function `A::Func1( B& )` is granted friend access to class `B`. Therefore, access to the private member `_b` is correct in `Func1` of class `A` but not in `Func2`.

A **friend** class is a class all of whose member functions are friend functions of a class, that is, whose member functions have access to the other class's private and protected members. Suppose the **friend** declaration in class `B` had been:

```
friend class A;
```

In that case, all member functions in class `A` would have been granted friend access to class `B`. The following code is an example of a friend class:

```

// classes_as_friends2.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;
class YourClass {
friend class YourOtherClass; // Declare a friend class
public:
    YourClass() : topSecret(0){}
    void printMember() { cout << topSecret << endl; }
private:
    int topSecret;
};

class YourOtherClass {
public:
    void change( YourClass& yc, int x ){yc.topSecret = x;}
};

int main() {
    YourClass yc1;
    YourOtherClass yoc1;
    yc1.printMember();
    yoc1.change( yc1, 5 );
    yc1.printMember();
}

```

Friendship is not mutual unless explicitly specified as such. In the above example, member functions of `YourClass` cannot access the private members of `YourOtherClass`.

A managed type (in C++/CLI) cannot have any friend functions, friend classes, or friend interfaces.

Friendship is not inherited, meaning that classes derived from `YourOtherClass` cannot access `YourClass`'s private members. Friendship is not transitive, so classes that are friends of `YourOtherClass` cannot access `YourClass`'s private members.

The following figure shows four class declarations: `Base`, `Derived`, `aFriend`, and `anotherFriend`. Only class `aFriend` has direct access to the private members of `Base` (and to any members `Base` might have inherited).



Implications of friend relationship

Implications of friend relationship

## Inline friend definitions

Friend functions can be defined (given a function body) inside class declarations. These functions are inline functions, and like member inline functions they behave as though they were defined immediately after all class members have been seen but before the class scope is closed (the end of the class declaration). Friend functions that are defined inside class declarations are in the scope of the enclosing class.

See also

Keywords

# private (C++)

---

## Syntax

```
private:
    [member-list]
private base-class
```

## Remarks

When preceding a list of class members, the **private** keyword specifies that those members are accessible only from member functions and friends of the class. This applies to all members declared up to the next access specifier or the end of the class.

When preceding the name of a base class, the **private** keyword specifies that the public and protected members of the base class are private members of the derived class.

Default access of members in a class is private. Default access of members in a structure or union is public.

Default access of a base class is private for classes and public for structures. Unions cannot have base classes.

For related information, see [friend](#), [public](#), [protected](#), and the member-access table in [Controlling Access to Class Members](#).

## /clr Specific

In CLR types, the C++ access specifier keywords (**public**, **private**, and **protected**) can affect the visibility of types and methods with regard to assemblies. For more information, see [Member Access Control](#).

[!NOTE] Files compiled with [/LN](#) are not affected by this behavior. In this case, all managed classes (either public or private) will be visible.

## END /clr Specific

## Example

```
// keyword_private.cpp
class BaseClass {
public:
    // privMem accessible from member function
    int pubFunc() { return privMem; }
private:
    void privMem;
};

class DerivedClass : public BaseClass {
public:
```

```

void usePrivate( int i )
{ privMem = i; }    // C2248: privMem not accessible
                    // from derived class
};

class DerivedClass2 : private BaseClass {
public:
    // pubFunc() accessible from derived class
    int usePublic() { return pubFunc(); }
};

int main() {
    BaseClass aBase;
    DerivedClass aDerived;
    DerivedClass2 aDerived2;
    aBase.privMem = 1;    // C2248: privMem not accessible
    aDerived.privMem = 1; // C2248: privMem not accessible
                        //    in derived class
    aDerived2.pubFunc();  // C2247: pubFunc() is private in
                        //    derived class
}

```

## See also

[Controlling Access to Class Members](#)

[Keywords](#)

# protected (C++)

---

## Syntax

```
protected:  
    [member-list]  
protected base-class
```

## Remarks

The **protected** keyword specifies access to class members in the *member-list* up to the next access specifier (**public** or **private**) or the end of the class definition. Class members declared as **protected** can be used only by the following:

- Member functions of the class that originally declared these members.
- Friends of the class that originally declared these members.
- Classes derived with public or protected access from the class that originally declared these members.
- Direct privately derived classes that also have private access to protected members.

When preceding the name of a base class, the **protected** keyword specifies that the public and protected members of the base class are protected members of its derived classes.

Protected members are not as private as **private** members, which are accessible only to members of the class in which they are declared, but they are not as public as **public** members, which are accessible in any function.

Protected members that are also declared as **static** are accessible to any friend or member function of a derived class. Protected members that are not declared as **static** are accessible to friends and member functions in a derived class only through a pointer to, reference to, or object of the derived class.

For related information, see [friend](#), [public](#), [private](#), and the member-access table in [Controlling Access to Class Members](#).

## /clr Specific

In CLR types, the C++ access specifier keywords (**public**, **private**, and **protected**) can affect the visibility of types and methods with regard to assemblies. For more information, see [Member Access Control](#).

[!NOTE] Files compiled with [/LN](#) are not affected by this behavior. In this case, all managed classes (either public or private) will be visible.

## END /clr Specific

## Example

```

// keyword_protected.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;
class X {
public:
    void setProtMemb( int i ) { m_protMemb = i; }
    void Display() { cout << m_protMemb << endl; }
protected:
    int m_protMemb;
    void Protfunc() { cout << "\nAccess allowed\n"; }
} x;

class Y : public X {
public:
    void useProtfunc() { Protfunc(); }
} y;

int main() {
    // x.m_protMemb;           error, m_protMemb is protected
    x.setProtMemb( 0 );      // OK, uses public access function
    x.Display();
    y.setProtMemb( 5 );      // OK, uses public access function
    y.Display();
    // x.Protfunc();           error, Protfunc() is protected
    y.useProtfunc();         // OK, uses public access function
                             // in derived class
}

```

## See also

[Controlling Access to Class Members](#)

[Keywords](#)



# public (C++)

---

## Syntax

```
public:  
    [member-list]  
public base-class
```

## Remarks

When preceding a list of class members, the **public** keyword specifies that those members are accessible from any function. This applies to all members declared up to the next access specifier or the end of the class.

When preceding the name of a base class, the **public** keyword specifies that the public and protected members of the base class are public and protected members, respectively, of the derived class.

Default access of members in a class is private. Default access of members in a structure or union is public.

Default access of a base class is private for classes and public for structures. Unions cannot have base classes.

For more information, see [private](#), [protected](#), [friend](#), and the member-access table in [Controlling Access to Class Members](#).

## /clr Specific

In CLR types, the C++ access specifier keywords (**public**, **private**, and **protected**) can affect the visibility of types and methods with regard to assemblies. For more information, see [Member Access Control](#).

[!NOTE] Files compiled with [/LN](#) are not affected by this behavior. In this case, all managed classes (either public or private) will be visible.

## END /clr Specific

## Example

```
// keyword_public.cpp  
class BaseClass {  
public:  
    int pubFunc() { return 0; }  
};  
  
class DerivedClass : public BaseClass {};  
  
int main() {  
    BaseClass aBase;  
    DerivedClass aDerived;  
    aBase.pubFunc();    // pubFunc() is accessible
```

```
        //    from any function
    aDerived.pubFunc();    // pubFunc() is still public in
                          //    derived class
}
```

## See also

[Controlling Access to Class Members](#)

[Keywords](#)

# Brace initialization

---

It is not always necessary to define a constructor for a class, especially ones that are relatively simple. Users can initialize objects of a class or struct by using uniform initialization, as shown in the following example:

```
// no_constructor.cpp
// Compile with: cl /EHsc no_constructor.cpp
#include <time.h>

// No constructor
struct TempData
{
    int StationId;
    time_t timeSet;
    double current;
    double maxTemp;
    double minTemp;
};

// Has a constructor
struct TempData2
{
    TempData2(double minimum, double maximum, double cur, int id, time_t t) :
        stationId{id}, timeSet{t}, current{cur}, maxTemp{maximum}, minTemp{minimum}
    {}
    int stationId;
    time_t timeSet;
    double current;
    double maxTemp;
    double minTemp;
};

int main()
{
    time_t time_to_set;

    // Member initialization (in order of declaration):
    TempData td{ 45978, time(&time_to_set), 28.9, 37.0, 16.7 };

    // Default initialization = {0,0,0,0,0}
    TempData td_default{};

    // Uninitialized = if used, emits warning C4700 uninitialized local variable
    TempData td_noInit;

    // Member declaration (in order of ctor parameters)
    TempData2 td2{ 16.7, 37.0, 28.9, 45978, time(&time_to_set) };

    return 0;
}
```

Note that when a class or struct has no constructor, you provide the list elements in the order that the members are declared in the class. If the class has a constructor, provide the elements in the order of the parameters. If a type has a default constructor, either implicitly or explicitly declared, you can use default brace initialization (with empty braces). For example, the following class may be initialized by using both default and non-default brace initialization:

```
#include <string>
using namespace std;

class class_a {
public:
    class_a() {}
    class_a(string str) : m_string{ str } {}
    class_a(string str, double dbl) : m_string{ str }, m_double{ dbl } {}
    double m_double;
    string m_string;
};

int main()
{
    class_a c1{};
    class_a c1_1;

    class_a c2{ "ww" };
    class_a c2_1("xx");

    // order of parameters is the same as the constructor
    class_a c3{ "yy", 4.4 };
    class_a c3_1("zz", 5.5);
}
```

If a class has non-default constructors, the order in which class members appear in the brace initializer is the order in which the corresponding parameters appear in the constructor, not the order in which the members are declared (as with `class_a` in the previous example). Otherwise, if the type has no declared constructor, the order in which the members appear in the brace initializer is the same as the order in which they are declared; in this case, you can initialize as many of the public members as you wish, but you cannot skip any member. The following example shows the order that's used in brace initialization when there is no declared constructor:

```
class class_d {
public:
    float m_float;
    string m_string;
    wchar_t m_char;
};

int main()
```

```

{
    class_d d1{};
    class_d d1{ 4.5 };
    class_d d2{ 4.5, "string" };
    class_d d3{ 4.5, "string", 'c' };

    class_d d4{ "string", 'c' }; // compiler error
    class_d d5{ "string", 'c', 2.0 }; // compiler error
}

```

If the default constructor is explicitly declared but marked as deleted, default brace initialization cannot be used:

```

class class_f {
public:
    class_f() = delete;
    class_f(string x): m_string { x } {}
    string m_string;
};
int main()
{
    class_f cf{ "hello" };
    class_f cf1{}; // compiler error C2280: attempting to reference a deleted
function
}

```

You can use brace initialization anywhere you would typically do initialization—for example, as a function parameter or a return value, or with the **new** keyword:

```

class_d* cf = new class_d{4.5};
kr->add_d({ 4.5 });
return { 4.5 };

```

In **/std:c++17** mode, the rules for empty brace initialization are slightly more restrictive. See [Derived constructors and extended aggregate initialization](#).

## initializer\_list constructors

The [initializer\\_list Class](#) represents a list of objects of a specified type that can be used in a constructor, and in other contexts. You can construct an `initializer_list` by using brace initialization:

```

initializer_list<int> int_list{5, 6, 7};

```

[!IMPORTANT] To use this class, you must include the `<initializer_list>` header.

An `initializer_list` can be copied. In this case, the members of the new list are references to the members of the original list:

```
initializer_list<int> ilist1{ 5, 6, 7 };
initializer_list<int> ilist2( ilist1 );
if (ilist1.begin() == ilist2.begin())
    cout << "yes" << endl; // expect "yes"
```

The standard library container classes, and also `string`, `wstring`, and `regex`, have `initializer_list` constructors. The following examples show how to do brace initialization with these constructors:

```
vector<int> v1{ 9, 10, 11 };
map<int, string> m1{ {1, "a"}, {2, "b"} };
string s{ 'a', 'b', 'c' };
regex rgx{ 'x', 'y', 'z' };
```

## See also

[Classes and Structs](#)

[Constructors](#)

# Object lifetime and resource management (RAII)

---

Unlike managed languages, C++ doesn't have automatic *garbage collection*. That's an internal process that releases heap memory and other resources as a program runs. A C++ program is responsible for returning all acquired resources to the operating system. Failure to release an unused resource is called a *leak*. Leaked resources are unavailable to other programs until the process exits. Memory leaks in particular are a common cause of bugs in C-style programming.

Modern C++ avoids using heap memory as much as possible by declaring objects on the stack. When a resource is too large for the stack, then it should be *owned* by an object. As the object gets initialized, it acquires the resource it owns. The object is then responsible for releasing the resource in its destructor. The owning object itself is declared on the stack. The principle that *objects own resources* is also known as "resource acquisition is initialization," or RAII.

When a resource-owning stack object goes out of scope, its destructor is automatically invoked. In this way, garbage collection in C++ is closely related to object lifetime, and is deterministic. A resource is always released at a known point in the program, which you can control. Only deterministic destructors like those in C++ can handle memory and non-memory resources equally.

The following example shows a simple object `w`. It's declared on the stack at function scope, and is destroyed at the end of the function block. The object `w` owns no *resources* (such as heap-allocated memory). Its only member `g` is itself declared on the stack, and simply goes out of scope along with `w`. No special code is needed in the `widget` destructor.

```
class widget {
private:
    gadget g;    // lifetime automatically tied to enclosing object
public:
    void draw();
};

void functionUsingWidget () {
    widget w;    // lifetime automatically tied to enclosing scope
                // constructs w, including the w.g gadget member
    // ...
    w.draw();
    // ...
} // automatic destruction and deallocation for w and w.g
  // automatic exception safety,
  // as if "finally { w.dispose(); w.g.dispose(); }"
```

In the following example, `w` owns a memory resource and so must have code in its destructor to delete the memory.

```
class widget
{
```

```

private:
    int* data;
public:
    widget(const int size) { data = new int[size]; } // acquire
    ~widget() { delete[] data; } // release
    void do_something() {}
};

void functionUsingWidget() {
    widget w(1000000); // lifetime automatically tied to enclosing scope
                        // constructs w, including the w.data member

    w.do_something();

} // automatic destruction and deallocation for w and w.data

```

Since C++11, there's a better way to write the previous example: by using a smart pointer from the standard library. The smart pointer handles the allocation and deletion of the memory it owns. Using a smart pointer eliminates the need for an explicit destructor in the `widget` class.

```

#include <memory>
class widget
{
private:
    std::unique_ptr<int> data;
public:
    widget(const int size) { data = std::make_unique<int>(size); }
    void do_something() {}
};

void functionUsingWidget() {
    widget w(1000000); // lifetime automatically tied to enclosing scope
                        // constructs w, including the w.data gadget member

    // ...
    w.do_something();
    // ...
} // automatic destruction and deallocation for w and w.data

```

By using smart pointers for memory allocation, you may eliminate the potential for memory leaks. This model works for other resources, such as file handles or sockets. You can manage your own resources in a similar way in your classes. For more information, see [Smart pointers](#).

The design of C++ ensures objects are destroyed when they go out of scope. That is, they get destroyed as blocks are exited, in reverse order of construction. When an object is destroyed, its bases and members are destroyed in a particular order. Objects declared outside of any block, at global scope, can lead to problems. It may be difficult to debug, if the constructor of a global object throws an exception.

See also



Welcome back to C++  
C++ Language Reference  
C++ Standard Library

# Pimpl For Compile-Time Encapsulation (Modern C++)

---

The *pimpl idiom* is a modern C++ technique to hide implementation, to minimize coupling, and to separate interfaces. Pimpl is short for "pointer to implementation." You may already be familiar with the concept but know it by other names like Cheshire Cat or Compiler Firewall idiom.

## Why use pimpl?

Here's how the pimpl idiom can improve the software development lifecycle:

- Minimization of compilation dependencies.
- Separation of interface and implementation.
- Portability.

## Pimpl header

```
// my_class.h
class my_class {
    // ... all public and protected stuff goes here ...
private:
    class impl; unique_ptr<impl> pimpl; // opaque type here
};
```

The pimpl idiom avoids rebuild cascades and brittle object layouts. It's well suited for (transitively) popular types.

## Pimpl implementation

Define the `impl` class in the `.cpp` file.

```
// my_class.cpp
class my_class::impl { // defined privately here
    // ... all private data and functions: all of these
    //      can now change without recompiling callers ...
};
my_class::my_class(): pimpl( new impl )
{
    // ... set impl values ...
}
```

## Best practices

Consider whether to add support for non-throwing swap specialization.

## See also

[Welcome back to C++](#)

[C++ Language Reference](#)

[C++ Standard Library](#)

# Portability at ABI boundaries

---

Use sufficiently portable types and conventions at binary interface boundaries. A "portable type" is a C built-in type or a struct that contains only C built-in types. Class types can only be used when caller and callee agree on layout, calling convention, etc. That's only possible when both are compiled with the same compiler and compiler settings.

## How to flatten a class for C portability

When callers may be compiled with another compiler/language, then "flatten" to an **extern "C"** API with a specific calling convention:

```
// class widget {  
//   widget();  
//   ~widget();  
//   double method( int, gadget& );  
// };  
extern "C" {          // functions using explicit "this"  
    struct widget;    // opaque type (forward declaration only)  
    widget* STDCALL widget_create();      // constructor creates new "this"  
    void STDCALL widget_destroy(widget*); // destructor consumes "this"  
    double STDCALL widget_method(widget*, int, gadget*); // method uses "this"  
}
```

## See also

[Welcome back to C++](#)

[C++ Language Reference](#)

[C++ Standard Library](#)

# Constructors (C++)

---

To customize how class members are initialized, or to invoke functions when an object of your class is created, define a *constructor*. A constructor has the same name as the class and no return value. You can define as many overloaded constructors as needed to customize initialization in various ways. Typically, constructors have public accessibility so that code outside the class definition or inheritance hierarchy can create objects of the class. But you can also declare a constructor as **protected** or **private**.

Constructors can optionally take a member init list. This is a more efficient way to initialize class members than assigning values in the constructor body. The following example shows a class `Box` with three overloaded constructors. The last two use member init lists:

```
class Box {
public:
    // Default constructor
    Box() {}

    // Initialize a Box with equal dimensions (i.e. a cube)
    explicit Box(int i) : m_width(i), m_length(i), m_height(i) // member init list
    {}

    // Initialize a Box with custom dimensions
    Box(int width, int length, int height)
        : m_width(width), m_length(length), m_height(height)
    {}

    int Volume() { return m_width * m_length * m_height; }

private:
    // Will have value of 0 when default constructor is called.
    // If we didn't zero-init here, default constructor would
    // leave them uninitialized with garbage values.
    int m_width{ 0 };
    int m_length{ 0 };
    int m_height{ 0 };
};
```

When you declare an instance of a class, the compiler chooses which constructor to invoke based on the rules of overload resolution:

```
int main()
{
    Box b; // Calls Box()

    // Using uniform initialization (preferred):
    Box b2 {5}; // Calls Box(int)
    Box b3 {5, 8, 12}; // Calls Box(int, int, int)
```

```
// Using function-style notation:
Box b4(2, 4, 6); // Calls Box(int, int, int)
}
```

- Constructors may be declared as **inline**, **explicit**, **friend** or **constexpr**.
- A constructor can initialize an object that has been declared as **const**, **volatile** or **const volatile**. The object becomes **const** after the constructor completes.
- To define a constructor in an implementation file, give it a qualified name as with any other member function: `Box::Box(){...}`.

## Member initializer lists

A constructor can optionally have a member initializer list, which initializes class members prior to execution of the constructor body. (Note that a member initializer list is not the same thing as an *initializer list* of type `std::initializer_list<T>`.)

Using a member initializer list is preferred over assigning values in the body of the constructor because it directly initializes the member. In the following example shows the member initializer list consists of all the **identifier(argument)** expressions after the colon:

```
Box(int width, int length, int height)
    : m_width(width), m_length(length), m_height(height)
{ }
```

The identifier must refer to a class member; it is initialized with the value of the argument. The argument can be one of the constructor parameters, a function call or a `std::initializer_list<T>`.

**const** members and members of reference type must be initialized in the member initializer list.

Calls to parameterized base class constructors should be made in the initializer list to ensure the base class is fully initialized prior to execution of the derived constructor.

## Default constructors

*Default constructors* typically have no parameters, but they can have parameters with default values.

```
class Box {
public:
    Box() { /*perform any required default initialization steps*/ }

    // All params have default values
    Box (int w = 1, int l = 1, int h = 1): m_width(w), m_height(h), m_length(l){}
    ...
}
```

Default constructors are one of the [special member functions](#). If no constructors are declared in a class, the compiler provides an implicit **inline** default constructor.

```
#include <iostream>
using namespace std;

class Box {
public:
    int Volume() {return m_width * m_height * m_length;}
private:
    int m_width { 0 };
    int m_height { 0 };
    int m_length { 0 };
};

int main() {
    Box box1; // Invoke compiler-generated constructor
    cout << "box1.Volume: " << box1.Volume() << endl; // Outputs 0
}
```

If you rely on an implicit default constructor, be sure to initialize members in the class definition, as shown in the previous example. Without those initializers, the members would be uninitialized and the `Volume()` call would produce a garbage value. In general, it is good practice to initialize members in this way even when not relying on an implicit default constructor.

You can prevent the compiler from generating an implicit default constructor by defining it as [deleted](#):

```
// Default constructor
Box() = delete;
```

A compiler-generated default constructor will be defined as deleted if any class members are not default-constructible. For example, all members of class type, and their class-type members, must have a default constructor and destructors that are accessible. All data members of reference type, as well as **const** members must have a default member initializer.

When you call a compiler-generated default constructor and try to use parentheses, a warning is issued:

```
class myclass{};
int main(){
    myclass mc();    // warning C4930: prototyped function not called (was a variable
                    // definition intended?)
}
```

This is an example of the Most Vexing Parse problem. Because the example expression can be interpreted either as the declaration of a function or as the invocation of a default constructor, and because C++ parsers

favor declarations over other things, the expression is treated as a function declaration. For more information, see [Most Vexing Parse](#).

If any non-default constructors are declared, the compiler does not provide a default constructor:

```
class Box {
public:
    Box(int width, int length, int height)
        : m_width(width), m_length(length), m_height(height){}
private:
    int m_width;
    int m_length;
    int m_height;
};

int main(){

    Box box1(1, 2, 3);
    Box box2{ 2, 3, 4 };
    Box box3; // C2512: no appropriate default constructor available
}
```

If a class has no default constructor, an array of objects of that class cannot be constructed by using square-bracket syntax alone. For example, given the previous code block, an array of Boxes cannot be declared like this:

```
Box boxes[3]; // C2512: no appropriate default constructor available
```

However, you can use a set of initializer lists to initialize an array of Box objects:

```
Box boxes[3]{ { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };
```

For more information, see [Initializers](#).

## Copy constructors

A *copy constructor* initializes an object by copying the member values from an object of the same type. If your class members are all simple types such as scalar values, the compiler-generated copy constructor is sufficient and you do not need to define your own. If your class requires more complex initialization, then you need to implement a custom copy constructor. For example, if a class member is a pointer then you need to define a copy constructor to allocate new memory and copy the values from the other's pointed-to object. The compiler-generated copy constructor simply copies the pointer, so that the new pointer still points to the other's memory location.

A copy constructor may have one of these signatures:



```
Box(Box& other); // Avoid if possible--allows modification of other.
Box(const Box& other);
Box(volatile Box& other);
Box(volatile const Box& other);

// Additional parameters OK if they have default values
Box(Box& other, int i = 42, string label = "Box");
```

When you define a copy constructor, you should also define a copy assignment operator (=). For more information, see [Assignment](#) and [Copy constructors and copy assignment operators](#).

You can prevent your object from being copied by defining the copy constructor as deleted:

```
Box (const Box& other) = delete;
```

Attempting to copy the object produces error *C2280: attempting to reference a deleted function*.

## Move constructors

A *move constructor* is a special member function that moves ownership of an existing object's data to a new variable without copying the original data. It takes an rvalue reference as its first parameter, and any additional parameters must have default values. Move constructors can significantly increase your program's efficiency when passing around large objects.

```
Box(Box&& other);
```

The compiler chooses a move constructor in certain situations where the object is being initialized by another object of the same type that is about to be destroyed and no longer needs its resources. The following example shows one case when a move constructor is selected by overload resolution. In the constructor that calls `get_Box()`, the returned value is an *xvalue* (eXpiring value). It is not assigned to any variable and is therefore about to go out of scope. To provide motivation for this example, let's give `Box` a large vector of strings that represent its contents. Rather than copying the vector and its strings, the move constructor "steals" it from the expiring value "box" so that the vector now belongs to the new object. The call to `std::move` is all that's needed because both `vector` and `string` classes implement their own move constructors.

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
using namespace std;

class Box {
public:
```

```

Box() { std::cout << "default" << std::endl; }
Box(int width, int height, int length)
    : m_width(width), m_height(height), m_length(length)
{
    std::cout << "int,int,int" << std::endl;
}
Box(Box& other)
    : m_width(other.m_width), m_height(other.m_height),
m_length(other.m_length)
{
    std::cout << "copy" << std::endl;
}
Box(Box&& other) : m_width(other.m_width), m_height(other.m_height),
m_length(other.m_length)
{
    m_contents = std::move(other.m_contents);
    std::cout << "move" << std::endl;
}
int Volume() { return m_width * m_height * m_length; }
void Add_Item(string item) { m_contents.push_back(item); }
void Get_Contents()
{
    for (const auto& item : m_contents)
    {
        cout << item << " ";
    }
}
private:
    int m_width{ 0 };
    int m_height{ 0 };
    int m_length{ 0 };
    vector<string> m_contents;
};

Box get_Box()
{
    Box b(5, 10, 18); // "int,int,int"
    b.Add_Item("Toupee");
    b.Add_Item("Megaphone");
    b.Add_Item("Suit");

    return b;
}

int main()
{
    Box b; // "default"
    Box b1(b); // "copy"
    Box b2(get_Box()); // "move"
    cout << "b2 contents: ";
    b2.Get_Contents(); // Prove that we have all the values

    char ch;
    cin >> ch; // keep window open

```

```
    return 0;
}
```

If a class does not define a move constructor, the compiler generates an implicit one if there is no user-declared copy constructor, copy assignment operator, move assignment operator, or destructor. If no explicit or implicit move constructor is defined, operations that would otherwise use a move constructor use the copy constructor instead. If a class declares a move constructor or move assignment operator, the implicitly declared copy constructor is defined as deleted.

An implicitly declared move constructor is defined as deleted if any members that are class types lack a destructor or the compiler cannot determine which constructor to use for the move operation.

For more information about how to write a non-trivial move constructor, see [Move Constructors and Move Assignment Operators \(C++\)](#).

## Explicitly defaulted and deleted constructors

You can explicitly *default* copy constructors, default constructors, move constructors, copy assignment operators, move assignment operators, and destructors. You can explicitly *delete* all of the special member functions.

```
class Box
{
public:
    Box2() = delete;
    Box2(const Box2& other) = default;
    Box2& operator=(const Box2& other) = default;
    Box2(Box2&& other) = default;
    Box2& operator=(Box2&& other) = default;
    //...
};
```

For more information, see [Explicitly Defaulted and Deleted Functions](#).

## constexpr constructors

A constructor may be declared as [constexpr](#) if

- it is either declared as defaulted or else it satisfies all the conditions for [constexpr functions](#) in general;
- the class has no virtual base classes;
- each of the parameters is a [literal type](#);
- the body is not a function try-block;
- all non-static data members and base class sub-objects are initialized;
- if the class is (a) a union having variant members, or (b) has anonymous unions, only one of the union members is initialized;
- every non-static data member of class type, and all base-class sub-objects have a constexpr constructor

## Initializer list constructors

If a constructor takes a `std::initializer_list<T>` as its parameter, and any other parameters have default arguments, that constructor will be selected in overload resolution when the class is instantiated through direct initialization. You can use the `initializer_list` to initialize any member that can accept it. For example, assume the `Box` class (shown previously) has a `std::vector<string>` member `m_contents`. You can provide a constructor like this:

```
Box(initializer_list<string> list, int w = 0, int h = 0, int l = 0)
    : m_contents(list), m_width(w), m_height(h), m_length(l)
{}

```

And then create `Box` objects like this:

```
Box b{ "apples", "oranges", "pears" }; // or ...
Box b2(initializer_list<string> { "bread", "cheese", "wine" }, 2, 4, 6);

```

## Explicit constructors

If a class has a constructor with a single parameter, or if all parameters except one have a default value, the parameter type can be implicitly converted to the class type. For example, if the `Box` class has a constructor like this:

```
Box(int size): m_width(size), m_length(size), m_height(size){}

```

It is possible to initialize a `Box` like this:

```
Box b = 42;

```

Or pass an `int` to a function that takes a `Box`:

```
class ShippingOrder
{
public:
    ShippingOrder(Box b, double postage) : m_box(b), m_postage(postage){}

private:
    Box m_box;
    double m_postage;
}
//elsewhere...
ShippingOrder so(42, 10.8);

```

Such conversions can be useful in some cases, but more often they can lead to subtle but serious errors in your code. As a general rule, you should use the **explicit** keyword on a constructor (and user-defined operators) to prevent this kind of implicit type conversion:

```
explicit Box(int size): m_width(size), m_length(size), m_height(size){}
```

When the constructor is explicit, this line causes a compiler error: `ShippingOrder so(42, 10.8);`. For more information, see [User-Defined Type Conversions](#).

## Order of construction

A constructor performs its work in this order:

1. It calls base class and member constructors in the order of declaration.
2. If the class is derived from virtual base classes, it initializes the object's virtual base pointers.
3. If the class has or inherits virtual functions, it initializes the object's virtual function pointers. Virtual function pointers point to the class's virtual function table to enable correct binding of virtual function calls to code.
4. It executes any code in its function body.

The following example shows the order in which base class and member constructors are called in the constructor for a derived class. First, the base constructor is called, then the base-class members are initialized in the order in which they appear in the class declaration, and then the derived constructor is called.

```
#include <iostream>

using namespace std;

class Contained1 {
public:
    Contained1() { cout << "Contained1 ctor\n"; }
};

class Contained2 {
public:
    Contained2() { cout << "Contained2 ctor\n"; }
};

class Contained3 {
public:
    Contained3() { cout << "Contained3 ctor\n"; }
};

class BaseContainer {
public:
    BaseContainer() { cout << "BaseContainer ctor\n"; }
private:
```

```

    Contained1 c1;
    Contained2 c2;
};

class DerivedContainer : public BaseContainer {
public:
    DerivedContainer() : BaseContainer() { cout << "DerivedContainer ctor\n"; }
private:
    Contained3 c3;
};

int main() {
    DerivedContainer dc;
}

```

Here's the output:

```

Contained1 ctor
Contained2 ctor
BaseContainer ctor
Contained3 ctor
DerivedContainer ctor

```

A derived class constructor always calls a base class constructor, so that it can rely on completely constructed base classes before any extra work is done. The base class constructors are called in order of derivation—for example, if **ClassA** is derived from **ClassB**, which is derived from **ClassC**, the **ClassC** constructor is called first, then the **ClassB** constructor, then the **ClassA** constructor.

If a base class does not have a default constructor, you must supply the base class constructor parameters in the derived class constructor:

```

class Box {
public:
    Box(int width, int length, int height){
        m_width = width;
        m_length = length;
        m_height = height;
    }

private:
    int m_width;
    int m_length;
    int m_height;
};

class StorageBox : public Box {
public:
    StorageBox(int width, int length, int height, const string label&) :
    Box(width, length, height){

```

```

        m_label = label;
    }
private:
    string m_label;
};

int main(){

    const string aLabel = "aLabel";
    StorageBox sb(1, 2, 3, aLabel);
}

```

If a constructor throws an exception, the order of destruction is the reverse of the order of construction:

1. The code in the body of the constructor function is unwound.
2. Base class and member objects are destroyed, in the reverse order of declaration.
3. If the constructor is non-delegating, all fully-constructed base class objects and members are destroyed. However, because the object itself is not fully constructed, the destructor is not run.

## Derived constructors and extended aggregate initialization

If the constructor of a base class is non-public, but accessible to a derived class, then under **/std:c++17** mode in Visual Studio 2017 and later you can't use empty braces to initialize an object of the derived type.

The following example shows C++14 conformant behavior:

```

struct Derived;

struct Base {
    friend struct Derived;
private:
    Base() {}
};

struct Derived : Base {};

Derived d1; // OK. No aggregate init involved.
Derived d2 {}; // OK in C++14: Calls Derived::Derived()
               // which can call Base ctor.

```

In C++17, **Derived** is now considered an aggregate type. It means that the initialization of **Base** via the private default constructor happens directly, as part of the extended aggregate initialization rule. Previously, the **Base** private constructor was called via the **Derived** constructor, and it succeeded because of the friend declaration.

The following example shows C++17 behavior in Visual Studio 2017 and later in **/std:c++17** mode:

```

struct Derived;

struct Base {
    friend struct Derived;
private:
    Base() {}
};

struct Derived : Base {
    Derived() {} // add user-defined constructor
                // to call with {} initialization
};

Derived d1; // OK. No aggregate init involved.

Derived d2 {}; // error C2248: 'Base::Base': cannot access
               // private member declared in class 'Base'

```

## Constructors for classes that have multiple inheritance

If a class is derived from multiple base classes, the base class constructors are invoked in the order in which they are listed in the declaration of the derived class:

```

#include <iostream>
using namespace std;

class BaseClass1 {
public:
    BaseClass1() { cout << "BaseClass1 ctor\n"; }
};
class BaseClass2 {
public:
    BaseClass2() { cout << "BaseClass2 ctor\n"; }
};
class BaseClass3 {
public:
    BaseClass3() { cout << "BaseClass3 ctor\n"; }
};
class DerivedClass : public BaseClass1,
                    public BaseClass2,
                    public BaseClass3
                    {
public:
    DerivedClass() { cout << "DerivedClass ctor\n"; }
};

int main() {
    DerivedClass dc;
}

```



You should expect the following output:

```
BaseClass1 ctor
BaseClass2 ctor
BaseClass3 ctor
DerivedClass ctor
```

## Delegating constructors

A *delegating constructor* calls a different constructor in the same class to do some of the work of initialization. This is useful when you have multiple constructors that all have to perform similar work. You can write the main logic in one constructor and invoke it from others. In the following trivial example, `Box(int)` delegates its work to `Box(int,int,int)`:

```
class Box {
public:
    // Default constructor
    Box() {}

    // Initialize a Box with equal dimensions (i.e. a cube)
    Box(int i) : Box(i, i, i); // delegating constructor
    {}

    // Initialize a Box with custom dimensions
    Box(int width, int length, int height)
        : m_width(width), m_length(length), m_height(height)
    {}
    //... rest of class as before
};
```

The object created by the constructors is fully initialized as soon as any constructor is finished. For more information, see [Delegating Constructors](#).

## Inheriting constructors (C++11)

A derived class can inherit the constructors from a direct base class by using a **using** declaration as shown in the following example:

```
#include <iostream>
using namespace std;

class Base
{
public:
    Base() { cout << "Base()" << endl; }
    Base(const Base& other) { cout << "Base(Base&)" << endl; }
    explicit Base(int i) : num(i) { cout << "Base(int)" << endl; }
```

```

    explicit Base(char c) : letter(c) { cout << "Base(char)" << endl; }

private:
    int num;
    char letter;
};

class Derived : Base
{
public:
    // Inherit all constructors from Base
    using Base::Base;

private:
    // Can't initialize newMember from Base constructors.
    int newMember{ 0 };
};

int main()
{
    cout << "Derived d1(5) calls: ";
    Derived d1(5);
    cout << "Derived d1('c') calls: ";
    Derived d2('c');
    cout << "Derived d3 = d2 calls: " ;
    Derived d3 = d2;
    cout << "Derived d4 calls: ";
    Derived d4;
}

/* Output:
Derived d1(5) calls: Base(int)
Derived d1('c') calls: Base(char)
Derived d3 = d2 calls: Base(Base&)
Derived d4 calls: Base()*/

```

=vs-2017"">

**Visual Studio 2017 and later:** The **using** statement in **/std:c++17** mode brings into scope all constructors from the base class except those that have an identical signature to constructors in the derived class. In general, it is best to use inheriting constructors when the derived class declares no new data members or constructors. See also [Improvements in Visual Studio 2017 version 15.7](#).

A class template can inherit all the constructors from a type argument if that type specifies a base class:

```

template< typename T >
class Derived : T {
    using T::T;    // declare the constructors from T
    // ...
};

```

A deriving class cannot inherit from multiple base classes if those base classes have constructors that have an identical signature.

## Constructors and composite classes

Classes that contain class-type members are known as *composite classes*. When a class-type member of a composite class is created, the constructor is called before the class's own constructor. When a contained class lacks a default constructor, you must use an initialization list in the constructor of the composite class. In the earlier `StorageBox` example, if you change the type of the `m_label` member variable to a new `Label` class, you must call both the base class constructor and initialize the `m_label` variable in the `StorageBox` constructor:

```
class Label {
public:
    Label(const string& name, const string& address) { m_name = name; m_address =
address; }
    string m_name;
    string m_address;
};

class StorageBox : public Box {
public:
    StorageBox(int width, int length, int height, Label label)
        : Box(width, length, height), m_label(label){}
private:
    Label m_label;
};

int main(){
    // passing a named Label
    Label label1{ "some_name", "some_address" };
    StorageBox sb1(1, 2, 3, label1);

    // passing a temporary label
    StorageBox sb2(3, 4, 5, Label{ "another name", "another address" });

    // passing a temporary label as an initializer list
    StorageBox sb3(1, 2, 3, {"myname", "myaddress"});
}
```

### In this section

- [Copy constructors and copy assignment operators](#)
- [Move constructors and move assignment operators](#)
- [Delegating constructors](#)

### See also

[Classes and structs](#)

# Copy Constructors and Copy Assignment Operators (C++)

---

[!NOTE] Starting in C++11, two kinds of assignment are supported in the language: *copy assignment* and *move assignment*. In this article "assignment" means copy assignment unless explicitly stated otherwise. For information about move assignment, see [Move Constructors and Move Assignment Operators \(C++\)](#).

Both the assignment operation and the initialization operation cause objects to be copied.

- **Assignment:** When one object's value is assigned to another object, the first object is copied to the second object. Therefore,

```
Point a, b;  
...  
a = b;
```

causes the value of **b** to be copied to **a**.

- **Initialization:** Initialization occurs when a new object is declared, when arguments are passed to functions by value, or when values are returned from functions by value.

You can define the semantics of "copy" for objects of class type. For example, consider this code:

```
TextFile a, b;  
a.Open( "FILE1.DAT" );  
b.Open( "FILE2.DAT" );  
b = a;
```

The preceding code could mean "copy the contents of FILE1.DAT to FILE2.DAT" or it could mean "ignore FILE2.DAT and make **b** a second handle to FILE1.DAT." You must attach appropriate copying semantics to each class, as follows.

- By using the assignment operator **operator=** together with a reference to the class type as the return type and the parameter that is passed by **const** reference—for example `ClassName& operator=(const ClassName& x);`.
- By using the copy constructor.

If you do not declare a copy constructor, the compiler generates a member-wise copy constructor for you. If you do not declare a copy assignment operator, the compiler generates a member-wise copy assignment operator for you. Declaring a copy constructor does not suppress the compiler-generated copy assignment operator, nor vice versa. If you implement either one, we recommend that you also implement the other one so that the meaning of the code is clear.

The copy constructor takes an argument of type *class-name*&, where *class-name* is the name of the class for which the constructor is defined. For example:

```
// spec1_copying_class_objects.cpp
class Window
{
public:
    Window( const Window& ); // Declare copy constructor.
    // ...
};

int main()
{
}
```

[!NOTE] Make the type of the copy constructor's argument **const class-name&** whenever possible. This prevents the copy constructor from accidentally changing the object from which it is copying. It also enables copying from **const** objects.

## Compiler generated copy constructors

Compiler-generated copy constructors, like user-defined copy constructors, have a single argument of type "reference to *class-name*." An exception is when all base classes and member classes have copy constructors declared as taking a single argument of type **const class-name&**. In such a case, the compiler-generated copy constructor's argument is also **const**.

When the argument type to the copy constructor is not **const**, initialization by copying a **const** object generates an error. The reverse is not true: If the argument is **const**, you can initialize by copying an object that is not **const**.

Compiler-generated assignment operators follow the same pattern with regard to **const**. They take a single argument of type *class-name*& unless the assignment operators in all base and member classes take arguments of type **const class-name&**. In this case, the class's generated assignment operator takes a **const** argument.

[!NOTE] When virtual base classes are initialized by copy constructors, compiler-generated or user-defined, they are initialized only once: at the point when they are constructed.

The implications are similar to those of the copy constructor. When the argument type is not **const**, assignment from a **const** object generates an error. The reverse is not true: If a **const** value is assigned to a value that is not **const**, the assignment succeeds.

For more information about overloaded assignment operators, see [Assignment](#).

# Move Constructors and Move Assignment Operators (C++)

---

This topic describes how to write a *move constructor* and a move assignment operator for a C++ class. A move constructor enables the resources owned by an rvalue object to be moved into an lvalue without copying. For more information about move semantics, see [Rvalue Reference Declarator: &&](#).

This topic builds upon the following C++ class, `MemoryBlock`, which manages a memory buffer.

```
// MemoryBlock.h
#pragma once
#include <iostream>
#include <algorithm>

class MemoryBlock
{
public:

    // Simple constructor that initializes the resource.
    explicit MemoryBlock(size_t length)
        : _length(length)
        , _data(new int[length])
    {
        std::cout << "In MemoryBlock(size_t). length = "
                    << _length << "." << std::endl;
    }

    // Destructor.
    ~MemoryBlock()
    {
        std::cout << "In ~MemoryBlock(). length = "
                    << _length << ".";

        if (_data != nullptr)
        {
            std::cout << " Deleting resource.";
            // Delete the resource.
            delete[] _data;
        }

        std::cout << std::endl;
    }

    // Copy constructor.
    MemoryBlock(const MemoryBlock& other)
        : _length(other._length)
        , _data(new int[other._length])
    {
        std::cout << "In MemoryBlock(const MemoryBlock&). length = "
```

```

        << other._length << ". Copying resource." << std::endl;

    std::copy(other._data, other._data + _length, _data);
}

// Copy assignment operator.
MemoryBlock& operator=(const MemoryBlock& other)
{
    std::cout << "In operator=(const MemoryBlock&). length = "
        << other._length << ". Copying resource." << std::endl;

    if (this != &other)
    {
        // Free the existing resource.
        delete[] _data;

        _length = other._length;
        _data = new int[_length];
        std::copy(other._data, other._data + _length, _data);
    }
    return *this;
}

// Retrieves the length of the data resource.
size_t Length() const
{
    return _length;
}

private:
    size_t _length; // The length of the resource.
    int* _data; // The resource.
};

```

The following procedures describe how to write a move constructor and a move assignment operator for the example C++ class.

To create a move constructor for a C++ class

1. Define an empty constructor method that takes an rvalue reference to the class type as its parameter, as demonstrated in the following example:

```

MemoryBlock(MemoryBlock&& other)
    : _data(nullptr)
    , _length(0)
{
}

```

2. In the move constructor, assign the class data members from the source object to the object that is being constructed:

```
_data = other._data;  
_length = other._length;
```

3. Assign the data members of the source object to default values. This prevents the destructor from freeing resources (such as memory) multiple times:

```
other._data = nullptr;  
other._length = 0;
```

To create a move assignment operator for a C++ class

1. Define an empty assignment operator that takes an rvalue reference to the class type as its parameter and returns a reference to the class type, as demonstrated in the following example:

```
MemoryBlock& operator=(MemoryBlock&& other)  
{  
}
```

2. In the move assignment operator, add a conditional statement that performs no operation if you try to assign the object to itself.

```
if (this != &other)  
{  
}
```

3. In the conditional statement, free any resources (such as memory) from the object that is being assigned to.

The following example frees the `_data` member from the object that is being assigned to:

```
// Free the existing resource.  
delete[] _data;
```

Follow steps 2 and 3 in the first procedure to transfer the data members from the source object to the object that is being constructed:

```
// Copy the data pointer and its length from the  
// source object.  
_data = other._data;  
_length = other._length;
```



```
// Release the data pointer from the source object so that
// the destructor does not free the memory multiple times.
other._data = nullptr;
other._length = 0;
```

4. Return a reference to the current object, as shown in the following example:

```
return *this;
```

## Example

The following example shows the complete move constructor and move assignment operator for the `MemoryBlock` class:

```
// Move constructor.
MemoryBlock(MemoryBlock&& other)
: _data(nullptr)
, _length(0)
{
    std::cout << "In MemoryBlock(MemoryBlock&&). length = "
                << other._length << ". Moving resource." << std::endl;

    // Copy the data pointer and its length from the
    // source object.
    _data = other._data;
    _length = other._length;

    // Release the data pointer from the source object so that
    // the destructor does not free the memory multiple times.
    other._data = nullptr;
    other._length = 0;
}

// Move assignment operator.
MemoryBlock& operator=(MemoryBlock&& other)
{
    std::cout << "In operator=(MemoryBlock&&). length = "
                << other._length << "." << std::endl;

    if (this != &other)
    {
        // Free the existing resource.
        delete[] _data;

        // Copy the data pointer and its length from the
        // source object.
        _data = other._data;
        _length = other._length;
    }
}
```

```

        // Release the data pointer from the source object so that
        // the destructor does not free the memory multiple times.
        other._data = nullptr;
        other._length = 0;
    }
    return *this;
}

```

## Example

The following example shows how move semantics can improve the performance of your applications. The example adds two elements to a vector object and then inserts a new element between the two existing elements. The `vector` class uses move semantics to perform the insertion operation efficiently by moving the elements of the vector instead of copying them.

```

// rvalue-references-move-semantics.cpp
// compile with: /EHsc
#include "MemoryBlock.h"
#include <vector>

using namespace std;

int main()
{
    // Create a vector object and add a few elements to it.
    vector<MemoryBlock> v;
    v.push_back(MemoryBlock(25));
    v.push_back(MemoryBlock(75));

    // Insert a new element into the second position of the vector.
    v.insert(v.begin() + 1, MemoryBlock(50));
}

```

This example produces the following output:

```

In MemoryBlock(size_t). length = 25.
In MemoryBlock(MemoryBlock&&). length = 25. Moving resource.
In ~MemoryBlock(). length = 0.
In MemoryBlock(size_t). length = 75.
In MemoryBlock(MemoryBlock&&). length = 25. Moving resource.
In ~MemoryBlock(). length = 0.
In MemoryBlock(MemoryBlock&&). length = 75. Moving resource.
In ~MemoryBlock(). length = 0.
In MemoryBlock(size_t). length = 50.
In MemoryBlock(MemoryBlock&&). length = 50. Moving resource.
In MemoryBlock(MemoryBlock&&). length = 50. Moving resource.
In operator=(MemoryBlock&&). length = 75.
In operator=(MemoryBlock&&). length = 50.
In ~MemoryBlock(). length = 0.

```

```
In ~MemoryBlock(). length = 0.  
In ~MemoryBlock(). length = 25. Deleting resource.  
In ~MemoryBlock(). length = 50. Deleting resource.  
In ~MemoryBlock(). length = 75. Deleting resource.
```

Before Visual Studio 2010, this example produced the following output:

```
In MemoryBlock(size_t). length = 25.  
In MemoryBlock(const MemoryBlock&). length = 25. Copying resource.  
In ~MemoryBlock(). length = 25. Deleting resource.  
In MemoryBlock(size_t). length = 75.  
In MemoryBlock(const MemoryBlock&). length = 25. Copying resource.  
In ~MemoryBlock(). length = 25. Deleting resource.  
In MemoryBlock(const MemoryBlock&). length = 75. Copying resource.  
In ~MemoryBlock(). length = 75. Deleting resource.  
In MemoryBlock(size_t). length = 50.  
In MemoryBlock(const MemoryBlock&). length = 50. Copying resource.  
In MemoryBlock(const MemoryBlock&). length = 50. Copying resource.  
In operator=(const MemoryBlock&). length = 75. Copying resource.  
In operator=(const MemoryBlock&). length = 50. Copying resource.  
In ~MemoryBlock(). length = 50. Deleting resource.  
In ~MemoryBlock(). length = 50. Deleting resource.  
In ~MemoryBlock(). length = 25. Deleting resource.  
In ~MemoryBlock(). length = 50. Deleting resource.  
In ~MemoryBlock(). length = 75. Deleting resource.
```

The version of this example that uses move semantics is more efficient than the version that does not use move semantics because it performs fewer copy, memory allocation, and memory deallocation operations.

## Robust Programming

To prevent resource leaks, always free resources (such as memory, file handles, and sockets) in the move assignment operator.

To prevent the unrecoverable destruction of resources, properly handle self-assignment in the move assignment operator.

If you provide both a move constructor and a move assignment operator for your class, you can eliminate redundant code by writing the move constructor to call the move assignment operator. The following example shows a revised version of the move constructor that calls the move assignment operator:

```
// Move constructor.  
MemoryBlock(MemoryBlock&& other)  
    : _data(nullptr)  
    , _length(0)  
{  
    *this = std::move(other);  
}
```

---

The [std::move](#) function preserves the rvalue property of the *other* parameter.

## See also

[Rvalue Reference Declarator: &&](#)  
[std::move](#)

# Delegating constructors

---

Many classes have multiple constructors that do similar things—for example, validate parameters:

```
class class_c {
public:
    int max;
    int min;
    int middle;

    class_c() {}
    class_c(int my_max) {
        max = my_max > 0 ? my_max : 10;
    }
    class_c(int my_max, int my_min) {
        max = my_max > 0 ? my_max : 10;
        min = my_min > 0 && my_min < max ? my_min : 1;
    }
    class_c(int my_max, int my_min, int my_middle) {
        max = my_max > 0 ? my_max : 10;
        min = my_min > 0 && my_min < max ? my_min : 1;
        middle = my_middle < max && my_middle > min ? my_middle : 5;
    }
};
```

You could reduce the repetitive code by adding a function that does all of the validation, but the code for `class_c` would be easier to understand and maintain if one constructor could delegate some of the work to another one. To add delegating constructors, use the `constructor (. . .) : constructor (. . .)` syntax:

```
class class_c {
public:
    int max;
    int min;
    int middle;

    class_c(int my_max) {
        max = my_max > 0 ? my_max : 10;
    }
    class_c(int my_max, int my_min) : class_c(my_max) {
        min = my_min > 0 && my_min < max ? my_min : 1;
    }
    class_c(int my_max, int my_min, int my_middle) : class_c (my_max, my_min){
        middle = my_middle < max && my_middle > min ? my_middle : 5;
    }
};

int main() {
```

```

class_c c1{ 1, 3, 2 };
}

```

As you step through the previous example, notice that the constructor `class_c(int, int, int)` first calls the constructor `class_c(int, int)`, which in turn calls `class_c(int)`. Each of the constructors performs only the work that is not performed by the other constructors.

The first constructor that's called initializes the object so that all of its members are initialized at that point. You can't do member initialization in a constructor that delegates to another constructor, as shown here:

```

class class_a {
public:
    class_a() {}
    // member initialization here, no delegate
    class_a(string str) : m_string{ str } {}

    //can't do member initialization here
    // error C3511: a call to a delegating constructor shall be the only member-
initializer
    class_a(string str, double dbl) : class_a(str) , m_double{ dbl } {}

    // only member assignment
    class_a(string str, double dbl) : class_a(str) { m_double = dbl; }
    double m_double{ 1.0 };
    string m_string;
};

```

The next example shows the use of non-static data-member initializers. Notice that if a constructor also initializes a given data member, the member initializer is overridden:

```

class class_a {
public:
    class_a() {}
    class_a(string str) : m_string{ str } {}
    class_a(string str, double dbl) : class_a(str) { m_double = dbl; }
    double m_double{ 1.0 };
    string m_string{ m_double < 10.0 ? "alpha" : "beta" };
};

int main() {
    class_a a{ "hello", 2.0 }; //expect a.m_double == 2.0, a.m_string == "hello"
    int y = 4;
}

```

The constructor delegation syntax doesn't prevent the accidental creation of constructor recursion—Constructor1 calls Constructor2 which calls Constructor1—and no errors are thrown until there is a stack overflow. It's your responsibility to avoid cycles.

```
class class_f{
public:
    int max;
    int min;

    // don't do this
    class_f() : class_f(6, 3){ }
    class_f(int my_max, int my_min) : class_f() { }
};
```

# Destructors (C++)

---

A destructor is a member function that is invoked automatically when the object goes out of scope or is explicitly destroyed by a call to **delete**. A destructor has the same name as the class, preceded by a tilde (~). For example, the destructor for class `String` is declared: `~String()`.

If you do not define a destructor, the compiler will provide a default one; for many classes this is sufficient. You only need to define a custom destructor when the class stores handles to system resources that need to be released, or pointers that own the memory they point to.

Consider the following declaration of a `String` class:

```
// spec1_destructors.cpp
#include <string>

class String {
public:
    String( char *ch ); // Declare constructor
    ~String();          // and destructor.
private:
    char    *_text;
    size_t  sizeOfText;
};

// Define the constructor.
String::String( char *ch ) {
    sizeOfText = strlen( ch ) + 1;

    // Dynamically allocate the correct amount of memory.
    _text = new char[ sizeOfText ];

    // If the allocation succeeds, copy the initialization string.
    if( _text )
        strcpy_s( _text, sizeOfText, ch );
}

// Define the destructor.
String::~~String() {
    // Deallocate the memory that was previously reserved
    // for this string.
    delete[] _text;
}

int main() {
    String str("The piper in the glen...");
}
```

In the preceding example, the destructor `String::~~String` uses the **delete** operator to deallocate the space dynamically allocated for text storage.



# Declaring destructors

Destructors are functions with the same name as the class but preceded by a tilde (~)

Several rules govern the declaration of destructors. Destructors:

- Do not accept arguments.
- Do not return a value (or **void**).
- Cannot be declared as **const**, **volatile**, or **static**. However, they can be invoked for the destruction of objects declared as **const**, **volatile**, or **static**.
- Can be declared as **virtual**. Using virtual destructors, you can destroy objects without knowing their type — the correct destructor for the object is invoked using the virtual function mechanism. Note that destructors can also be declared as pure virtual functions for abstract classes.

## Using destructors

Destructors are called when one of the following events occurs:

- A local (automatic) object with block scope goes out of scope.
- An object allocated using the **new** operator is explicitly deallocated using **delete**.
- The lifetime of a temporary object ends.
- A program ends and global or static objects exist.
- The destructor is explicitly called using the destructor function's fully qualified name.

Destructors can freely call class member functions and access class member data.

There are two restrictions on the use of destructors:

- You cannot take its address.
- Derived classes do not inherit the destructor of their base class.

## Order of destruction

When an object goes out of scope or is deleted, the sequence of events in its complete destruction is as follows:

1. The class's destructor is called, and the body of the destructor function is executed.
2. Destructors for nonstatic member objects are called in the reverse order in which they appear in the class declaration. The optional member initialization list used in construction of these members does not affect the order of construction or destruction.
3. Destructors for non-virtual base classes are called in the reverse order of declaration.
4. Destructors for virtual base classes are called in the reverse order of declaration.

```
// order_of_destruction.cpp
#include <cstdio>

struct A1      { virtual ~A1() { printf("A1 dtor\n"); } };
struct A2 : A1 { virtual ~A2() { printf("A2 dtor\n"); } };
struct A3 : A2 { virtual ~A3() { printf("A3 dtor\n"); } };

struct B1      { ~B1() { printf("B1 dtor\n"); } };
struct B2 : B1 { ~B2() { printf("B2 dtor\n"); } };
struct B3 : B2 { ~B3() { printf("B3 dtor\n"); } };

int main() {
    A1 * a = new A3;
    delete a;
    printf("\n");

    B1 * b = new B3;
    delete b;
    printf("\n");

    B3 * b2 = new B3;
    delete b2;
}
```

Output: A3 dtor

A2 dtor

A1 dtor

B1 dtor

B3 dtor

B2 dtor

B1 dtor

## Virtual base classes

Destructors for virtual base classes are called in the reverse order of their appearance in a directed acyclic graph (depth-first, left-to-right, postorder traversal). the following figure depicts an inheritance graph.



Inheritance graph that shows virtual base classes

Inheritance graph that shows virtual base classes

The following lists the class heads for the classes shown in the figure.

```
class A
class B
class C : virtual public A, virtual public B
class D : virtual public A, virtual public B
class E : public C, public D, virtual public B
```

To determine the order of destruction of the virtual base classes of an object of type **E**, the compiler builds a list by applying the following algorithm:

1. Traverse the graph left, starting at the deepest point in the graph (in this case, **E**).
2. Perform leftward traversals until all nodes have been visited. Note the name of the current node.
3. Revisit the previous node (down and to the right) to find out whether the node being remembered is a virtual base class.
4. If the remembered node is a virtual base class, scan the list to see whether it has already been entered. If it is not a virtual base class, ignore it.
5. If the remembered node is not yet in the list, add it to the bottom of the list.
6. Traverse the graph up and along the next path to the right.
7. Go to step 2.
8. When the last upward path is exhausted, note the name of the current node.
9. Go to step 3.
10. Continue this process until the bottom node is again the current node.

Therefore, for class **E**, the order of destruction is:

1. The non-virtual base class **E**.
2. The non-virtual base class **D**.
3. The non-virtual base class **C**.
4. The virtual base class **B**.
5. The virtual base class **A**.

This process produces an ordered list of unique entries. No class name appears twice. Once the list is constructed, it is walked in reverse order, and the destructor for each of the classes in the list from the last to the first is called.

The order of construction or destruction is primarily important when constructors or destructors in one class rely on the other component being created first or persisting longer — for example, if the destructor for **A** (in the figure shown above) relied on **B** still being present when its code executed, or vice versa.

Such interdependencies between classes in an inheritance graph are inherently dangerous because classes derived later can alter which is the leftmost path, thereby changing the order of construction and destruction.

## Non-virtual base classes

The destructors for non-virtual base classes are called in the reverse order in which the base class names are declared. Consider the following class declaration:

```
class MultInherit : public Base1, public Base2
...
```

In the preceding example, the destructor for `Base2` is called before the destructor for `Base1`.

## Explicit destructor calls

Calling a destructor explicitly is seldom necessary. However, it can be useful to perform cleanup of objects placed at absolute addresses. These objects are commonly allocated using a user-defined **new** operator that takes a placement argument. The **delete** operator cannot deallocate this memory because it is not allocated from the free store (for more information, see [The new and delete Operators](#)). A call to the destructor, however, can perform appropriate cleanup. To explicitly call the destructor for an object, `s`, of class `String`, use one of the following statements:

```
s.String::~~String();    // non-virtual call
ps->String::~~String();  // non-virtual call

s.~String();             // Virtual call
ps->~String();            // Virtual call
```

The notation for explicit calls to destructors, shown in the preceding, can be used regardless of whether the type defines a destructor. This allows you to make such explicit calls without knowing if a destructor is defined for the type. An explicit call to a destructor where none is defined has no effect.

## Robust programming

A class needs a destructor if it acquires a resource, and to manage the resource safely it probably has to implement a copy constructor and a copy assignment.

If these special functions are not defined by the user, they are implicitly defined by the compiler. The implicitly generated constructors and assignment operators perform shallow, memberwise copy, which is almost certainly wrong if an object is managing a resource.

In the next example, the implicitly generated copy constructor will make the pointers `str1.text` and `str2.text` refer to the same memory, and when we return from `copy_strings()`, that memory will be deleted twice, which is undefined behavior:

```
void copy_strings()
{
    String str1("I have a sense of impending disaster...");
    String str2 = str1; // str1.text and str2.text now refer to the same object
} // delete[] _text; deallocates the same memory twice
// undefined behavior
```

Explicitly defining a destructor, copy constructor, or copy assignment operator prevents implicit definition of the move constructor and the move assignment operator. In this case, failing to provide move operations is usually, if copying is expensive, a missed optimization opportunity.

## See also

[Copy Constructors and Copy Assignment Operators](#)

[Move Constructors and Move Assignment Operators](#)

# Overview of Member Functions

---

Member functions are either static or nonstatic. The behavior of static member functions differs from other member functions because static member functions have no implicit **this** argument. Nonstatic member functions have a **this** pointer. Member functions, whether static or nonstatic, can be defined either in or outside the class declaration.

If a member function is defined inside a class declaration, it is treated as an inline function, and there is no need to qualify the function name with its class name. Although functions defined inside class declarations are already treated as inline functions, you can use the **inline** keyword to document code.

An example of declaring a function within a class declaration follows:

```
// overview_of_member_functions1.cpp
class Account
{
public:
    // Declare the member function Deposit within the declaration
    // of class Account.
    double Deposit( double HowMuch )
    {
        balance += HowMuch;
        return balance;
    }
private:
    double balance;
};

int main()
{
}
```

If a member function's definition is outside the class declaration, it is treated as an inline function only if it is explicitly declared as **inline**. In addition, the function name in the definition must be qualified with its class name using the scope-resolution operator (`::`).

The following example is identical to the previous declaration of class `Account`, except that the `Deposit` function is defined outside the class declaration:

```
// overview_of_member_functions2.cpp
class Account
{
public:
    // Declare the member function Deposit but do not define it.
    double Deposit( double HowMuch );
private:
    double balance;
}
```

```
};

inline double Account::Deposit( double HowMuch )
{
    balance += HowMuch;
    return balance;
}

int main()
{
}
```

[!NOTE] Although member functions can be defined either inside a class declaration or separately, no member functions can be added to a class after the class is defined.

Classes containing member functions can have many declarations, but the member functions themselves must have only one definition in a program. Multiple definitions cause an error message at link time. If a class contains inline function definitions, the function definitions must be identical to observe this "one definition" rule.

# virtual Specifier

---

The [virtual](#) keyword can be applied only to nonstatic class member functions. It signifies that binding of calls to the function is deferred until run time. For more information, see [Virtual Functions](#).



# override Specifier

---

You can use the **override** keyword to designate member functions that override a virtual function in a base class.

## Syntax

```
function-declaration override;
```

## Remarks

**override** is context-sensitive and has special meaning only when it's used after a member function declaration; otherwise, it's not a reserved keyword.

## Example

Use **override** to help prevent inadvertent inheritance behavior in your code. The following example shows where, without using **override**, the member function behavior of the derived class may not have been intended. The compiler doesn't emit any errors for this code.

```
class BaseClass
{
    virtual void funcA();
    virtual void funcB() const;
    virtual void funcC(int = 0);
    void funcD();
};

class DerivedClass: public BaseClass
{
    virtual void funcA(); // ok, works as intended

    virtual void funcB(); // DerivedClass::funcB() is non-const, so it does not
                          // override BaseClass::funcB() const and it is a new
member function

    virtual void funcC(double = 0.0); // DerivedClass::funcC(double) has a
different                                     // parameter type than
BaseClass::funcC(int), so                      // DerivedClass::funcC(double) is a new
member function
};
```

When you use **override**, the compiler generates errors instead of silently creating new member functions.

```

class BaseClass
{
    virtual void funcA();
    virtual void funcB() const;
    virtual void funcC(int = 0);
    void funcD();
};

class DerivedClass: public BaseClass
{
    virtual void funcA() override; // ok

    virtual void funcB() override; // compiler error: DerivedClass::funcB() does
not                                     // override BaseClass::funcB() const

    virtual void funcC( double = 0.0 ) override; // compiler error:
                                                // DerivedClass::funcC(double)
does not                                     // override BaseClass::funcC(int)

    void funcD() override; // compiler error: DerivedClass::funcD() does not
                           // override the non-virtual BaseClass::funcD()
};

```

To specify that functions cannot be overridden and that classes cannot be inherited, use the [final](#) keyword.

## See also

[final Specifier](#)

[Keywords](#)

# final Specifier

---

You can use the **final** keyword to designate virtual functions that cannot be overridden in a derived class. You can also use it to designate classes that cannot be inherited.

## Syntax

```
function-declaration final;  
class class-name final base-classes
```

## Remarks

**final** is context-sensitive and has special meaning only when it's used after a function declaration or class name; otherwise, it's not a reserved keyword.

When **final** is used in class declarations, **base-classes** is an optional part of the declaration.

## Example

The following example uses the **final** keyword to specify that a virtual function cannot be overridden.

```
class BaseClass  
{  
    virtual void func() final;  
};  
  
class DerivedClass: public BaseClass  
{  
    virtual void func(); // compiler error: attempting to  
                        // override a final function  
};
```

For information about how to specify that member functions can be overridden, see [override Specifier](#).

The next example uses the **final** keyword to specify that a class cannot be inherited.

```
class BaseClass final  
{  
};  
  
class DerivedClass: public BaseClass // compiler error: BaseClass is  
                                    // marked as non-inheritable  
{  
};
```

## See also

[Keywords](#)

[override Specifier](#)

# Inheritance (C++)

---

This section explains how to use derived classes to produce extensible programs.

## Overview

New classes can be derived from existing classes using a mechanism called "inheritance" (see the information beginning in [Single Inheritance](#)). Classes that are used for derivation are called "base classes" of a particular derived class. A derived class is declared using the following syntax:

```
class Derived : [virtual] [access-specifier] Base
{
    // member list
};
class Derived : [virtual] [access-specifier] Base1,
    [virtual] [access-specifier] Base2, . . .
{
    // member list
};
```

After the tag (name) for the class, a colon appears followed by a list of base specifications. The base classes so named must have been declared previously. The base specifications may contain an access specifier, which is one of the keywords **public**, **protected** or **private**. These access specifiers appear before the base class name and apply only to that base class. These specifiers control the derived class's permission to use to members of the base class. See [Member-Access Control](#) for information on access to base class members. If the access specifier is omitted, the access to that base is considered **private**. The base specifications may contain the keyword **virtual** to indicate virtual inheritance. This keyword may appear before or after the access specifier, if any. If virtual inheritance is used, the base class is referred to as a virtual base class.

Multiple base classes can be specified, separated by commas. If a single base class is specified, the inheritance model is [Single inheritance](#). If more than one base class is specified, the inheritance model is called [Multiple inheritance](#).

The following topics are included:

- [Single inheritance](#)
- [Multiple base classes](#)
- [Virtual functions](#)
- [Explicit overrides](#)
- [Abstract classes](#)
- [Summary of scope rules](#)

The `__super` and `__interface` keywords are documented in this section.

## See also

[C++ Language Reference](#)

# Virtual Functions

---

A virtual function is a member function that you expect to be redefined in derived classes. When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class's version of the function.

Virtual functions ensure that the correct function is called for an object, regardless of the expression used to make the function call.

Suppose a base class contains a function declared as `virtual` and a derived class defines the same function. The function from the derived class is invoked for objects of the derived class, even if it is called using a pointer or reference to the base class. The following example shows a base class that provides an implementation of the `PrintBalance` function and two derived classes

```
// deriv_VirtualFunctions.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

class Account {
public:
    Account( double d ) { _balance = d; }
    virtual ~Account() {}
    virtual double GetBalance() { return _balance; }
    virtual void PrintBalance() { cerr << "Error. Balance not available for base
type." << endl; }
private:
    double _balance;
};

class CheckingAccount : public Account {
public:
    CheckingAccount(double d) : Account(d) {}
    void PrintBalance() { cout << "Checking account balance: " << GetBalance() <<
endl; }
};

class SavingsAccount : public Account {
public:
    SavingsAccount(double d) : Account(d) {}
    void PrintBalance() { cout << "Savings account balance: " << GetBalance(); }
};

int main() {
    // Create objects of type CheckingAccount and SavingsAccount.
    CheckingAccount checking( 100.00 );
    SavingsAccount savings( 1000.00 );

    // Call PrintBalance using a pointer to Account.
    Account *pAccount = &checking;
```

```

    pAccount->PrintBalance();

    // Call PrintBalance using a pointer to Account.
    pAccount = &savings;
    pAccount->PrintBalance();
}

```

In the preceding code, the calls to `PrintBalance` are identical, except for the object `pAccount` points to. Because `PrintBalance` is virtual, the version of the function defined for each object is called. The `PrintBalance` function in the derived classes `CheckingAccount` and `SavingsAccount` "override" the function in the base class `Account`.

If a class is declared that does not provide an overriding implementation of the `PrintBalance` function, the default implementation from the base class `Account` is used.

Functions in derived classes override virtual functions in base classes only if their type is the same. A function in a derived class cannot differ from a virtual function in a base class in its return type only; the argument list must differ as well.

When calling a function using pointers or references, the following rules apply:

- A call to a virtual function is resolved according to the underlying type of object for which it is called.
- A call to a nonvirtual function is resolved according to the type of the pointer or reference.

The following example shows how virtual and nonvirtual functions behave when called through pointers:

```

// deriv_VirtualFunctions2.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

class Base {
public:
    virtual void NameOf();    // Virtual function.
    void InvokingClass();    // Nonvirtual function.
};

// Implement the two functions.
void Base::NameOf() {
    cout << "Base::NameOf\n";
}

void Base::InvokingClass() {
    cout << "Invoked by Base\n";
}

class Derived : public Base {
public:
    void NameOf();    // Virtual function.
    void InvokingClass();    // Nonvirtual function.
}

```



```

};

// Implement the two functions.
void Derived::NameOf() {
    cout << "Derived::NameOf\n";
}

void Derived::InvokingClass() {
    cout << "Invoked by Derived\n";
}

int main() {
    // Declare an object of type Derived.
    Derived aDerived;

    // Declare two pointers, one of type Derived * and the other
    // of type Base *, and initialize them to point to aDerived.
    Derived *pDerived = &aDerived;
    Base     *pBase    = &aDerived;

    // Call the functions.
    pBase->NameOf();           // Call virtual function.
    pBase->InvokingClass();    // Call nonvirtual function.
    pDerived->NameOf();        // Call virtual function.
    pDerived->InvokingClass(); // Call nonvirtual function.
}

```

```

Derived::NameOf
Invoked by Base
Derived::NameOf
Invoked by Derived

```

Note that regardless of whether the `NameOf` function is invoked through a pointer to `Base` or a pointer to `Derived`, it calls the function for `Derived`. It calls the function for `Derived` because `NameOf` is a virtual function, and both `pBase` and `pDerived` point to an object of type `Derived`.

Because virtual functions are called only for objects of class types, you cannot declare global or static functions as **virtual**.

The **virtual** keyword can be used when declaring overriding functions in a derived class, but it is unnecessary; overrides of virtual functions are always virtual.

Virtual functions in a base class must be defined unless they are declared using the *pure-specifier*. (For more information about pure virtual functions, see [Abstract Classes](#).)

The virtual function-call mechanism can be suppressed by explicitly qualifying the function name using the scope-resolution operator (`::`). Consider the earlier example involving the `Account` class. To call `PrintBalance` in the base class, use code such as the following:


```
CheckingAccount *pChecking = new CheckingAccount( 100.00 );  
  
pChecking->Account::PrintBalance(); // Explicit qualification.  
  
Account *pAccount = pChecking; // Call Account::PrintBalance  
  
pAccount->Account::PrintBalance(); // Explicit qualification.
```

Both calls to `PrintBalance` in the preceding example suppress the virtual function-call mechanism.

# Single Inheritance

---

In "single inheritance," a common form of inheritance, classes have only one base class. Consider the relationship illustrated in the following figure.

 Basic single-inheritance graph  
Simple Single-Inheritance Graph

Note the progression from general to specific in the figure. Another common attribute found in the design of most class hierarchies is that the derived class has a "kind of" relationship with the base class. In the figure, a **Book** is a kind of a **PrintedDocument**, and a **PaperbackBook** is a kind of a **book**.

One other item of note in the figure: **Book** is both a derived class (from **PrintedDocument**) and a base class (**PaperbackBook** is derived from **Book**). A skeletal declaration of such a class hierarchy is shown in the following example:

```
// deriv_SingleInheritance.cpp
// compile with: /LD
class PrintedDocument {};

// Book is derived from PrintedDocument.
class Book : public PrintedDocument {};


// PaperbackBook is derived from Book.
class PaperbackBook : public Book {};
```

**PrintedDocument** is considered a "direct base" class to **Book**; it is an "indirect base" class to **PaperbackBook**. The difference is that a direct base class appears in the base list of a class declaration and an indirect base does not.

The base class from which each class is derived is declared before the declaration of the derived class. It is not sufficient to provide a forward-referencing declaration for a base class; it must be a complete declaration.

In the preceding example, the access specifier **public** is used. The meaning of public, protected, and private inheritance is described in [Member-Access Control](#).

A class can serve as the base class for many specific classes, as illustrated in the following figure.

 Directed acyclic graph  
Sample of Directed Acyclic Graph

In the diagram shown above, called a "directed acyclic graph" (or "DAG"), some of the classes are base classes for more than one derived class. However, the reverse is not true: there is only one direct base class for any given derived class. The graph in the figure depicts a "single inheritance" structure.

[!NOTE] Directed acyclic graphs are not unique to single inheritance. They are also used to depict multiple-inheritance graphs.

In inheritance, the derived class contains the members of the base class plus any new members you add. As a result, a derived class can refer to members of the base class (unless those members are redefined in the derived class). The scope-resolution operator (`::`) can be used to refer to members of direct or indirect base classes when those members have been redefined in the derived class. Consider this example:

```
// deriv_SingleInheritance2.cpp
// compile with: /EHsc /c
#include <iostream>
using namespace std;
class Document {
public:
    char *Name;    // Document name.
    void PrintNameOf();    // Print name.
};

// Implementation of PrintNameOf function from class Document.
void Document::PrintNameOf() {
    cout << Name << endl;
}

class Book : public Document {
public:
    Book( char *name, long pagecount );
private:
    long PageCount;
};

// Constructor from class Book.
Book::Book( char *name, long pagecount ) {
    Name = new char[ strlen( name ) + 1 ];
    strcpy_s( Name, strlen(Name), name );
    PageCount = pagecount;
};
```

Note that the constructor for `Book`, (`Book::Book`), has access to the data member, `Name`. In a program, an object of type `Book` can be created and used as follows:

```
// Create a new object of type Book. This invokes the
// constructor Book::Book.
Book LibraryBook( "Programming Windows, 2nd Ed", 944 );

...

// Use PrintNameOf function inherited from class Document.
LibraryBook.PrintNameOf();
```

As the preceding example demonstrates, class-member and inherited data and functions are used identically. If the implementation for class `Book` calls for a reimplementaion of the `PrintNameOf` function, the function

that belongs to the `Document` class can be called only by using the scope-resolution (`::`) operator:

```
// deriv_SingleInheritance3.cpp
// compile with: /EHsc /LD
#include <iostream>
using namespace std;

class Document {
public:
    char *Name;           // Document name.
    void PrintNameOf() {} // Print name.
};

class Book : public Document {
    Book( char *name, long pagecount );
    void PrintNameOf();
    long PageCount;
};

void Book::PrintNameOf() {
    cout << "Name of book: ";
    Document::PrintNameOf();
}
```

Pointers and references to derived classes can be implicitly converted to pointers and references to their base classes if there is an accessible, unambiguous base class. The following code demonstrates this concept using pointers (the same principle applies to references):

```
// deriv_SingleInheritance4.cpp
// compile with: /W3
struct Document {
    char *Name;
    void PrintNameOf() {}
};

class PaperbackBook : public Document {};

int main() {
    Document * DocLib[10]; // Library of ten documents.
    for (int i = 0 ; i < 5 ; i++)
        DocLib[i] = new Document;
    for (int i = 5 ; i < 10 ; i++)
        DocLib[i] = new PaperbackBook;
}
```

In the preceding example, different types are created. However, because these types are all derived from the `Document` class, there is an implicit conversion to `Document *`. As a result, `DocLib` is a "heterogeneous list" (a list in which not all objects are of the same type) containing different kinds of objects.

Because the `Document` class has a `PrintNameOf` function, it can print the name of each book in the library, although it may omit some of the information specific to the type of document (page count for `Book`, number of bytes for `HelpFile`, and so on).

[!NOTE] Forcing the base class to implement a function such as `PrintNameOf` is often not the best design. [Virtual Functions](#) offers other design alternatives.

# Base Classes

---

The inheritance process creates a new derived class that is made up of the members of the base class(es) plus any new members added by the derived class. In a multiple-inheritance, it is possible to construct an inheritance graph where the same base class is part of more than one of the derived classes. The following figure shows such a graph.



Multiple instances of a base class

Multiple instances of a single base class

In the figure, pictorial representations of the components of `CollectibleString` and `CollectibleSortable` are shown. However, the base class, `Collectible`, is in `CollectibleSortableString` through the `CollectibleString` path and the `CollectibleSortable` path. To eliminate this redundancy, such classes can be declared as virtual base classes when they are inherited.

# Multiple Base Classes

---

A class can be derived from more than one base class. In a multiple-inheritance model (where classes are derived from more than one base class), the base classes are specified using the *base-list* grammar element. For example, the class declaration for `CollectionOfBook`, derived from `Collection` and `Book`, can be specified:

```
// deriv_MultipleBaseClasses.cpp
// compile with: /LD
class Collection {
};
class Book {};
class CollectionOfBook : public Book, public Collection {
    // New members
};
```

The order in which base classes are specified is not significant except in certain cases where constructors and destructors are invoked. In these cases, the order in which base classes are specified affects the following:

- The order in which initialization by constructor takes place. If your code relies on the `Book` portion of `CollectionOfBook` to be initialized before the `Collection` part, the order of specification is significant. Initialization takes place in the order the classes are specified in the *base-list*.
- The order in which destructors are invoked to clean up. Again, if a particular "part" of the class must be present when the other part is being destroyed, the order is significant. Destructors are called in the reverse order of the classes specified in the *base-list*.

[!NOTE] The order of specification of base classes can affect the memory layout of the class. Do not make any programming decisions based on the order of base members in memory.

When specifying the *base-list*, you cannot specify the same class name more than once. However, it is possible for a class to be an indirect base to a derived class more than once.

## Virtual base classes

Because a class can be an indirect base class to a derived class more than once, C++ provides a way to optimize the way such base classes work. Virtual base classes offer a way to save space and avoid ambiguities in class hierarchies that use multiple inheritance.


Each nonvirtual object contains a copy of the data members defined in the base class. This duplication wastes space and requires you to specify which copy of the base class members you want whenever you access them.

When a base class is specified as a virtual base, it can act as an indirect base more than once without duplication of its data members. A single copy of its data members is shared by all the base classes that use it as a virtual base.

When declaring a virtual base class, the **virtual** keyword appears in the base lists of the derived classes.




Consider the class hierarchy in the following figure, which illustrates a simulated lunch line.

Graph of simulated lunch line

Simulated lunch-line graph

In the figure, `Queue` is the base class for both `CashierQueue` and `LunchQueue`. However, when both classes are combined to form `LunchCashierQueue`, the following problem arises: the new class contains two subobjects of type `Queue`, one from `CashierQueue` and the other from `LunchQueue`. The following figure shows the conceptual memory layout (the actual memory layout might be optimized).


Simulated lunch-line object

Simulated lunch-line object

Note that there are two `Queue` subobjects in the `LunchCashierQueue` object. The following code declares `Queue` to be a virtual base class:


```
// deriv_VirtualBaseClasses.cpp
// compile with: /LD
class Queue {};
class CashierQueue : virtual public Queue {};
class LunchQueue : virtual public Queue {};
class LunchCashierQueue : public LunchQueue, public CashierQueue {};
```

The **virtual** keyword ensures that only one copy of the subobject `Queue` is included (see the following figure).

Simulated lunch-line object, virtual base classes


Simulated lunch-line object with virtual base classes

A class can have both a virtual component and a nonvirtual component of a given type. This happens in the conditions illustrated in the following figure.

Virtual and non-virtual components of a class

Virtual and non-virtual components of the same class

In the figure, `CashierQueue` and `LunchQueue` use `Queue` as a virtual base class. However, `TakeoutQueue` specifies `Queue` as a base class, not a virtual base class. Therefore, `LunchTakeoutCashierQueue` has two subobjects of type `Queue`: one from the inheritance path that includes `LunchCashierQueue` and one from the path that includes `TakeoutQueue`. This is illustrated in the following figure.

Virtual & non-virtual inheritance in object layout

Object layout with virtual and non-virtual inheritance

[!NOTE] Virtual inheritance provides significant size benefits when compared with nonvirtual inheritance. However, it can introduce extra processing overhead.

If a derived class overrides a virtual function that it inherits from a virtual base class, and if a constructor or a destructor for the derived base class calls that function using a pointer to the virtual base class, the compiler may introduce additional hidden "vtordisp" fields into the classes with virtual bases. The `/vd0` compiler option suppresses the addition of the hidden vtordisp constructor/destructor displacement member. The `/vd1`

compiler option, the default, enables them where they are necessary. Turn off `vtordisps` only if you are sure that all class constructors and destructors call virtual functions virtually.

The `/vd` compiler option affects an entire compilation module. Use the `vtordisp` pragma to suppress and then reenable `vtordisp` fields on a class-by-class basis:

```
#pragma vtordisp( off )
class GetReal : virtual public { ... };
#pragma vtordisp( on )
```

## Name ambiguities

Multiple inheritance introduces the possibility for names to be inherited along more than one path. The class-member names along these paths are not necessarily unique. These name conflicts are called "ambiguities."

Any expression that refers to a class member must make an unambiguous reference. The following example shows how ambiguities develop:

```
// deriv_NameAmbiguities.cpp
// compile with: /LD
// Declare two base classes, A and B.
class A {
public:
    unsigned a;
    unsigned b();
};

class B {
public:
    unsigned a(); // Note that class A also has a member "a"
    int b();      // and a member "b".
    char c;
};

// Define class C as derived from A and B.
class C : public A, public B {};
```

Given the preceding class declarations, code such as the following is ambiguous because it is unclear whether `b` refers to the `b` in `A` or in `B`:

```
C *pc = new C;

pc->b();
```

Consider the preceding example. Because the name `a` is a member of both class `A` and class `B`, the compiler cannot discern which `a` designates the function to be called. Access to a member is ambiguous if it can refer

to more than one function, object, type, or enumerator.

The compiler detects ambiguities by performing tests in this order:

1. If access to the name is ambiguous (as just described), an error message is generated.
2. If overloaded functions are unambiguous, they are resolved.
3. If access to the name violates member-access permission, an error message is generated. (For more information, see [Member-Access Control](#).)

When an expression produces an ambiguity through inheritance, you can manually resolve it by qualifying the name in question with its class name. To make the preceding example compile properly with no ambiguities, use code such as:

```
C *pc = new C;

pc->B::a();
```

[!NOTE] When `C` is declared, it has the potential to cause errors when `B` is referenced in the scope of `C`. No error is issued, however, until an unqualified reference to `B` is actually made in `C`'s scope.

## Dominance

It is possible for more than one name (function, object, or enumerator) to be reached through an inheritance graph. Such cases are considered ambiguous with nonvirtual base classes. They are also ambiguous with virtual base classes, unless one of the names "dominates" the others.

A name dominates another name if it is defined in both classes and one class is derived from the other. The dominant name is the name in the derived class; this name is used when an ambiguity would otherwise have arisen, as shown in the following example:

```
// deriv_Dominance.cpp
// compile with: /LD
class A {
public:
    int a;
};

class B : public virtual A {
public:
    int a();
};

class C : public virtual A {};

class D : public B, public C {
public:
    D() { a(); } // Not ambiguous. B::a() dominates A::a.
};
```

## Ambiguous conversions

Explicit and implicit conversions from pointers or references to class types can cause ambiguities. The next figure, Ambiguous Conversion of Pointers to Base Classes, shows the following:

- The declaration of an object of type **D**.
- The effect of applying the address-of operator (**&**) to that object. Note that the address-of operator always supplies the base address of the object.
- The effect of explicitly converting the pointer obtained using the address-of operator to the base-class type **A**. Note that coercing the address of the object to type **A\*** does not always provide the compiler with enough information as to which subobject of type **A** to select; in this case, two subobjects exist.



Ambiguous conversion of pointers to base classes

Ambiguous conversion of pointers to base classes

The conversion to type **A\*** (pointer to **A**) is ambiguous because there is no way to discern which subobject of type **A** is the correct one. Note that you can avoid the ambiguity by explicitly specifying which subobject you mean to use, as follows:

```
(A *) (B *)&d      // Use B subobject.  
(A *) (C *)&d      // Use C subobject.
```

## Ambiguities and virtual base classes

If virtual base classes are used, functions, objects, types, and enumerators can be reached through multiple-inheritance paths. Because there is only one instance of the base class, there is no ambiguity when accessing these names.

The following figure shows how objects are composed using virtual and nonvirtual inheritance.



Virtual derivation and non-virtual derivation

Virtual vs. non-virtual derivation

In the figure, accessing any member of class **A** through nonvirtual base classes causes an ambiguity; the compiler has no information that explains whether to use the subobject associated with **B** or the subobject associated with **C**. However, when **A** is specified as a virtual base class, there is no question which subobject is being accessed.

## See also

[Inheritance](#)

# Explicit Overrides (C++)

---

## Microsoft Specific

If the same virtual function is declared in two or more [interfaces](#) and if a class is derived from these interfaces, you can explicitly override each virtual function.

For information on explicit overrides in managed code using C++/CLI, see [Explicit Overrides](#).

## END Microsoft Specific

## Example

The following code example illustrates how to use explicit overrides:

```
// deriv_ExplicitOverrides.cpp
// compile with: /GR
extern "C" int printf_s(const char *, ...);

__interface IMyInt1 {
    void mf1();
    void mf1(int);
    void mf2();
    void mf2(int);
};

__interface IMyInt2 {
    void mf1();
    void mf1(int);
    void mf2();
    void mf2(int);
};

class CMyClass : public IMyInt1, public IMyInt2 {
public:
    void IMyInt1::mf1() {
        printf_s("In CMyClass::IMyInt1::mf1()\n");
    }

    void IMyInt1::mf1(int) {
        printf_s("In CMyClass::IMyInt1::mf1(int)\n");
    }

    void IMyInt1::mf2();
    void IMyInt1::mf2(int);

    void IMyInt2::mf1() {
        printf_s("In CMyClass::IMyInt2::mf1()\n");
    }
}
```

```

void IMyInt2::mf1(int) {
    printf_s("In CMyClass::IMyInt2::mf1(int)\n");
}

void IMyInt2::mf2();
void IMyInt2::mf2(int);
};

void CMyClass::IMyInt1::mf2() {
    printf_s("In CMyClass::IMyInt1::mf2()\n");
}

void CMyClass::IMyInt1::mf2(int) {
    printf_s("In CMyClass::IMyInt1::mf2(int)\n");
}

void CMyClass::IMyInt2::mf2() {
    printf_s("In CMyClass::IMyInt2::mf2()\n");
}

void CMyClass::IMyInt2::mf2(int) {
    printf_s("In CMyClass::IMyInt2::mf2(int)\n");
}

int main() {
    IMyInt1 *pIMyInt1 = new CMyClass();
    IMyInt2 *pIMyInt2 = dynamic_cast<IMyInt2 *>(pIMyInt1);

    pIMyInt1->mf1();
    pIMyInt1->mf1(1);
    pIMyInt1->mf2();
    pIMyInt1->mf2(2);
    pIMyInt2->mf1();
    pIMyInt2->mf1(3);
    pIMyInt2->mf2();
    pIMyInt2->mf2(4);

    // Cast to a CMyClass pointer so that the destructor gets called
    CMyClass *p = dynamic_cast<CMyClass *>(pIMyInt1);
    delete p;
}

```

```

In CMyClass::IMyInt1::mf1()
In CMyClass::IMyInt1::mf1(int)
In CMyClass::IMyInt1::mf2()
In CMyClass::IMyInt1::mf2(int)
In CMyClass::IMyInt2::mf1()
In CMyClass::IMyInt2::mf1(int)
In CMyClass::IMyInt2::mf2()
In CMyClass::IMyInt2::mf2(int)

```

See also

[Inheritance](#)

# Abstract Classes (C++)

---

Abstract classes act as expressions of general concepts from which more specific classes can be derived. You cannot create an object of an abstract class type; however, you can use pointers and references to abstract class types.

A class that contains at least one pure virtual function is considered an abstract class. Classes derived from the abstract class must implement the pure virtual function or they, too, are abstract classes.

Consider the example presented in [Virtual Functions](#). The intent of class `Account` is to provide general functionality, but objects of type `Account` are too general to be useful. Therefore, `Account` is a good candidate for an abstract class:

```
// deriv_AbstractClasses.cpp
// compile with: /LD
class Account {
public:
    Account( double d );    // Constructor.
    virtual double GetBalance();    // Obtain balance.
    virtual void PrintBalance() = 0;    // Pure virtual function.
private:
    double _balance;
};
```

The only difference between this declaration and the previous one is that `PrintBalance` is declared with the pure specifier (`= 0`).

## Restrictions on abstract classes

Abstract classes cannot be used for:

- Variables or member data
- Argument types
- Function return types
- Types of explicit conversions

Another restriction is that if the constructor for an abstract class calls a pure virtual function, either directly or indirectly, the result is undefined. However, constructors and destructors for abstract classes can call other member functions.

Pure virtual functions can be defined for abstract classes, but they can be called directly only by using the syntax:

*abstract-class-name::function-name()*



This helps when designing class hierarchies whose base class(es) include pure virtual destructors, because base class destructors are always called in the process of destroying an object. Consider the following example:

```
// Declare an abstract base class with a pure virtual destructor.
// deriv_RestriktionsonUsingAbstractClasses.cpp
class base {
public:
    base() {}
    virtual ~base()=0;
};

// Provide a definition for destructor.
base::~~base() {}

class derived:public base {
public:
    derived() {}
    ~derived(){}
};

int main() {
    derived *pDerived = new derived;
    delete pDerived;
}
```

When the object pointed to by `pDerived` is deleted, the destructor for class `derived` is called and then the destructor for class `base` is called. The empty implementation for the pure virtual function ensures that at least some implementation exists for the function.

[!NOTE] In the preceding example, the pure virtual function `base::~~base` is called implicitly from `derived::~~derived`. It is also possible to call pure virtual functions explicitly using a fully qualified member-function name.

## See also

[Inheritance](#)

# Summary of Scope Rules

---

The use of a name must be unambiguous within its scope (up to the point where overloading is determined). If the name denotes a function, the function must be unambiguous with respect to number and type of parameters. If the name remains unambiguous, [member-access](#) rules are applied.

## Constructor initializers

[Constructor initializers](#) are evaluated in the scope of the outermost block of the constructor for which they are specified. Therefore, they can use the constructor's parameter names.

## Global names

A name of an object, function, or enumerator is global if it is introduced outside any function or class or prefixed by the global unary scope operator (`::`), and if it is not used in conjunction with any of these binary operators:

- Scope-resolution (`::`)
- Member-selection for objects and references (`.`)
- Member-selection for pointers (`->`)

## Qualified names

Names used with the binary scope-resolution operator (`::`) are called "qualified names." The name specified after the binary scope-resolution operator must be a member of the class specified on the left of the operator or a member of its base class(es).

Names specified after the member-selection operator (`.` or `->`) must be members of the class type of the object specified on the left of the operator or members of its base class(es). Names specified on the right of the member-selection operator (`->`) can also be objects of another class type, provided that the left-hand side of `->` is a class object and that the class defines an overloaded member-selection operator (`->`) that evaluates to a pointer to some other class type. (This provision is discussed in more detail in [Class Member Access](#).)

The compiler searches for names in the following order, stopping when the name is found:

1. Current block scope if name is used inside a function; otherwise, global scope.
2. Outward through each enclosing block scope, including the outermost function scope (which includes function parameters).
3. If the name is used inside a member function, the class's scope is searched for the name.
4. The class's base classes are searched for the name.
5. The enclosing nested class scope (if any) and its bases are searched. The search continues until the outermost enclosing class scope is searched.
6. Global scope is searched.

However, you can make modifications to this search order as follows:

1. Names preceded by `::` force the search to begin at global scope.
2. Names preceded by the **class**, **struct**, and **union** keywords force the compiler to search only for **class**, **struct**, or **union** names.
3. Names on the left side of the scope-resolution operator (`::`) can be only **class**, **struct**, **namespace**, or **union** names.

If the name refers to a nonstatic member but is used in a static member function, an error message is generated. Similarly, if the name refers to any nonstatic member in an enclosing class, an error message is generated because enclosed classes do not have enclosing-class **this** pointers.

## Function parameter names

Function parameter names in function definitions are considered to be in the scope of the outermost block of the function. Therefore, they are local names and go out of scope when the function is exited.

Function parameter names in function declarations (prototypes) are in local scope of the declaration and go out of scope at the end of the declaration.

Default parameters are in the scope of the parameter for which they are the default, as described in the preceding two paragraphs. However, they cannot access local variables or nonstatic class members. Default parameters are evaluated at the point of the function call, but they are evaluated in the function declaration's original scope. Therefore, the default parameters for member functions are always evaluated in class scope.

## See also

[Inheritance](#)

# Inheritance Keywords

---

## Microsoft Specific

```
class [__single_inheritance] class-name;  
class [__multiple_inheritance] class-name;  
class [__virtual_inheritance] class-name;
```

where:

*class-name*

The name of the class being declared.

C++ allows you to declare a pointer to a class member prior to the definition of the class. For example:

```
class S;  
int S::*p;
```

In the code above, `p` is declared to be a pointer to integer member of class `S`. However, `class S` has not yet been defined in this code; it has only been declared. When the compiler encounters such a pointer, it must make a generalized representation of the pointer. The size of the representation is dependent on the inheritance model specified. There are four ways to specify an inheritance model to the compiler:

- In the IDE under **Pointer-to-member representation**
- At the command line using the `/vmg` switch
- Using the `pointers_to_members` pragma
- Using the inheritance keywords `__single_inheritance`, `__multiple_inheritance`, and `__virtual_inheritance`. This technique controls the inheritance model on a per-class basis.

[!NOTE] If you always declare a pointer to a member of a class after defining the class, you don't need to use any of these options.

Declaring a pointer to a member of a class prior to the class definition affects the size and speed of the resulting executable file. The more complex the inheritance used by a class, the greater the number of bytes required to represent a pointer to a member of the class and the larger the code required to interpret the pointer. Single inheritance is least complex, and virtual inheritance is most complex.

If the example above is changed to:

```
class __single_inheritance S;  
int S::*p;
```

regardless of command-line options or pragmas, pointers to members of `class S` will use the smallest possible representation.

[!NOTE] The same forward declaration of a class pointer-to-member representation should occur in every translation unit that declares pointers to members of that class, and the declaration should occur before the pointers to members are declared.

For compatibility with previous versions, `_single_inheritance`, `_multiple_inheritance`, and `_virtual_inheritance` are synonyms for `__single_inheritance`, `__multiple_inheritance`, and `__virtual_inheritance` unless compiler option `/Za` ([Disable language extensions](#)) is specified.

**END Microsoft Specific**

See also

[Keywords](#)

# virtual (C++)

---

The **virtual** keyword declares a virtual function or a virtual base class.

## Syntax

```
virtual [type-specifiers] member-function-declarator  
virtual [access-specifier] base-class-name
```

### Parameters

*type-specifiers*

Specifies the return type of the virtual member function.

*member-function-declarator*

Declares a member function.

*access-specifier*

Defines the level of access to the base class, **public**, **protected** or **private**. Can appear before or after the **virtual** keyword.

*base-class-name*

Identifies a previously declared class type.

## Remarks

See [Virtual Functions](#) for more information.

Also see the following keywords: [class](#), [private](#), [public](#), and [protected](#).

## See also

[Keywords](#)

# \_\_super

---

## Microsoft Specific

Allows you to explicitly state that you are calling a base-class implementation for a function that you are overriding.

## Syntax

```
__super::member_function();
```

## Remarks

All accessible base-class methods are considered during the overload resolution phase, and the function that provides the best match is the one that is called.

**\_\_super** can only appear within the body of a member function.

**\_\_super** cannot be used with a using declaration. See [using Declaration](#) for more information.

With the introduction of [attributes](#) that inject code, your code might contain one or more base classes whose names you may not know but that contain methods that you wish to call.

## Example

```
// deriv_super.cpp
// compile with: /c
struct B1 {
    void mf(int) {}
};

struct B2 {
    void mf(short) {}

    void mf(char) {}
};

struct D : B1, B2 {
    void mf(short) {
        __super::mf(1);    // Calls B1::mf(int)
        __super::mf('s');  // Calls B2::mf(char)
    }
};
```

## END Microsoft Specific

See also

[Keywords](#)



# \_\_interface

---

## Microsoft Specific

A Microsoft C++ interface can be defined as follows:

- Can inherit from zero or more base interfaces.
- Cannot inherit from a base class.
- Can only contain public, pure virtual methods.
- Cannot contain constructors, destructors, or operators.
- Cannot contain static methods.
- Cannot contain data members; properties are allowed.

## Syntax

```
modifier __interface interface-name {interface-definition};
```

## Remarks

A C++ [class](#) or [struct](#) could be implemented with these rules, but **\_\_interface** enforces them.

For example, the following is a sample interface definition:

```
__interface IMyInterface {  
    HRESULT CommitX();  
    HRESULT get_X(BSTR* pbstrName);  
};
```

For information on managed interfaces, see [interface class](#).

Notice that you do not have to explicitly indicate that the `CommitX` and `get_X` functions are pure virtual. An equivalent declaration for the first function would be:

```
virtual HRESULT CommitX() = 0;
```

**\_\_interface** implies the [novtable](#) **\_\_declspec** modifier.

## Example

The following sample shows how to use properties declared in an interface.

```

// deriv_interface.cpp
#define _ATL_ATTRIBUTES 1
#include <atlbase.h>
#include <atlcom.h>
#include <string.h>
#include <comdef.h>
#include <stdio.h>

[module(name="test")];

[ object, uuid("00000000-0000-0000-0000-000000000001"), library_block ]
__interface IFace {
    [ id(0) ] int int_data;
    [ id(5) ] BSTR bstr_data;
};

[ coclass, uuid("00000000-0000-0000-0000-000000000002") ]
class MyClass : public IFace {
private:
    int m_i;
    BSTR m_bstr;

public:
    MyClass()
    {
        m_i = 0;
        m_bstr = 0;
    }

    ~MyClass()
    {
        if (m_bstr)
            ::SysFreeString(m_bstr);
    }

    int get_int_data()
    {
        return m_i;
    }

    void put_int_data(int _i)
    {
        m_i = _i;
    }

    BSTR get_bstr_data()
    {
        BSTR bstr = ::SysAllocString(m_bstr);
        return bstr;
    }

    void put_bstr_data(BSTR bstr)

```

```

    {
        if (m_bstr)
            ::SysFreeString(m_bstr);
        m_bstr = ::SysAllocString(bstr);
    }
};

int main()
{
    _bstr_t bstr("Testing");
    CoInitialize(NULL);
    CComObject<MyClass>* p;
    CComObject<MyClass>::CreateInstance(&p);
    p->int_data = 100;
    printf_s("p->int_data = %d\n", p->int_data);
    p->bstr_data = bstr;
    printf_s("bstr_data = %S\n", p->bstr_data);
}

```

```

p->int_data = 100
bstr_data = Testing

```

## END Microsoft Specific

## See also

[Keywords](#)

[Interface Attributes](#)

# Special member functions

---

The *special member functions* are class (or struct) member functions that, in certain cases, the compiler automatically generates for you. These functions are the [default constructor](#), the [destructor](#), the [copy constructor and copy assignment operator](#), and the [move constructor and move assignment operator](#). If your class does not define one or more of the special member functions, then the compiler may implicitly declare and define the functions that are used. The compiler-generated implementations are called the *default* special member functions. The compiler does not generate functions if they are not needed.

You can explicitly declare a default special member function by using the **= default** keyword. This causes the compiler to define the function only if needed, in the same way as if the function was not declared at all.

In some cases, the compiler may generate *deleted* special member functions, which are not defined and therefore not callable. This can happen in cases where a call to a particular special member function on a class doesn't make sense, given other properties of the class. To explicitly prevent automatic generation of a special member function, you can declare it as deleted by using the **= delete** keyword.

The compiler generates a *default constructor*, a constructor that takes no arguments, only when you have not declared any other constructor. If you have declared only a constructor that takes parameters, code that attempts to call a default constructor causes the compiler to produce an error message. The compiler-generated default constructor performs simple member-wise [default initialization](#) of the object. Default initialization leaves all member variables in an indeterminate state.

The default destructor performs member-wise destruction of the object. It is virtual only if a base class destructor is virtual.

The default copy and move construction and assignment operations perform member-wise bit-pattern copies or moves of non-static data members. Move operations are only generated when no destructor or move or copy operations are declared. A default copy constructor is only generated when no copy constructor is declared. It is implicitly deleted if a move operation is declared. A default copy assignment operator is generated only when no copy assignment operator is explicitly declared. It is implicitly deleted if a move operation is declared.

## See also

[C++ Language Reference](#)

# Static Members (C++)

---

Classes can contain static member data and member functions. When a data member is declared as **static**, only one copy of the data is maintained for all objects of the class.

Static data members are not part of objects of a given class type. As a result, the declaration of a static data member is not considered a definition. The data member is declared in class scope, but definition is performed at file scope. These static members have external linkage. The following example illustrates this:

```
// static_data_members.cpp
class BufferedOutput
{
public:
    // Return number of bytes written by any object of this class.
    short BytesWritten()
    {
        return bytecount;
    }

    // Reset the counter.
    static void ResetCount()
    {
        bytecount = 0;
    }

    // Static member declaration.
    static long bytecount;
};

// Define bytecount in file scope.
long BufferedOutput::bytecount;

int main()
{
}
```

In the preceding code, the member `bytecount` is declared in class `BufferedOutput`, but it must be defined outside the class declaration.

Static data members can be referred to without referring to an object of class type. The number of bytes written using `BufferedOutput` objects can be obtained as follows:

```
long nBytes = BufferedOutput::bytecount;
```

For the static member to exist, it is not necessary that any objects of the class type exist. Static members can also be accessed using the member-selection (`.` and `->`) operators. For example:

```
BufferedOutput Console;  
  
long nBytes = Console.bytecount;
```

In the preceding case, the reference to the object ([Console](#)) is not evaluated; the value returned is that of the static object [bytecount](#).

Static data members are subject to class-member access rules, so private access to static data members is allowed only for class-member functions and friends. These rules are described in [Member-Access Control](#). The exception is that static data members must be defined in file scope regardless of their access restrictions. If the data member is to be explicitly initialized, an initializer must be provided with the definition.

The type of a static member is not qualified by its class name. Therefore, the type of [BufferedOutput::bytecount](#) is **long**.

## See also

[Classes and Structs](#)

# C++ classes as value types

---

C++ classes are by default value types. They can be specified as reference types, which enable polymorphic behavior to support object-oriented programming. Value types are sometimes viewed from the perspective of memory and layout control, whereas reference types are about base classes and virtual functions for polymorphic purposes. By default, value types are copyable, which means there is always a copy constructor and a copy assignment operator. For reference types, you make the class non-copyable (disable the copy constructor and copy assignment operator) and use a virtual destructor, which supports their intended polymorphism. Value types are also about the contents, which, when they are copied, always give you two independent values that can be modified separately. Reference types are about identity - what kind of object is it? For this reason, "reference types" are also referred to as "polymorphic types".

If you really want a reference-like type (base class, virtual functions), you need to explicitly disable copying, as shown in the `MyRefType` class in the following code.

```
// cl /EHsc /nologo /W4

class MyRefType {
private:
    MyRefType & operator=(const MyRefType &);
    MyRefType(const MyRefType &);
public:
    MyRefType () {}
};

int main()
{
    MyRefType Data1, Data2;
    // ...
    Data1 = Data2;
}
```

Compiling the above code will result in the following error:

```
test.cpp(15) : error C2248: 'MyRefType::operator =' : cannot access private member
declared in class 'MyRefType'
    meow.cpp(5) : see declaration of 'MyRefType::operator ='
    meow.cpp(3) : see declaration of 'MyRefType'
```

## Value types and move efficiency

Copy allocation overhead is avoided due to new copy optimizations. For example, when you insert a string in the middle of a vector of strings, there will be no copy re-allocation overhead, only a move- even if it results in a grow of the vector itself. This also applies to other operations, for instance performing an add operation on two very large objects. How do you enable these value operation optimizations? In some C++ compilers,

the compiler will enable this for you implicitly, much like copy constructors can be automatically generated by the compiler. However, in C++, your class will need to "opt-in" to move assignment and constructors by declaring it in your class definition. This is accomplished by using the double ampersand (&&) rvalue reference in the appropriate member function declarations and defining move constructor and move assignment methods. You also need to insert the correct code to "steal the guts" out of the source object.

How do you decide if you need move enabled? If you already know you need copy construction enabled, you probably want move enabled if it can be cheaper than a deep copy. However, if you know you need move support, it doesn't necessarily mean you want copy enabled. This latter case would be called a "move-only type". An example already in the standard library is `unique_ptr`. As a side note, the old `auto_ptr` is deprecated, and was replaced by `unique_ptr` precisely due to the lack of move semantics support in the previous version of C++.

By using move semantics you can return-by-value or insert-in-middle. Move is an optimization of copy. There is need for heap allocation as a workaround. Consider the following pseudocode:

```
#include <set>
#include <vector>
#include <string>
using namespace std;

//...
set<widget> LoadHugeData() {
    set<widget> ret;
    // ... load data from disk and populate ret
    return ret;
}
//...
widgets = LoadHugeData();    // efficient, no deep copy

vector<string> v = IfIHadAMillionStrings();
v.insert( begin(v)+v.size()/2, "scott" );    // efficient, no deep copy-shuffle
v.insert( begin(v)+v.size()/2, "Andrei" );    // (just 1M ptr/len assignments)
//...
HugeMatrix operator+(const HugeMatrix& , const HugeMatrix& );
HugeMatrix operator+(const HugeMatrix& , HugeMatrix&&);
HugeMatrix operator+( HugeMatrix&&, const HugeMatrix& );
HugeMatrix operator+( HugeMatrix&&, HugeMatrix&&);
//...
hm5 = hm1+hm2+hm3+hm4+hm5;    // efficient, no extra copies
```

## Enabling move for appropriate value types

For a value-like class where move can be cheaper than a deep copy, enable move construction and move assignment for efficiency. Consider the following pseudocode:

```
#include <memory>
#include <stdexcept>
using namespace std;
```



```
// ...
class my_class {
    unique_ptr<BigHugeData> data;
public:
    my_class( my_class&& other )    // move construction
        : data( move( other.data ) ) { }
    my_class& operator=( my_class&& other )    // move assignment
    { data = move( other.data ); return *this; }
    // ...
    void method() {    // check (if appropriate)
        if( !data )
            throw std::runtime_error("RUNTIME ERROR: Insufficient resources!");
    }
};
```

If you enable copy construction/assignment, also enable move construction/assignment if it can be cheaper than a deep copy.

Some *non-value* types are move-only, such as when you can't clone a resource, only transfer ownership.

Example: `unique_ptr`.

## See also

[C++ type system](#)

[Welcome back to C++](#)

[C++ Language Reference](#)

[C++ Standard Library](#)

# User-Defined Type Conversions (C++)

---

A *conversion* produces a new value of some type from a value of a different type. *Standard conversions* are built into the C++ language and support its built-in types, and you can create *user-defined conversions* to perform conversions to, from, or between user-defined types.

The standard conversions perform conversions between built-in types, between pointers or references to types related by inheritance, to and from void pointers, and to the null pointer. For more information, see [Standard Conversions](#). User-defined conversions perform conversions between user-defined types, or between user-defined types and built-in types. You can implement them as [Conversion constructors](#) or as [Conversion functions](#).

Conversions can either be explicit—when the programmer calls for one type to be converted to another, as in a cast or direct initialization—or implicit—when the language or program calls for a different type than the one given by the programmer.

Implicit conversions are attempted when:

- An argument supplied to a function does not have the same type as the matching parameter.
- The value returned from a function does not have the same type as the function return type.
- An initializer expression does not have the same type as the object it is initializing.
- An expression that controls a conditional statement, looping construct, or switch does not have the result type that's required to control it.
- An operand supplied to an operator does not have the same type as the matching operand-parameter. For built-in operators, both operands must have the same type, and are converted to a common type that can represent both. For more information, see [Standard Conversions](#). For user-defined operators, each operand must have the same type as the matching operand-parameter.

When one standard conversion can't complete an implicit conversion, the compiler can use a user-defined conversion, followed optionally by an additional standard conversion, to complete it.

When two or more user-defined conversions that perform the same conversion are available at a conversion site, the conversion is said to be ambiguous. Such ambiguities are an error because the compiler can't determine which one of the available conversions it should choose. However, it's not an error just to define multiple ways of performing the same conversion because the set of available conversions can be different at different locations in the source code—for example, depending on which header files are included in a source file. As long as only one conversion is available at the conversion site, there is no ambiguity. There are several ways that ambiguous conversions can arise, but the most common ones are:

- Multiple inheritance. The conversion is defined in more than one base class.
- Ambiguous function call. The conversion is defined as a conversion constructor of the target type and as a conversion function of the source type. For more information, see [Conversion functions](#).

You can usually resolve an ambiguity just by qualifying the name of the involved type more fully or by performing an explicit cast to clarify your intent.

Both conversion constructors and conversion functions obey member-access control rules, but the accessibility of the conversions is only considered if and when an unambiguous conversion can be determined. This means that a conversion can be ambiguous even if the access level of a competing conversion would prevent it from being used. For more information about member accessibility, see [Member Access Control](#).

## The explicit keyword and problems with implicit conversion

By default when you create a user-defined conversion, the compiler can use it to perform implicit conversions. Sometimes this is what you want, but other times the simple rules that guide the compiler in making implicit conversions can lead it to accept code that you don't want it to.

One well-known example of an implicit conversion that can cause problems is the conversion to **bool**. There are many reasons that you might want to create a class type that can be used in a Boolean context—for example, so that it can be used to control an **if** statement or loop—but when the compiler performs a user-defined conversion to a built-in type, the compiler is allowed to apply an additional standard conversion afterwards. The intent of this additional standard conversion is to allow for things like promotion from **short** to **int**, but it also opens the door for less-obvious conversions—for example, from **bool** to **int**, which allows your class type to be used in integer contexts you never intended. This particular problem is known as the *Safe Bool Problem*. This kind of problem is where the **explicit** keyword can help.

The **explicit** keyword tells the compiler that the specified conversion can't be used to perform implicit conversions. If you wanted the syntactic convenience of implicit conversions before the **explicit** keyword was introduced, you had to either accept the unintended consequences that implicit conversion sometimes created or use less-convenient, named conversion functions as a workaround. Now, by using the **explicit** keyword, you can create convenient conversions that can only be used to perform explicit casts or direct initialization, and that won't lead to the kind of problems exemplified by the Safe Bool Problem.

The **explicit** keyword can be applied to conversion constructors since C++98, and to conversion functions since C++11. The following sections contain more information about how to use the **explicit** keyword.

## Conversion constructors

Conversion constructors define conversions from user-defined or built-in types to a user-defined type. The following example demonstrates a conversion constructor that converts from the built-in type **double** to a user-defined type `Money`.

```
#include <iostream>

class Money
{
public:
    Money() : amount{ 0.0 } {};
    Money(double _amount) : amount{ _amount } {};

    double amount;
```

```
};

void display_balance(const Money balance)
{
    std::cout << "The balance is: " << balance.amount << std::endl;
}

int main(int argc, char* argv[])
{
    Money payable{ 79.99 };

    display_balance(payable);
    display_balance(49.95);
    display_balance(9.99f);

    return 0;
}
```

Notice that the first call to the function `display_balance`, which takes an argument of type `Money`, doesn't require a conversion because its argument is the correct type. However, on the second call to `display_balance`, a conversion is needed because the type of the argument, a **double** with a value of `49.95`, is not what the function expects. The function can't use this value directly, but because there's a conversion from the type of the argument—**double**—to the type of the matching parameter—`Money`—a temporary value of type `Money` is constructed from the argument and used to complete the function call. In the third call to `display_balance`, notice that the argument is not a **double**, but is instead a **float** with a value of `9.99`—and yet the function call can still be completed because the compiler can perform a standard conversion—in this case, from **float** to **double**—and then perform the user-defined conversion from **double** to `Money` to complete the necessary conversion.

## Declaring conversion constructors

The following rules apply to declaring a conversion constructor:

- The target type of the conversion is the user-defined type that's being constructed.
- Conversion constructors typically take exactly one argument, which is of the source type. However, a conversion constructor can specify additional parameters if each additional parameter has a default value. The source type remains the type of the first parameter.
- Conversion constructors, like all constructors, do not specify a return type. Specifying a return type in the declaration is an error.
- Conversion constructors can be explicit.

## Explicit conversion constructors

By declaring a conversion constructor to be **explicit**, it can only be used to perform direct initialization of an object or to perform an explicit cast. This prevents functions that accept an argument of the class type from also implicitly accepting arguments of the conversion constructor's source type, and prevents the class type

from being copy-initialized from a value of the source type. The following example demonstrates how to define an explicit conversion constructor, and the effect it has on what code is well-formed.

```
#include <iostream>

class Money
{
public:
    Money() : amount{ 0.0 } {};
    explicit Money(double _amount) : amount{ _amount } {};

    double amount;
};

void display_balance(const Money balance)
{
    std::cout << "The balance is: " << balance.amount << std::endl;
}

int main(int argc, char* argv[])
{
    Money payable{ 79.99 };           // Legal: direct initialization is explicit.

    display_balance(payable);          // Legal: no conversion required
    display_balance(49.95);            // Error: no suitable conversion exists to
    convert from double to Money.
    display_balance((Money)9.99f);     // Legal: explicit cast to Money

    return 0;
}
```

In this example, notice that you can still use the explicit conversion constructor to perform direct initialization of `payable`. If instead you were to copy-initialize `Money payable = 79.99;`, it would be an error. The first call to `display_balance` is unaffected because the argument is the correct type. The second call to `display_balance` is an error, because the conversion constructor can't be used to perform implicit conversions. The third call to `display_balance` is legal because of the explicit cast to `Money`, but notice that the compiler still helped complete the cast by inserting an implicit cast from **float** to **double**.

Although the convenience of allowing implicit conversions can be tempting, doing so can introduce hard-to-find bugs. The rule of thumb is to make all conversion constructors explicit except when you're sure that you want a specific conversion to occur implicitly.

## Conversion functions

Conversion functions define conversions from a user-defined type to other types. These functions are sometimes referred to as "cast operators" because they, along with conversion constructors, are called when a value is cast to a different type. The following example demonstrates a conversion function that converts from the user-defined type, `Money`, to a built-in type, **double**:

```

#include <iostream>

class Money
{
public:
    Money() : amount{ 0.0 } {};
    Money(double _amount) : amount{ _amount } {};

    operator double() const { return amount; }
private:
    double amount;
};

void display_balance(const Money balance)
{
    std::cout << "The balance is: " << balance << std::endl;
}

```

Notice that the member variable `amount` is made private and that a public conversion function to type **double** is introduced just to return the value of `amount`. In the function `display_balance`, an implicit conversion occurs when the value of `balance` is streamed to standard output by using the stream insertion operator `<<`. Because no stream-insertion operator is defined for the user-defined type `Money`, but there is one for built-in type **double**, the compiler can use the conversion function from `Money` to **double** to satisfy the stream-insertion operator.

Conversion functions are inherited by derived classes. Conversion functions in a derived class only override an inherited conversion function when they convert to exactly the same type. For example, a user-defined conversion function of the derived class **operator int** does not override—or even influence—a user-defined conversion function of the base class **operator short**, even though the standard conversions define a conversion relationship between **int** and **short**.

## Declaring conversion functions

The following rules apply to declaring a conversion function:

- The target type of the conversion must be declared prior to the declaration of the conversion function. Classes, structures, enumerations, and typedefs cannot be declared within the declaration of the conversion function.

```
operator struct String { char string_storage; }() // illegal
```

- Conversion functions take no arguments. Specifying any parameters in the declaration is an error.
- Conversion functions have a return type that is specified by the name of the conversion function, which is also the name of the conversion's target type. Specifying a return type in the declaration is an error.
- Conversion functions can be virtual.

- Conversion functions can be explicit.

## Explicit conversion functions

When a conversion function is declared to be explicit, it can only be used to perform an explicit cast. This prevents functions that accept an argument of the conversion function's target type from also implicitly accepting arguments of the class type, and prevents instances of the target type from being copy-initialized from a value of the class type. The following example demonstrates how to define an explicit conversion function and the effect it has on what code is well-formed.

```
#include <iostream>

class Money
{
public:
    Money() : amount{ 0.0 } {};
    Money(double _amount) : amount{ _amount } {};

    explicit operator double() const { return amount; }
private:
    double amount;
};

void display_balance(const Money balance)
{
    std::cout << "The balance is: " << (double)balance << std::endl;
}
```

Here the conversion function **operator double** has been made explicit, and an explicit cast to type **double** has been introduced in the function **display\_balance** to perform the conversion. If this cast were omitted, the compiler would be unable to locate a suitable stream-insertion operator **<<** for type **Money** and an error would occur.

# Mutable Data Members (C++)

---

This keyword can only be applied to non-static and non-const data members of a class. If a data member is declared **mutable**, then it is legal to assign a value to this data member from a **const** member function.

## Syntax

```
mutable member-variable-declaration;
```

## Remarks

For example, the following code will compile without error because `m_accessCount` has been declared to be **mutable**, and therefore can be modified by `GetFlag` even though `GetFlag` is a const member function.

```
// mutable.cpp
class X
{
public:
    bool GetFlag() const
    {
        m_accessCount++;
        return m_flag;
    }
private:
    bool m_flag;
    mutable int m_accessCount;
};

int main()
{
}
```

## See also

[Keywords](#)



# Nested Class Declarations

---

A class can be declared within the scope of another class. Such a class is called a "nested class." Nested classes are considered to be within the scope of the enclosing class and are available for use within that scope. To refer to a nested class from a scope other than its immediate enclosing scope, you must use a fully qualified name.

The following example shows how to declare nested classes:

```
// nested_class_declarations.cpp
class BufferedIO
{
public:
    enum IOError { None, Access, General };

    // Declare nested class BufferedInput.
    class BufferedInput
    {
    public:
        int read();
        int good()
        {
            return _inputerror == None;
        }
    private:
        IOError _inputerror;
    };

    // Declare nested class BufferedOutput.
    class BufferedOutput
    {
        // Member list
    };
};

int main()
{
}
```

`BufferedIO::BufferedInput` and `BufferedIO::BufferedOutput` are declared within `BufferedIO`. These class names are not visible outside the scope of class `BufferedIO`. However, an object of type `BufferedIO` does not contain any objects of types `BufferedInput` or `BufferedOutput`.

Nested classes can directly use names, type names, names of static members, and enumerators only from the enclosing class. To use names of other class members, you must use pointers, references, or object names.

In the preceding `BufferedIO` example, the enumeration `IOError` can be accessed directly by member functions in the nested classes, `BufferedIO::BufferedInput` or `BufferedIO::BufferedOutput`, as shown

in function `good`.

[!NOTE] Nested classes declare only types within class scope. They do not cause contained objects of the nested class to be created. The preceding example declares two nested classes but does not declare any objects of these class types.

An exception to the scope visibility of a nested class declaration is when a type name is declared together with a forward declaration. In this case, the class name declared by the forward declaration is visible outside the enclosing class, with its scope defined to be the smallest enclosing non-class scope. For example:

```
// nested_class_declarations_2.cpp
class C
{
public:
    typedef class U u_t; // class U visible outside class C scope
    typedef class V {} v_t; // class V not visible outside class C
};

int main()
{
    // okay, forward declaration used above so file scope is used
    U* pu;

    // error, type name only exists in class C scope
    u_t* pu2; // C2065

    // error, class defined above so class C scope
    V* pv; // C2065

    // okay, fully qualified name
    C::V* pv2;
}
```

## Access privilege in nested classes

Nesting a class within another class does not give special access privileges to member functions of the nested class. Similarly, member functions of the enclosing class have no special access to members of the nested class.

## Member functions in nested classes

Member functions declared in nested classes can be defined in file scope. The preceding example could have been written:

```
// member_functions_in_nested_classes.cpp
class BufferedIO
{
public:
    enum IOError { None, Access, General };
};
```

```

class BufferedInput
{
public:
    int read(); // Declare but do not define member
    int good(); // functions read and good.
private:
    IOError _inputerror;
};

class BufferedOutput
{
    // Member list.
};

// Define member functions read and good in
// file scope.
int BufferedIO::BufferedInput::read()
{
    return(1);
}

int BufferedIO::BufferedInput::good()
{
    return _inputerror == None;
}
int main()
{
}

```

In the preceding example, the *qualified-type-name* syntax is used to declare the function name. The declaration:

```
BufferedIO::BufferedInput::read()
```

means "the `read` function that is a member of the `BufferedInput` class that is in the scope of the `BufferedIO` class." Because this declaration uses the *qualified-type-name* syntax, constructs of the following form are possible:

```

typedef BufferedIO::BufferedInput BIO_INPUT;

int BIO_INPUT::read()

```

The preceding declaration is equivalent to the previous one, but it uses a **typedef** name in place of the class names.

## Friend functions in nested classes

Friend functions declared in a nested class are considered to be in the scope of the nested class, not the enclosing class. Therefore, the friend functions gain no special access privileges to members or member functions of the enclosing class. If you want to use a name that is declared in a nested class in a friend function and the friend function is defined in file scope, use qualified type names as follows:

```
// friend_functions_and_nested_classes.cpp

#include <string.h>

enum
{
    sizeofMessage = 255
};

char *rgszMessage[sizeofMessage];

class BufferedIO
{
public:
    class BufferedInput
    {
    public:
        friend int GetExtendedErrorStatus();
        static char *message;
        static int  messageSize;
        int iMsgNo;
    };
};

char *BufferedIO::BufferedInput::message;
int BufferedIO::BufferedInput::messageSize;

int GetExtendedErrorStatus()
{
    int iMsgNo = 1; // assign arbitrary value as message number

    strcpy_s( BufferedIO::BufferedInput::message,
              BufferedIO::BufferedInput::messageSize,
              rgszMessage[iMsgNo] );

    return iMsgNo;
}

int main()
{
}
```

The following code shows the function `GetExtendedErrorStatus` declared as a friend function. In the function, which is defined in file scope, a message is copied from a static array into a class member. Note that a better implementation of `GetExtendedErrorStatus` is to declare it as:

```
int GetExtendedErrorStatus( char *message )
```

With the preceding interface, several classes can use the services of this function by passing a memory location where they want the error message copied.

## See also

[Classes and Structs](#)

# Anonymous Class Types

---

Classes can be anonymous — that is, they can be declared without an *identifier*. This is useful when you replace a class name with a **typedef** name, as in the following:

```
typedef struct
{
    unsigned x;
    unsigned y;
} POINT;
```

[!NOTE] The use of anonymous classes shown in the previous example is useful for preserving compatibility with existing C code. In some C code, the use of **typedef** in conjunction with anonymous structures is prevalent.

Anonymous classes are also useful when you want a reference to a class member to appear as though it were not contained in a separate class, as in the following:

```
struct PValue
{
    POINT ptLoc;
    union
    {
        int  iValue;
        long lValue;
    };
};

PValue ptv;
```

In the preceding code, *iValue* can be accessed using the object member-selection operator (.) as follows:

```
int i = ptv.iValue;
```

Anonymous classes are subject to certain restrictions. (For more information about anonymous unions, see [Unions](#).) Anonymous classes:

- Cannot have a constructor or destructor.
- Cannot be passed as arguments to functions (unless type checking is defeated using ellipses).
- Cannot be returned as return values from functions.

## Anonymous structs

## Microsoft Specific

A Microsoft C extension allows you to declare a structure variable within another structure without giving it a name. These nested structures are called anonymous structures. C++ does not allow anonymous structures.

You can access the members of an anonymous structure as if they were members in the containing structure.

```
// anonymous_structures.c
#include <stdio.h>

struct phone
{
    int areacode;
    long number;
};

struct person
{
    char name[30];
    char gender;
    int age;
    int weight;
    struct phone; // Anonymous structure; no name needed
} Jim;

int main()
{
    Jim.number = 1234567;
    printf_s("%d\n", Jim.number);
}
//Output: 1234567
```

## END Microsoft Specific

# Pointers to Members

---

Declarations of pointers to members are special cases of pointer declarations. They're declared using the following sequence:

```
storage-class-specifiersopt cv-qualifiersopt type-specifier ms-modifieropt qualified-name :: * cv-qualifiersopt identifier pm-initializeropt ;
```

1. The declaration specifier:

- An optional storage class specifier.
- Optional **const** and **volatile** specifiers.
- The type specifier: the name of a type. It's the type of the member to be pointed to, not the class.

2. The declarator:

- An optional Microsoft-specific modifier. For more information, see [Microsoft-Specific Modifiers](#).
- The qualified name of the class containing the members to be pointed to.
- The :: operator.
- The \* operator.
- Optional **const** and **volatile** specifiers.
- The identifier naming the pointer to member.

3. An optional pointer-to-member initializer:

- The = operator.
- The & operator.
- The qualified name of the class.
- The :: operator.
- The name of a non-static member of the class of the appropriate type.

As always, multiple declarators (and any associated initializers) are allowed in a single declaration. A pointer to member may not point to a static member of the class, a member of reference type, or **void**.

A pointer to a member of a class differs from a normal pointer: it has both type information for the type of the member and for the class to which the member belongs. A normal pointer identifies (has the address of) only a single object in memory. A pointer to a member of a class identifies that member in any instance of the class. The following example declares a class, **Window**, and some pointers to member data.



```
// pointers_to_members1.cpp
class Window
{
public:
    Window(); // Default constructor.
    Window( int x1, int y1, // Constructor specifying
           int x2, int y2 ); // window size.
    bool SetCaption( const char *szTitle ); // Set window caption.
    const char *GetCaption(); // Get window caption.
    char *szWinCaption; // Window caption.
};

// Declare a pointer to the data member szWinCaption.
char * Window::* pwCaption = &Window::szWinCaption;
int main()
{
}
```

In the preceding example, `pwCaption` is a pointer to any member of class `Window` that's of type `char*`. The type of `pwCaption` is `char * Window::*`. The next code fragment declares pointers to the `SetCaption` and `GetCaption` member functions.

```
const char * (Window::*pfnwGC)() = &Window::GetCaption;
bool (Window::*pfnwSC)( const char * ) = &Window::SetCaption;
```

The pointers `pfnwGC` and `pfnwSC` point to `GetCaption` and `SetCaption` of the `Window` class, respectively. The code copies information to the window caption directly using the pointer to member `pwCaption`:

```
Window wMainWindow;
Window *pwChildWindow = new Window;
char *szUntitled = "Untitled - ";
int cUntitledLen = strlen( szUntitled );

strcpy_s( wMainWindow.*pwCaption, cUntitledLen, szUntitled );
(wMainWindow.*pwCaption)[cUntitledLen - 1] = '1'; //same as
//wMainWindow.SzWinCaption [cUntitledLen - 1] = '1';
strcpy_s( pwChildWindow->*pwCaption, cUntitledLen, szUntitled );
(pwChildWindow->*pwCaption)[cUntitledLen - 1] = '2'; //same as //pwChildWindow-
>szWinCaption[cUntitledLen - 1] = '2';
```

The difference between the `.*` and `->*` operators (the pointer-to-member operators) is that the `.*` operator selects members given an object or object reference, while the `->*` operator selects members through a pointer. For more information about these operators, see [Expressions with Pointer-to-Member Operators](#).

The result of the pointer-to-member operators is the type of the member. In this case, it's `char *`.

The following code fragment invokes the member functions `GetCaption` and `SetCaption` using pointers to members:

```
// Allocate a buffer.
enum {
    sizeofBuffer = 100
};
char szCaptionBase[sizeofBuffer];

// Copy the main window caption into the buffer
// and append " [View 1]".
strcpy_s( szCaptionBase, sizeofBuffer, (wMainWindow.*pfnwGC)() );
strcat_s( szCaptionBase, sizeofBuffer, " [View 1]" );
// Set the child window's caption.
(pwChildWindow->*pfnwSC)( szCaptionBase );
```

## Restrictions on Pointers to Members

The address of a static member isn't a pointer to a member. It's a regular pointer to the one instance of the static member. Only one instance of a static member exists for all objects of a given class. That means you can use the ordinary address-of (&) and dereference (\*) operators.

## Pointers to Members and Virtual Functions

Invoking a virtual function through a pointer-to-member function works as if the function had been called directly. The correct function is looked up in the v-table and invoked.

The key to virtual functions working, as always, is invoking them through a pointer to a base class. (For more information about virtual functions, see [Virtual Functions](#).)

The following code shows how to invoke a virtual function through a pointer-to-member function:

```
// virtual_functions.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

class Base
{
public:
    virtual void Print();
};
void (Base::* bfnPrint)() = &Base::Print;
void Base::Print()
{
    cout << "Print function for class Base\n";
}

class Derived : public Base
```

```

{
    public:
    void Print(); // Print is still a virtual function.
};

void Derived :: Print()
{
    cout << "Print function for class Derived\n";
}

int main()
{
    Base    *bPtr;
    Base    bObject;
    Derived dObject;
    bPtr = &bObject;    // Set pointer to address of bObject.
    (bPtr->*bfnPrint)();
    bPtr = &dObject;    // Set pointer to address of dObject.
    (bPtr->*bfnPrint)();
}

```

//Output: Print function for class Base  
Print function for class Derived

# this pointer

---

The **this** pointer is a pointer accessible only within the nonstatic member functions of a **class**, **struct**, or **union** type. It points to the object for which the member function is called. Static member functions don't have a **this** pointer.

## Syntax

```
this  
this->member-identifier
```

## Remarks

An object's **this** pointer isn't part of the object itself. It's not reflected in the result of a **sizeof** statement on the object. When a nonstatic member function is called for an object, the compiler passes the object's address to the function as a hidden argument. For example, the following function call:

```
myDate.setMonth( 3 );
```

can be interpreted as:

```
setMonth( &myDate, 3 );
```

The object's address is available from within the member function as the **this** pointer. Most **this** pointer uses are implicit. It's legal, though unnecessary, to use an explicit **this** when referring to members of the class. For example:

```
void Date::setMonth( int mn )  
{  
    month = mn;           // These three statements  
    this->month = mn;      // are equivalent  
    (*this).month = mn;  
}
```

The expression **\*this** is commonly used to return the current object from a member function:

```
return *this;
```

The **this** pointer is also used to guard against self-reference:

```
if (&Object != this) {  
    // do not execute in cases of self-reference
```

[!NOTE] Because the **this** pointer is nonmodifiable, assignments to the **this** pointer are not allowed. Earlier implementations of C++ allowed assignment to **this**.

Occasionally, the **this** pointer is used directly — for example, to manipulate self-referential data structures, where the address of the current object is required.

## Example

```
// this_pointer.cpp  
// compile with: /EHsc  
  
#include <iostream>  
#include <string.h>  
  
using namespace std;  
  
class Buf  
{  
public:  
    Buf( char* szBuffer, size_t sizeOfBuffer );  
    Buf& operator=( const Buf & );  
    void Display() { cout << buffer << endl; }  
  
private:  
    char*    buffer;  
    size_t   sizeOfBuffer;  
};  
  
Buf::Buf( char* szBuffer, size_t sizeOfBuffer )  
{  
    sizeOfBuffer++; // account for a NULL terminator  
  
    buffer = new char[ sizeOfBuffer ];  
    if (buffer)  
    {  
        strcpy_s( buffer, sizeOfBuffer, szBuffer );  
        sizeOfBuffer = sizeOfBuffer;  
    }  
}  
  
Buf& Buf::operator=( const Buf &otherbuf )  
{  
    if( &otherbuf != this )  
    {  
        if (buffer)  
            delete [] buffer;
```

```

        sizeofBuffer = strlen( otherbuf.buffer ) + 1;
        buffer = new char[sizeofBuffer];
        strcpy_s( buffer, sizeofBuffer, otherbuf.buffer );
    }
    return *this;
}

int main()
{
    Buf myBuf( "my buffer", 10 );
    Buf yourBuf( "your buffer", 12 );

    // Display 'my buffer'
    myBuf.Display();

    // assignment operator
    myBuf = yourBuf;

    // Display 'your buffer'
    myBuf.Display();
}

```

```

my buffer
your buffer

```

## Type of the this pointer

The **this** pointer's type can be modified in the function declaration by the **const** and **volatile** keywords. To declare a function that has either of these attributes, add the keyword(s) after the function argument list.

Consider an example:

```

// type_of_this_pointer1.cpp
class Point
{
    unsigned X() const;
};
int main()
{
}

```

The preceding code declares a member function, `X`, in which the **this** pointer is treated as a **const** pointer to a **const** object. Combinations of *cv-mod-list* options can be used, but they always modify the object pointed to by the **this** pointer, not the pointer itself. The following declaration declares function `X`, where the **this** pointer is a **const** pointer to a **const** object:

```
// type_of_this_pointer2.cpp
class Point
{
    unsigned X() const;
};
int main()
{
}
```

The type of **this** in a member function is described by the following syntax. The *cv-qualifier-list* is determined from the member function's declarator. It can be **const** or **volatile** (or both). *class-type* is the name of the class:

*[cv-qualifier-list] class-type \* **const this***

In other words, the **this** pointer is always a const pointer. It can't be reassigned. The **const** or **volatile** qualifiers used in the member function declaration apply to the class instance the **this** pointer points at, in the scope of that function.

The following table explains more about how these modifiers work.

## Semantics of this modifiers

Modifier	Meaning
<b>const</b>	Can't change member data; can't invoke member functions that aren't <b>const</b> .
<b>volatile</b>	Member data is loaded from memory each time it's accessed; disables certain optimizations.

It's an error to pass a **const** object to a member function that isn't **const**.

Similarly, it's also an error to pass a **volatile** object to a member function that isn't **volatile**.

Member functions declared as **const** can't change member data — in such functions, the **this** pointer is a pointer to a **const** object.

[!NOTE] Constructors and destructors can't be declared as **const** or **volatile**. They can, however, be invoked on **const** or **volatile** objects.

## See also

[Keywords](#)

# C++ Bit Fields

Classes and structures can contain members that occupy less storage than an integral type. These members are specified as bit fields. The syntax for bit-field *member-declarator* specification follows:

## Syntax

*declarator* : *constant-expression*

## Remarks


The (optional) *declarator* is the name by which the member is accessed in the program. It must be an integral type (including enumerated types). The *constant-expression* specifies the number of bits the member occupies in the structure. Anonymous bit fields — that is, bit-field members with no identifier — can be used for padding.

[!NOTE] An unnamed bit field of width 0 forces alignment of the next bit field to the next **type** boundary, where **type** is the type of the member.

The following example declares a structure that contains bit fields:

```
// bit_fields1.cpp
// compile with: /LD
struct Date {
    unsigned short nWeekDay : 3;    // 0..7   (3 bits)
    unsigned short nMonthDay : 6;    // 0..31  (6 bits)
    unsigned short nMonth    : 5;    // 0..12  (5 bits)
    unsigned short nYear     : 8;    // 0..100 (8 bits)
};
```

The conceptual memory layout of an object of type **Date** is shown in the following figure.

Memory layout of a date object

Memory Layout of Date Object

Note that **nYear** is 8 bits long and would overflow the word boundary of the declared type, **unsigned short**. Therefore, it is begun at the beginning of a new **unsigned short**. It is not necessary that all bit fields fit in one object of the underlying type; new units of storage are allocated, according to the number of bits requested in the declaration.

### Microsoft Specific

The ordering of data declared as bit fields is from low to high bit, as shown in the figure above.

### END Microsoft Specific

If the declaration of a structure includes an unnamed field of length 0, as shown in the following example,



```
// bit_fields2.cpp
// compile with: /LD
struct Date {
    unsigned nWeekDay : 3;    // 0..7   (3 bits)
    unsigned nMonthDay : 6;    // 0..31  (6 bits)
    unsigned      : 0;    // Force alignment to next boundary.
    unsigned nMonth : 5;    // 0..12  (5 bits)
    unsigned nYear  : 8;    // 0..100 (8 bits)
};
```

then the memory layout is as shown in the following figure:



Layout of Date object with zero-length bit field

Layout of Date Object with Zero-Length Bit Field

The underlying type of a bit field must be an integral type, as described in [Built-in types](#).

If the initializer for a reference of type `const T&` is an lvalue that refers to a bit field of type `T`, the reference is not bound to the bit field directly. Instead, the reference is bound to a temporary initialized to hold the value of the bit field.

## Restrictions on bit fields

The following list details erroneous operations on bit fields:

- Taking the address of a bit field.
- Initializing a non-**const** reference with a bit field.

## See also

[Classes and Structs](#)

# Lambda Expressions in C++

---

In C++11 and later, a lambda expression—often called a *lambda*—is a convenient way of defining an anonymous function object (a *closure*) right at the location where it is invoked or passed as an argument to a function. Typically lambdas are used to encapsulate a few lines of code that are passed to algorithms or asynchronous methods. This article defines what lambdas are, compares them to other programming techniques, describes their advantages, and provides a basic example.

## Related Topics

- [Lambda expressions vs. function objects](#)
- [Working with lambda expressions](#)
- [constexpr lambda expressions](#)


## Parts of a Lambda Expression

The ISO C++ Standard shows a simple lambda that is passed as the third argument to the `std::sort()` function:

```
#include <algorithm>
#include <cmath>

void abssort(float* x, unsigned n) {
    std::sort(x, x + n,
        // Lambda expression begins
        [](float a, float b) {
            return (std::abs(a) < std::abs(b));
        } // end of lambda expression
    );
}
```

This illustration shows the parts of a lambda:

 Structural elements of a lambda expression

1. *capture clause* (Also known as the *lambda-introducer* in the C++ specification.)
2. *parameter list* Optional. (Also known as the *lambda declarator*)
3. *mutable specification* Optional.
4. *exception-specification* Optional.
5. *trailing-return-type* Optional.
6. *lambda body*.

## Capture Clause

A lambda can introduce new variables in its body (in **C++14**), and it can also access, or *capture*, variables from the surrounding scope. A lambda begins with the capture clause (*lambda-introducer* in the Standard syntax), which specifies which variables are captured, and whether the capture is by value or by reference. Variables that have the ampersand (&) prefix are accessed by reference and variables that do not have it are accessed by value.

An empty capture clause, `[ ]`, indicates that the body of the lambda expression accesses no variables in the enclosing scope.

You can use the default capture mode (*capture-default* in the Standard syntax) to indicate how to capture any outside variables that are referenced in the lambda: `[&]` means all variables that you refer to are captured by reference, and `[=]` means they are captured by value. You can use a default capture mode, and then specify the opposite mode explicitly for specific variables. For example, if a lambda body accesses the external variable `total` by reference and the external variable `factor` by value, then the following capture clauses are equivalent:

```
[&total, factor]
[factor, &total]
[&, factor]
[factor, &]
[=, &total]
[&total, =]
```

Only variables that are mentioned in the lambda are captured when a capture-default is used.

If a capture clause includes a capture-default `&`, then no *identifier* in a *capture* of that capture clause can have the form `& identifier`. Likewise, if the capture clause includes a capture-default `=`, then no *capture* of that capture clause can have the form `= identifier`. An identifier or **this** cannot appear more than once in a capture clause. The following code snippet illustrates some examples.

```
struct S { void f(int i); };

void S::f(int i) {
    [&, i]{};      // OK
    [&, &i]{};     // ERROR: i preceded by & when & is the default
    [=, this]{};   // ERROR: this when = is the default
    [=, *this]{};  // OK: captures this by value. See below.
    [i, i]{};     // ERROR: i repeated
}
```

A capture followed by an ellipsis is a pack expansion, as shown in this [variadic template](#) example:

```
template<class... Args>
void f(Args... args) {
    auto x = [args...] { return g(args...); };
}
```

```
x();  
}
```

To use lambda expressions in the body of a class method, pass the **this** pointer to the capture clause to provide access to the methods and data members of the enclosing class.

**Visual Studio 2017 version 15.3 and later** (available with [/std:c++17](#)): The **this** pointer may be captured by value by specifying **\*this** in the capture clause. Capture by value means that the entire *closure*, which is the anonymous function object that encapsulates the lambda expression, is copied to every call site where the lambda is invoked. Capture by value is useful when the lambda will execute in parallel or asynchronous operations, especially on certain hardware architectures such as NUMA.

For an example that shows how to use lambda expressions with class methods, see "Example: Using a Lambda Expression in a Method" in [Examples of Lambda Expressions](#).

When you use the capture clause, we recommend that you keep these points in mind, particularly when you use lambdas with multithreading:

- Reference captures can be used to modify variables outside, but value captures cannot. (**mutable** allows copies to be modified, but not originals.)
- Reference captures reflect updates to variables outside, but value captures do not.
- Reference captures introduce a lifetime dependency, but value captures have no lifetime dependencies. This is especially important when the lambda runs asynchronously. If you capture a local by reference in an async lambda, that local will very possibly be gone by the time the lambda runs, resulting in an access violation at run time.

## Generalized capture (C++ 14)

In C++14, you can introduce and initialize new variables in the capture clause, without the need to have those variables exist in the lambda function's enclosing scope. The initialization can be expressed as any arbitrary expression; the type of the new variable is deduced from the type produced by the expression. One benefit of this feature is that in C++14 you can capture move-only variables (such as `std::unique_ptr`) from the surrounding scope and use them in a lambda.

```
pNums = make_unique<vector<int>>(nums);  
//...  
auto a = [ptr = move(pNums)]()  
{  
    // use ptr  
};
```

## Parameter List

In addition to capturing variables, a lambda can accept input parameters. A parameter list (*lambda declarator* in the Standard syntax) is optional and in most aspects resembles the parameter list for a function.

```
auto y = [] (int first, int second)
{
    return first + second;
};
```

In **C++ 14**, if the parameter type is generic, you can use the `auto` keyword as the type specifier. This tells the compiler to create the function call operator as a template. Each instance of `auto` in a parameter list is equivalent to a distinct type parameter.

```
auto y = [] (auto first, auto second)
{
    return first + second;
};
```

A lambda expression can take another lambda expression as its argument. For more information, see "Higher-Order Lambda Expressions" in the topic [Examples of Lambda Expressions](#).

Because a parameter list is optional, you can omit the empty parentheses if you do not pass arguments to the lambda expression and its lambda-declarator does not contain *exception-specification*, *trailing-return-type*, or **mutable**.

## Mutable Specification

Typically, a lambda's function call operator is `const-by-value`, but use of the **mutable** keyword cancels this out. It does not produce mutable data members. The mutable specification enables the body of a lambda expression to modify variables that are captured by value. Some of the examples later in this article show how to use **mutable**.

## Exception Specification

You can use the `noexcept` exception specification to indicate that the lambda expression does not throw any exceptions. As with ordinary functions, the Microsoft C++ compiler generates warning [C4297](#) if a lambda expression declares the `noexcept` exception specification and the lambda body throws an exception, as shown here:

```
// throw_lambda_expression.cpp
// compile with: /W4 /EHsc
int main() // C4297 expected
{
    []() noexcept { throw 5; }();
}
```

For more information, see [Exception Specifications \(throw\)](#).

## Return Type

The return type of a lambda expression is automatically deduced. You don't have to use the `auto` keyword unless you specify a *trailing-return-type*. The *trailing-return-type* resembles the return-type part of an ordinary method or function. However, the return type must follow the parameter list, and you must include the trailing-return-type keyword `->` before the return type.

You can omit the return-type part of a lambda expression if the lambda body contains just one return statement or the expression does not return a value. If the lambda body contains one return statement, the compiler deduces the return type from the type of the return expression. Otherwise, the compiler deduces the return type to be **void**. Consider the following example code snippets that illustrate this principle.

```
auto x1 = [](int i){ return i; }; // OK: return type is int
auto x2 = []{ return{ 1, 2 }; }; // ERROR: return type is void, deducing
                                // return type from braced-init-list is not
                                valid
```

A lambda expression can produce another lambda expression as its return value. For more information, see "Higher-Order Lambda Expressions" in [Examples of Lambda Expressions](#).

## Lambda Body

The lambda body (*compound-statement* in the Standard syntax) of a lambda expression can contain anything that the body of an ordinary method or function can contain. The body of both an ordinary function and a lambda expression can access these kinds of variables:

- Captured variables from the enclosing scope, as described previously.
- Parameters
- Locally-declared variables
- Class data members, when declared inside a class and **this** is captured
- Any variable that has static storage duration—for example, global variables

The following example contains a lambda expression that explicitly captures the variable `n` by value and implicitly captures the variable `m` by reference:

```
// captures_lambda_expression.cpp
// compile with: /W4 /EHsc
#include <iostream>
using namespace std;

int main()
{
    int m = 0;
    int n = 0;
    [&, n] (int a) mutable { m = ++n + a; }(4);
    cout << m << endl << n << endl;
}
```

```
5
0
```

Because the variable `n` is captured by value, its value remains `0` after the call to the lambda expression. The **mutable** specification allows `n` to be modified within the lambda.

Although a lambda expression can only capture variables that have automatic storage duration, you can use variables that have static storage duration in the body of a lambda expression. The following example uses the `generate` function and a lambda expression to assign a value to each element in a `vector` object. The lambda expression modifies the static variable to generate the value of the next element.

```
void fillVector(vector<int>& v)
{
    // A local static variable.
    static int nextValue = 1;

    // The lambda expression that appears in the following call to
    // the generate function modifies and uses the local static
    // variable nextValue.
    generate(v.begin(), v.end(), [] { return nextValue++; });
    //WARNING: this is not thread-safe and is shown for illustration only
}
```

For more information, see [generate](#).

The following code example uses the function from the previous example, and adds an example of a lambda expression that uses the C++ Standard Library algorithm `generate_n`. This lambda expression assigns an element of a `vector` object to the sum of the previous two elements. The **mutable** keyword is used so that the body of the lambda expression can modify its copies of the external variables `x` and `y`, which the lambda expression captures by value. Because the lambda expression captures the original variables `x` and `y` by value, their values remain `1` after the lambda executes.

```
// compile with: /W4 /EHsc
#include <algorithm>
#include <iostream>
#include <vector>
#include <string>

using namespace std;

template <typename C> void print(const string& s, const C& c) {
    cout << s;

    for (const auto& e : c) {
        cout << e << " ";
    }
}
```

```

    cout << endl;
}

void fillVector(vector<int>& v)
{
    // A local static variable.
    static int nextValue = 1;

    // The lambda expression that appears in the following call to
    // the generate function modifies and uses the local static
    // variable nextValue.
    generate(v.begin(), v.end(), [] { return nextValue++; });
    //WARNING: this is not thread-safe and is shown for illustration only
}

int main()
{
    // The number of elements in the vector.
    const int elementCount = 9;

    // Create a vector object with each element set to 1.
    vector<int> v(elementCount, 1);

    // These variables hold the previous two elements of the vector.
    int x = 1;
    int y = 1;

    // Sets each element in the vector to the sum of the
    // previous two elements.
    generate_n(v.begin() + 2,
        elementCount - 2,
        [=]() mutable throw() -> int { // lambda is the 3rd parameter
            // Generate current value.
            int n = x + y;
            // Update previous two values.
            x = y;
            y = n;
            return n;
        });
    print("vector v after call to generate_n() with lambda: ", v);

    // Print the local variables x and y.
    // The values of x and y hold their initial values because
    // they are captured by value.
    cout << "x: " << x << " y: " << y << endl;

    // Fill the vector with a sequence of numbers
    fillVector(v);
    print("vector v after 1st call to fillVector(): ", v);
    // Fill the vector with the next sequence of numbers
    fillVector(v);
    print("vector v after 2nd call to fillVector(): ", v);
}

```



```
vector v after call to generate_n() with lambda: 1 1 2 3 5 8 13 21 34
x: 1 y: 1
vector v after 1st call to fillVector(): 1 2 3 4 5 6 7 8 9
vector v after 2nd call to fillVector(): 10 11 12 13 14 15 16 17 18
```

For more information, see [generate\\_n](#).

## constexpr lambda expressions

**Visual Studio 2017 version 15.3 and later** (available with [/std:c++17](#)): A lambda expression may be declared as `constexpr` or used in a constant expression when the initialization of each data member that it captures or introduces is allowed within a constant expression.

```
int y = 32;
auto answer = [y]() constexpr
{
    int x = 10;
    return y + x;
};

constexpr int Increment(int n)
{
    return [n] { return n + 1; }();
}
```

A lambda is implicitly `constexpr` if its result satisfies the requirements of a `constexpr` function:

```
auto answer = [](int n)
{
    return 32 + n;
};

constexpr int response = answer(10);
```

If a lambda is implicitly or explicitly `constexpr`, conversion to a function pointer produces a `constexpr` function:

```
auto Increment = [](int n)
{
    return n + 1;
};

constexpr int(*inc)(int) = Increment;
```

---

## Microsoft-specific

Lambdas are not supported in the following common language runtime (CLR) managed entities: **ref class**, **ref struct**, **value class**, or **value struct**.

If you are using a Microsoft-specific modifier such as [\\_\\_declspec](#), you can insert it into a lambda expression immediately after the [parameter-declaration-clause](#)—for example:

```
auto Sqr = [](int t) __declspec(code_seg("PagedMem")) -> int { return t*t; };
```

To determine whether a modifier is supported by lambdas, see the article about it in the [Microsoft-Specific Modifiers](#) section of the documentation.

In addition to C++11 Standard lambda functionality, Visual Studio supports stateless lambdas, which are omni-convertible to function pointers that use arbitrary calling conventions.

## See also

[C++ Language Reference](#)

[Function Objects in the C++ Standard Library](#)

[Function Call](#)

[for\\_each](#)

# Lambda Expression Syntax

---

This article demonstrates the syntax and structural elements of lambda expressions. For a description of lambda expressions, see [Lambda Expressions](#).

## Function Objects vs. Lambdas

When you write code, you probably use function pointers and function objects to solve problems and perform calculations, especially when you use [C++ Standard Library algorithms](#). Function pointers and function objects each have advantages and disadvantages—for example, function pointers have minimal syntactic overhead but do not retain state within a scope, and function objects can maintain state but require the syntactic overhead of a class definition.

A lambda combines the benefits of function pointers and function objects and avoids their disadvantages. Like a function objects, a lambda is flexible and can maintain state, but unlike a function object, its compact syntax doesn't require an explicit class definition. By using lambdas, you can write code that's less cumbersome and less prone to errors than the code for an equivalent function object.

The following examples compare the use of a lambda to the use of a function object. The first example uses a lambda to print to the console whether each element in a [vector](#) object is even or odd. The second example uses a function object to accomplish the same task.

## Example 1: Using a Lambda

This example passes a lambda to the **for\_each** function. The lambda prints a result that states whether each element in a [vector](#) object is even or odd.

### Code

```
// even_lambda.cpp
// compile with: cl /EHsc /nologo /W4 /MTd
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    // Create a vector object that contains 9 elements.
    vector<int> v;
    for (int i = 1; i < 10; ++i) {
        v.push_back(i);
    }

    // Count the number of even numbers in the vector by
    // using the for_each function and a lambda.
    int evenCount = 0;
    for_each(v.begin(), v.end(), [&evenCount] (int n) {
```

```

        cout << n;
        if (n % 2 == 0) {
            cout << " is even " << endl;
            ++evenCount;
        } else {
            cout << " is odd " << endl;
        }
    });

    // Print the count of even numbers to the console.
    cout << "There are " << evenCount
        << " even numbers in the vector." << endl;
}

```

```

1 is odd
2 is even
3 is odd
4 is even
5 is odd
6 is even
7 is odd
8 is even
9 is odd
There are 4 even numbers in the vector.

```

## Comments

In the example, the third argument to the **for\_each** function is a lambda. The `[&evenCount]` part specifies the capture clause of the expression, `(int n)` specifies the parameter list, and remaining part specifies the body of the expression.

## Example 2: Using a Function Object

Sometimes a lambda would be too unwieldy to extend much further than the previous example. The next example uses a function object instead of a lambda, together with the **for\_each** function, to produce the same results as Example 1. Both examples store the count of even numbers in a `vector` object. To maintain the state of the operation, the `FunctorClass` class stores the `m_evenCount` variable by reference as a member variable. To perform the operation, `FunctorClass` implements the function-call operator, **operator()**. The Microsoft C++ compiler generates code that is comparable in size and performance to the lambda code in Example 1. For a basic problem like the one in this article, the simpler lambda design is probably better than the function-object design. However, if you think that the functionality might require significant expansion in the future, then use a function object design so that code maintenance will be easier.

For more information about the **operator()**, see [Function Call](#). For more information about the **for\_each** function, see [for\\_each](#).

## Code

```

// even_functor.cpp
// compile with: /EHsc
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;

class FunctorClass
{
public:
    // The required constructor for this example.
    explicit FunctorClass(int& evenCount)
        : m_evenCount(evenCount) { }

    // The function-call operator prints whether the number is
    // even or odd. If the number is even, this method updates
    // the counter.
    void operator()(int n) const {
        cout << n;

        if (n % 2 == 0) {
            cout << " is even " << endl;
            ++m_evenCount;
        } else {
            cout << " is odd " << endl;
        }
    }

private:
    // Default assignment operator to silence warning C4512.
    FunctorClass& operator=(const FunctorClass&);

    int& m_evenCount; // the number of even variables in the vector.
};

int main()
{
    // Create a vector object that contains 9 elements.
    vector<int> v;
    for (int i = 1; i < 10; ++i) {
        v.push_back(i);
    }

    // Count the number of even numbers in the vector by
    // using the for_each function and a function object.
    int evenCount = 0;
    for_each(v.begin(), v.end(), FunctorClass(evenCount));

    // Print the count of even numbers to the console.
    cout << "There are " << evenCount
        << " even numbers in the vector." << endl;
}

```

```
1 is odd
2 is even
3 is odd
4 is even
5 is odd
6 is even
7 is odd
8 is even
9 is odd
There are 4 even numbers in the vector.
```

## See also

[Lambda Expressions](#)

[Examples of Lambda Expressions](#)

[generate](#)

[generate\\_n](#)

[for\\_each](#)

[Exception Specifications \(throw\)](#)

[Compiler Warning \(level 1\) C4297](#)

[Microsoft-Specific Modifiers](#)

# Examples of Lambda Expressions

---

This article shows how to use lambda expressions in your programs. For an overview of lambda expressions, see [Lambda Expressions](#). For more information about the structure of a lambda expression, see [Lambda Expression Syntax](#).

## Declaring Lambda Expressions

### Example 1

Because a lambda expression is typed, you can assign it to an **auto** variable or to a [function](#) object, as shown here:

### Code

```
// declaring_lambda_expressions1.cpp
// compile with: /EHsc /W4
#include <functional>
#include <iostream>

int main()
{
    using namespace std;

    // Assign the lambda expression that adds two numbers to an auto variable.
    auto f1 = [](int x, int y) { return x + y; };

    cout << f1(2, 3) << endl;

    // Assign the same lambda expression to a function object.
    function<int(int, int)> f2 = [](int x, int y) { return x + y; };

    cout << f2(3, 4) << endl;
}
```

### Output

```
5
7
```

### Remarks

For more information, see [auto](#), [function Class](#), and [Function Call](#).

Although lambda expressions are most often declared in the body of a function, you can declare them anywhere that you can initialize a variable.

## Example 2

The Microsoft C++ compiler binds a lambda expression to its captured variables when the expression is declared instead of when the expression is called. The following example shows a lambda expression that captures the local variable `i` by value and the local variable `j` by reference. Because the lambda expression captures `i` by value, the reassignment of `i` later in the program does not affect the result of the expression. However, because the lambda expression captures `j` by reference, the reassignment of `j` does affect the result of the expression.

## Code

```
// declaring_lambda_expressions2.cpp
// compile with: /EHsc /W4
#include <functional>
#include <iostream>

int main()
{
    using namespace std;

    int i = 3;
    int j = 5;

    // The following lambda expression captures i by value and
    // j by reference.
    function<int (void)> f = [i, &j] { return i + j; };

    // Change the values of i and j.
    i = 22;
    j = 44;

    // Call f and print its result.
    cout << f() << endl;
}
```

## Output

47

[\[In This Article\]](#)

## Calling Lambda Expressions



You can call a lambda expression immediately, as shown in the next code snippet. The second snippet shows how to pass a lambda as an argument to C++ Standard Library algorithms such as `find_if`.

## Example 1

This example declares a lambda expression that returns the sum of two integers and calls the expression immediately with the arguments 5 and 4:

### Code

```
// calling_lambda_expressions1.cpp
// compile with: /EHsc
#include <iostream>

int main()
{
    using namespace std;
    int n = [] (int x, int y) { return x + y; }(5, 4);
    cout << n << endl;
}
```

### Output

```
9
```

## Example 2

This example passes a lambda expression as an argument to the `find_if` function. The lambda expression returns **true** if its parameter is an even number.

### Code

```
// calling_lambda_expressions2.cpp
// compile with: /EHsc /W4
#include <list>
#include <algorithm>
#include <iostream>

int main()
{
    using namespace std;

    // Create a list of integers with a few initial elements.
    list<int> numbers;
    numbers.push_back(13);
    numbers.push_back(17);
    numbers.push_back(42);
```

```

numbers.push_back(46);
numbers.push_back(99);

// Use the find_if function and a lambda expression to find the
// first even number in the list.
const list<int>::const_iterator result =
    find_if(numbers.begin(), numbers.end(), [](int n) { return (n % 2) == 0;
});

// Print the result.
if (result != numbers.end()) {
    cout << "The first even number in the list is " << *result << "." << endl;
} else {
    cout << "The list contains no even numbers." << endl;
}
}

```

## Output

```
The first even number in the list is 42.
```

## Remarks

For more information about the `find_if` function, see [find\\_if](#). For more information about the C++ Standard Library functions that perform common algorithms, see [<algorithm>](#).

[\[In This Article\]](#)

## Nesting Lambda Expressions

### Example

You can nest a lambda expression inside another one, as shown in this example. The inner lambda expression multiplies its argument by 2 and returns the result. The outer lambda expression calls the inner lambda expression with its argument and adds 3 to the result.

### Code

```

// nesting_lambda_expressions.cpp
// compile with: /EHsc /W4
#include <iostream>

int main()
{
    using namespace std;

    // The following lambda expression contains a nested lambda
    // expression.

```

```

    int timestwoplusthree = [](int x) { return [](int y) { return y * 2; }(x) + 3;
}(5);

    // Print the result.
    cout << timestwoplusthree << endl;
}

```

## Output

13

## Remarks

In this example, `[](int y) { return y * 2; }` is the nested lambda expression.

[\[In This Article\]](#)

## Higher-Order Lambda Functions

### Example

Many programming languages support the concept of a *higher-order function*. A higher-order function is a lambda expression that takes another lambda expression as its argument or returns a lambda expression. You can use the `function` class to enable a C++ lambda expression to behave like a higher-order function. The following example shows a lambda expression that returns a `function` object and a lambda expression that takes a `function` object as its argument.

## Code

```

// higher_order_lambda_expression.cpp
// compile with: /EHsc /W4
#include <iostream>
#include <functional>

int main()
{
    using namespace std;

    // The following code declares a lambda expression that returns
    // another lambda expression that adds two numbers.
    // The returned lambda expression captures parameter x by value.
    auto addtwointegers = [](int x) -> function<int(int)> {
        return [=](int y) { return x + y; };
    };

    // The following code declares a lambda expression that takes another
    // lambda expression as its argument.
    // The lambda expression applies the argument z to the function f

```

```

// and multiplies by 2.
auto higherorder = [](const function<int(int)>& f, int z) {
    return f(z) * 2;
};

// Call the lambda expression that is bound to higherorder.
auto answer = higherorder(addtwointegers(7), 8);

// Print the result, which is (7+8)*2.
cout << answer << endl;
}

```

## Output

30

[\[In This Article\]](#)

## Using a Lambda Expression in a Function

### Example

You can use lambda expressions in the body of a function. The lambda expression can access any function or data member that the enclosing function can access. You can explicitly or implicitly capture the **this** pointer to provide access to functions and data members of the enclosing class. **Visual Studio 2017 version 15.3 and later** (available with `/std:c++17`): Capture **this** by value (`[*this]`) when the lambda will be used in asynchronous or parallel operations where the code might execute after the original object goes out of scope.

You can use the **this** pointer explicitly in a function, as shown here:

```

// capture "this" by reference
void ApplyScale(const vector<int>& v) const
{
    for_each(v.begin(), v.end(),
        [this](int n) { cout << n * _scale << endl; });
}

// capture "this" by value (Visual Studio 2017 version 15.3 and later)
void ApplyScale2(const vector<int>& v) const
{
    for_each(v.begin(), v.end(),
        [*this](int n) { cout << n * _scale << endl; });
}

```

You can also capture the **this** pointer implicitly:

```

void ApplyScale(const vector<int>& v) const
{
    for_each(v.begin(), v.end(),
        [=](int n) { cout << n * _scale << endl; });
}

```

The following example shows the `Scale` class, which encapsulates a scale value.

```

// function_lambda_expression.cpp
// compile with: /EHsc /W4
#include <algorithm>
#include <iostream>
#include <vector>

using namespace std;

class Scale
{
public:
    // The constructor.
    explicit Scale(int scale) : _scale(scale) {}

    // Prints the product of each element in a vector object
    // and the scale value to the console.
    void ApplyScale(const vector<int>& v) const
    {
        for_each(v.begin(), v.end(), [=](int n) { cout << n * _scale << endl; });
    }

private:
    int _scale;
};

int main()
{
    vector<int> values;
    values.push_back(1);
    values.push_back(2);
    values.push_back(3);
    values.push_back(4);

    // Create a Scale object that scales elements by 3 and apply
    // it to the vector object. Does not modify the vector.
    Scale s(3);
    s.ApplyScale(values);
}

```

Output

```
3
6
9
12
```

## Remarks

The `ApplyScale` function uses a lambda expression to print the product of the scale value and each element in a `vector` object. The lambda expression implicitly captures **this** so that it can access the `_scale` member.

[\[In This Article\]](#)

## Using Lambda Expressions with Templates

### Example

Because lambda expressions are typed, you can use them with C++ templates. The following example shows the `negate_all` and `print_all` functions. The `negate_all` function applies the unary **operator-** to each element in the `vector` object. The `print_all` function prints each element in the `vector` object to the console.

### Code

```
// template_lambda_expression.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <iostream>

using namespace std;

// Negates each element in the vector object. Assumes signed data type.
template <typename T>
void negate_all(vector<T>& v)
{
    for_each(v.begin(), v.end(), [](T& n) { n = -n; });
}

// Prints to the console each element in the vector object.
template <typename T>
void print_all(const vector<T>& v)
{
    for_each(v.begin(), v.end(), [](const T& n) { cout << n << endl; });
}

int main()
{
    // Create a vector of signed integers with a few elements.
    vector<int> v;
    v.push_back(34);
```

```

    v.push_back(-43);
    v.push_back(56);

    print_all(v);
    negate_all(v);
    cout << "After negate_all():" << endl;
    print_all(v);
}

```

## Output

```

34
-43
56
After negate_all():
-34
43
-56

```

## Remarks

For more information about C++ templates, see [Templates](#).

[\[In This Article\]](#)

# Handling Exceptions

## Example

The body of a lambda expression follows the rules for both structured exception handling (SEH) and C++ exception handling. You can handle a raised exception in the body of a lambda expression or defer exception handling to the enclosing scope. The following example uses the **for\_each** function and a lambda expression to fill a **vector** object with the values of another one. It uses a **try/catch** block to handle invalid access to the first vector.

## Code

```

// eh_lambda_expression.cpp
// compile with: /EHsc /W4
#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;

int main()
{
    // Create a vector that contains 3 elements.
    vector<int> elements(3);
}

```

```

// Create another vector that contains index values.
vector<int> indices(3);
indices[0] = 0;
indices[1] = -1; // This is not a valid subscript. It will trigger an
exception.
indices[2] = 2;

// Use the values from the vector of index values to
// fill the elements vector. This example uses a
// try/catch block to handle invalid access to the
// elements vector.
try
{
    for_each(indices.begin(), indices.end(), [&](int index) {
        elements.at(index) = index;
    });
}
catch (const out_of_range& e)
{
    cerr << "Caught '" << e.what() << "'." << endl;
};
}

```

## Output

```
Caught 'invalid vector<T> subscript'.
```

## Remarks

For more information about exception handling, see [Exception Handling](#).

[\[In This Article\]](#)

## Using Lambda Expressions with Managed Types (C++/CLI)

### Example

The capture clause of a lambda expression cannot contain a variable that has a managed type. However, you can pass an argument that has a managed type to the parameter list of a lambda expression. The following example contains a lambda expression that captures the local unmanaged variable `ch` by value and takes a [xref:System.String?displayProperty=fullName](#) object as its parameter.

### Code

```

// managed_lambda_expression.cpp
// compile with: /clr
using namespace System;

```



```
int main()
{
    char ch = '!'; // a local unmanaged variable

    // The following lambda expression captures local variables
    // by value and takes a managed String object as its parameter.
    [=](String ^s) {
        Console::WriteLine(s + Convert::ToChar(ch));
    }("Hello");
}
```

## Output

Hello!

## Remarks

You can also use lambda expressions with the STL/CLR library. For more information, see [STL/CLR Library Reference](#).

[!IMPORTANT] Lambdas are not supported in these common language runtime (CLR) managed entities: **ref class**, **ref struct**, **value class**, and **value struct**.

[\[In This Article\]](#)

## See also

[Lambda Expressions](#)

[Lambda Expression Syntax](#)

[auto](#)

[function Class](#)

[find\\_if](#)

[<algorithm>](#)

[Function Call](#)

[Templates](#)

[Exception Handling](#)

[STL/CLR Library Reference](#)

# constexpr lambda expressions in C++

---

**Visual Studio 2017 version 15.3 and later** (available with [/std:c++17](#)): A lambda expression may be declared as **constexpr** or used in a constant expression when the initialization of each data member that it captures or introduces is allowed within a constant expression.

```
int y = 32;
auto answer = [y]() constexpr
{
    int x = 10;
    return y + x;
};

constexpr int Increment(int n)
{
    return [n] { return n + 1; }();
}
```

A lambda is implicitly **constexpr** if its result satisfies the requirements of a **constexpr** function:

```
auto answer = [](int n)
{
    return 32 + n;
};

constexpr int response = answer(10);
```

If a lambda is implicitly or explicitly **constexpr**, and you convert it to a function pointer, the resulting function is also **constexpr**:

```
auto Increment = [](int n)
{
    return n + 1;
};

constexpr int(*inc)(int) = Increment;
```

## See also

[C++ Language Reference](#)

[Function Objects in the C++ Standard Library](#)

[Function Call](#)

[for\\_each](#)

# Arrays (C++)

---

An array is a sequence of objects of the same type that occupy a contiguous area of memory. Traditional C-style arrays are the source of many bugs, but are still common, especially in older code bases. In modern C++, we strongly recommend using `std::vector` or `std::array` instead of C-style arrays described in this section. Both of these standard library types store their elements as a contiguous block of memory but provide much greater type safety along with iterators that are guaranteed to point to a valid location within the sequence. For more information, see [Containers \(Modern C++\)](#).

## Stack declarations

In a C++ array declaration, the array size is specified after the variable name, not after the type name as in some other languages. The following example declares an array of 1000 doubles to be allocated on the stack. The number of elements must be supplied as an integer literal or else as a constant expression because the compiler has to know how much stack space to allocate; it cannot use a value computed at run-time. Each element in the array is assigned a default value of 0. If you do not assign a default value, each element will initially contain whatever random values happen to be at that location.

```
constexpr size_t size = 1000;

// Declare an array of doubles to be allocated on the stack
double numbers[size] {0};

// Assign a new value to the first element
numbers[0] = 1;

// Assign a value to each subsequent element
// (numbers[1] is the second element in the array.)
for (size_t i = 1; i < size; i++)
{
    numbers[i] = numbers[i-1] * 1.1;
}

// Access each element
for (size_t i = 0; i < size; i++)
{
    std::cout << numbers[i] << " ";
}
```

The first element in the array is the 0th element, and the last element is the  $(n-1)$  element, where  $n$  is the number of elements the array can contain. The number of elements in the declaration must be of an integral type and must be greater than 0. It is your responsibility to ensure that your program never passes a value to the subscript operator that is greater than `(size - 1)`.

A zero-sized array is legal only when the array is the last field in a **struct** or **union** and when the Microsoft extensions (/Ze) are enabled.

Stack-based arrays are faster to allocate and access than heap-based arrays, but the number of elements can't be so large that it uses up too much stack memory. How much is too much depends on your program. You can use profiling tools to determine whether an array is too large.

## Heap declarations

If you require an array that is too large to be allocated on the stack, or whose size cannot be known at compile time, you can allocate it on the heap with a `new[]` expression. The operator returns a pointer to the first element. You can use the subscript operator with the pointer variable just as with a stack-based array. You can also use [pointer arithmetic](#) to move the pointer to any arbitrary elements in the array. It is your responsibility to ensure that:

- you always keep a copy of the original pointer address so that you can delete the memory when you no longer need the array.
- you do not increment or decrement the pointer address past the array bounds.

The following example shows how to define an array on the heap at run time, and how to access the array elements using the subscript operator or by using pointer arithmetic:

```
void do_something(size_t size)
{
    // Declare an array of doubles to be allocated on the heap
    double* numbers = new double[size]{ 0 };

    // Assign a new value to the first element
    numbers[0] = 1;

    // Assign a value to each subsequent element
    // (numbers[1] is the second element in the array.)
    for (size_t i = 1; i < size; i++)
    {
        numbers[i] = numbers[i - 1] * 1.1;
    }

    // Access each element with subscript operator
    for (size_t i = 0; i < size; i++)
    {
        std::cout << numbers[i] << " ";
    }

    // Access each element with pointer arithmetic
    // Use a copy of the pointer for iterating
    double* p = numbers;

    for (size_t i = 0; i < size; i++)
    {
        // Dereference the pointer, then increment it
        std::cout << *p++ << " ";
    }
}
```

```

// Alternate method:
// Reset p to numbers[0]:
p = numbers;

// Use address of pointer to compute bounds.
// The compiler computes size as the number
// of elements * (bytes per element).
while (p < (numbers + size))
{
    // Dereference the pointer, then increment it
    std::cout << *p++ << " ";
}

delete[] numbers; // don't forget to do this!
}
int main()
{
    do_something(108);
}

```

## Initializing arrays

You can initialize an array in a loop, one element at a time, or in a single statement. The contents of the following two arrays are identical:

```

int a[10];
for (int i = 0; i < 10; ++i)
{
    a[i] = i + 1;
}

int b[10]{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

```

## Passing arrays to functions

When an array is passed to a function, it is passed as a pointer to the first element. This is true for both stack-based and heap-based arrays. The pointer contains no additional size or type information. This behavior is called *pointer decay*. When you pass an array to a function, you must always specify the number of elements in a separate parameter. This behavior also implies that the array elements are not copied when the array is passed to a function. To prevent the function from modifying the elements, specify the parameter as a pointer to **const** elements.

The following example shows a function that accepts an array and a length. The pointer points to the original array, not a copy. Because the parameter is not **const**, the function can modify the array elements.

```
void process(double p*, const size_t len)
{
    std::cout << "process:\n";
    for (size_t i = 0; i < len; ++i)
    {
        // do something with p[i]
    }
}
```

Declare the array as const to make it read-only within the function block:

```
void process(const double p*, const size_t len);
```

The same function can also be declared in these ways, with no change in behavior. The array is still passed as a pointer to the first element:

```
// Unsized array
void process(const double p[] const size_t len);

// Fixed-size array. Length must still be specified explicitly.
void process(const double p[1000], const size_t len);
```

## Multidimensional arrays

Arrays constructed from other arrays are multidimensional arrays. These multidimensional arrays are specified by placing multiple bracketed constant expressions in sequence. For example, consider this declaration:

```
int i2[5][7];
```

It specifies an array of type **int**, conceptually arranged in a two-dimensional matrix of five rows and seven columns, as shown in the following figure:



Conceptual layout of a multi-dimensional array

Conceptual layout of a multi-dimensional array

In declarations of multidimensioned arrays that have an initializer list (as described in [Initializers](#)), the constant expression that specifies the bounds for the first dimension can be omitted. For example:

```
// arrays2.cpp
// compile with: /c
const int cMarkets = 4;
// Declare a float that represents the transportation costs.
double TransportCosts[][cMarkets] = {
```

```

    { 32.19, 47.29, 31.99, 19.11 },
    { 11.29, 22.49, 33.47, 17.29 },
    { 41.97, 22.09, 9.76, 22.55 }
};

```

The preceding declaration defines an array that is three rows by four columns. The rows represent factories and the columns represent markets to which the factories ship. The values are the transportation costs from the factories to the markets. The first dimension of the array is left out, but the compiler fills it in by examining the initializer.

Use of the indirection operator (\*) on an n-dimensional array type yields an n-1 dimensional array. If n is 1, a scalar (or array element) is yielded.

C++ arrays are stored in row-major order. Row-major order means the last subscript varies the fastest.

## Example

The technique of omitting the bounds specification for the first dimension of a multidimensional array can also be used in function declarations as follows:

```

// multidimensional_arrays.cpp
// compile with: /EHsc
// arguments: 3
#include <limits>    // Includes DBL_MAX
#include <iostream>

const int cMkts = 4, cFacts = 2;

// Declare a float that represents the transportation costs
double TransportCosts[][cMkts] = {
    { 32.19, 47.29, 31.99, 19.11 },
    { 11.29, 22.49, 33.47, 17.29 },
    { 41.97, 22.09, 9.76, 22.55 }
};

// Calculate size of unspecified dimension
const int cFactories = sizeof TransportCosts /
    sizeof( double[cMkts] );

double FindMinToMkt( int Mkt, double myTransportCosts[][cMkts], int mycFacts);

using namespace std;

int main( int argc, char *argv[] ) {
    double MinCost;

    if (argv[1] == 0) {
        cout << "You must specify the number of markets." << endl;
        exit(0);
    }
    MinCost = FindMinToMkt( *argv[1] - '0', TransportCosts, cFacts);
}

```

```

    cout << "The minimum cost to Market " << argv[1] << " is: "
          << MinCost << "\n";
}

double FindMinToMkt(int Mkt, double myTransportCosts[][cMkts], int mycFacts) {
    double MinCost = DBL_MAX;

    for( size_t i = 0; i < cFacts; ++i )
        MinCost = (MinCost < TransportCosts[i][Mkt]) ?
                    MinCost : TransportCosts[i][Mkt];

    return MinCost;
}

```

The minimum cost to Market 3 is: 17.29

The function `FindMinToMkt` is written such that adding new factories does not require any code changes, just a recompilation.

## Initializing Arrays

If a class has a constructor, arrays of that class are initialized by a constructor. If there are fewer items in the initializer list than elements in the array, the default constructor is used for the remaining elements. If no default constructor is defined for the class, the initializer list must be complete — that is, there must be one initializer for each element in the array.

Consider the `Point` class that defines two constructors:

```

// initializing_arrays1.cpp
class Point
{
public:
    Point()    // Default constructor.
    {
    }
    Point( int, int )    // Construct from two ints
    {
    }
};

// An array of Point objects can be declared as follows:
Point aPoint[3] = {
    Point( 3, 3 )    // Use int, int constructor.
};

int main()
{
}

```



The first element of `aPoint` is constructed using the constructor `Point( int, int );`; the remaining two elements are constructed using the default constructor.

Static member arrays (whether **const** or not) can be initialized in their definitions (outside the class declaration). For example:

```
// initializing_arrays2.cpp
class WindowColors
{
public:
    static const char *rgszWindowPartList[7];

const char *WindowColors::rgszWindowPartList[7] = {
    "Active Title Bar", "Inactive Title Bar", "Title Bar Text",
    "Menu Bar", "Menu Bar Text", "Window Background", "Frame"    };
int main()
{
}
```

## Accessing array elements

You can access individual elements of an array by using the array subscript operator (`[ ]`). If a one-dimensional array is used in an expression that has no subscript, the array name evaluates to a pointer to the first element in the array.

```
// using_arrays.cpp
int main() {
    char chArray[10];
    char *pch = chArray;    // Evaluates to a pointer to the first element.
    char  ch = chArray[0];   // Evaluates to the value of the first element.
    ch = chArray[3];        // Evaluates to the value of the fourth element.
}
```

When you use multidimensional arrays, you can use various combinations in expressions.

```
// using_arrays_2.cpp
// compile with: /EHsc /W1
#include <iostream>
using namespace std;
int main() {
    double multi[4][4][3];    // Declare the array.
    double (*p2multi)[3];
    double (*p1multi);

    cout << multi[3][2][2] << "\n";    // C4700 Use three subscripts.
}
```

```

    p2multi = multi[3];           // Make p2multi point to
                                // fourth "plane" of multi.

    p1multi = multi[3][2];       // Make p1multi point to
                                // fourth plane, third row
                                // of multi.

}

```

In the preceding code, `multi` is a three-dimensional array of type **double**. The `p2multi` pointer points to an array of type **double** of size three. In this example, the array is used with one, two, and three subscripts. Although it is more common to specify all subscripts, as in the `cout` statement, it is sometimes useful to select a specific subset of array elements, as shown in the statements that follow `cout`.

## Overloading subscript operator

Like other operators, the subscript operator (`[]`) can be redefined by the user. The default behavior of the subscript operator, if not overloaded, is to combine the array name and the subscript using the following method:

```
*((array_name) + (subscript))
```

As in all addition that involves pointer types, scaling is performed automatically to adjust for the size of the type. Therefore, the resultant value is not  $n$  bytes from the origin of array-name; rather, it is the  $n$ th element of the array. For more information about this conversion, see [Additive operators](#).

Similarly, for multidimensional arrays, the address is derived using the following method:

```
((array_name) + (subscript1 * max2 * max3 * ... * maxn) + (subscript2 * max3 * ... *
maxn) + ... + subscriptn))
```

## Arrays in Expressions

When an identifier of an array type appears in an expression other than `sizeof`, address-of (`&`), or initialization of a reference, it is converted to a pointer to the first array element. For example:

```

char szError1[] = "Error: Disk drive not ready.";
char *psz = szError1;

```

The pointer `psz` points to the first element of the array `szError1`. Arrays, unlike pointers, are not modifiable l-values. Therefore, the following assignment is illegal:

```
szError1 = psz;
```

## See also

[std::array](#)

# References (C++)

---

A reference, like a pointer, stores the address of an object that is located elsewhere in memory. Unlike a pointer, a reference after it is initialized cannot be made to refer to a different object or set to null. There are two kinds of references: lvalue references which refer to a named variable and rvalue references which refer to a [temporary object](#). The & operator signifies an lvalue reference and the && operator signifies either an rvalue reference, or a universal reference (either rvalue or lvalue) depending on the context.

References may be declared using the following syntax:

```
[storage-class-specifiers] [cv-qualifiers] type-specifiers [ms-modifier] declarator [= expression];
```

Any valid declarator specifying a reference may be used. Unless the reference is a reference to function or array type, the following simplified syntax applies:

```
[storage-class-specifiers] [cv-qualifiers] type-specifiers [& or &&] [cv-qualifiers] identifier [= expression];
```

References are declared using the following sequence:

1. The declaration specifiers:

- An optional storage class specifier.
- Optional **const** and/or **volatile** qualifiers.
- The type specifier: the name of a type.

2. The declarator:

- An optional Microsoft-specific modifier. For more information, see [Microsoft-Specific Modifiers](#).
- The **&** operator or **&&** operator.
- Optional **const** and/or **volatile** qualifiers.
- The identifier.

3. An optional initializer.

The more complex declarator forms for pointers to arrays and functions also apply to references to arrays and functions. For more information, see [pointers](#).

Multiple declarators and initializers may appear in a comma-separated list following a single declaration specifier. For example:

```
int &i;  
int &i, &j;
```

References, pointers and objects may be declared together:

```
int &ref, *ptr, k;
```

A reference holds the address of an object, but behaves syntactically like an object.

In the following program, notice that the name of the object, `s`, and the reference to the object, `SRef`, can be used identically in programs:

## Example

```
// references.cpp
#include <stdio.h>
struct S {
    short i;
};

int main() {
    S s;    // Declare the object.
    S& SRef = s;    // Declare the reference.
    s.i = 3;

    printf_s("%d\n", s.i);
    printf_s("%d\n", SRef.i);

    SRef.i = 4;
    printf_s("%d\n", s.i);
    printf_s("%d\n", SRef.i);
}
```

```
3
3
4
4
```

## See also

[Reference-Type Function Arguments](#)

[Reference-Type Function Returns](#)

[References to Pointers](#)

# Lvalue Reference Declarator: &

---

Holds the address of an object but behaves syntactically like an object.

## Syntax

```
type-id & cast-expression
```

## Remarks

You can think of an lvalue reference as another name for an object. An lvalue reference declaration consists of an optional list of specifiers followed by a reference declarator. A reference must be initialized and cannot be changed.

Any object whose address can be converted to a given pointer type can also be converted to the similar reference type. For example, any object whose address can be converted to type `char *` can also be converted to type `char &`.

Do not confuse reference declarations with use of the [address-of operator](#). When the `&identifier` is preceded by a type, such as `int` or `char`, `identifier` is declared as a reference to the type. When `&identifier` is not preceded by a type, the usage is that of the address-of operator.

## Example

The following example demonstrates the reference declarator by declaring a `Person` object and a reference to that object. Because `rFriend` is a reference to `myFriend`, updating either variable changes the same object.

```
// reference_declarator.cpp
// compile with: /EHsc
// Demonstrates the reference declarator.
#include <iostream>
using namespace std;

struct Person
{
    char* Name;
    short Age;
};

int main()
{
    // Declare a Person object.
    Person myFriend;

    // Declare a reference to the Person object.
    Person& rFriend = myFriend;
```

```
// Set the fields of the Person object.  
// Updating either variable changes the same object.  
myFriend.Name = "Bill";  
rFriend.Age = 40;  
  
// Print the fields of the Person object to the console.  
cout << rFriend.Name << " is " << myFriend.Age << endl;  
}
```

```
Bill is 40
```

## See also

[References](#)

[Reference-Type Function Arguments](#)

[Reference-Type Function Returns](#)

[References to Pointers](#)

# Rvalue Reference Declarator: &&

---

Holds a reference to an rvalue expression.

## Syntax

```
type-id && cast-expression
```

## Remarks

Rvalue references enable you to distinguish an lvalue from an rvalue. Lvalue references and rvalue references are syntactically and semantically similar, but they follow somewhat different rules. For more information about lvalues and rvalues, see [Lvalues and Rvalues](#). For more information about lvalue references, see [Lvalue Reference Declarator: &](#).

The following sections describe how rvalue references support the implementation of *move semantics* and *perfect forwarding*.

## Move Semantics

Rvalue references support the implementation of *move semantics*, which can significantly increase the performance of your applications. Move semantics enables you to write code that transfers resources (such as dynamically allocated memory) from one object to another. Move semantics works because it enables resources to be transferred from temporary objects that cannot be referenced elsewhere in the program.

To implement move semantics, you typically provide a *move constructor*, and optionally a move assignment operator (**operator=**), to your class. Copy and assignment operations whose sources are rvalues then automatically take advantage of move semantics. Unlike the default copy constructor, the compiler does not provide a default move constructor. For more information about how to write a move constructor and how to use it in your application, see [Move Constructors and Move Assignment Operators \(C++\)](#).

You can also overload ordinary functions and operators to take advantage of move semantics. Visual Studio 2010 introduces move semantics into the C++ Standard Library. For example, the `string` class implements operations that perform move semantics. Consider the following example that concatenates several strings and prints the result:

```
// string_concatenation.cpp
// compile with: /EHsc
#include <iostream>
#include <string>
using namespace std;

int main()
{
    string s = string("h") + "e" + "ll" + "o";
```

```
    cout << s << endl;
}
```

Before Visual Studio 2010, each call to **operator+** allocates and returns a new temporary **string** object (an rvalue). **operator+** cannot append one string to the other because it does not know whether the source strings are lvalues or rvalues. If the source strings are both lvalues, they might be referenced elsewhere in the program and therefore must not be modified. By using rvalue references, **operator+** can be modified to take rvalues, which cannot be referenced elsewhere in the program. Therefore, **operator+** can now append one string to another. This can significantly reduce the number of dynamic memory allocations that the **string** class must perform. For more information about the **string** class, see [basic\\_string Class](#).

Move semantics also helps when the compiler cannot use Return Value Optimization (RVO) or Named Return Value Optimization (NRVO). In these cases, the compiler calls the move constructor if the type defines it. For more information about Named Return Value Optimization, see [Named Return Value Optimization in Visual Studio 2005](#).

To better understand move semantics, consider the example of inserting an element into a **vector** object. If the capacity of the **vector** object is exceeded, the **vector** object must reallocate memory for its elements and then copy each element to another memory location to make room for the inserted element. When an insertion operation copies an element, it creates a new element, calls the copy constructor to copy the data from the previous element to the new element, and then destroys the previous element. Move semantics enables you to move objects directly without having to perform expensive memory allocation and copy operations.

To take advantage of move semantics in the **vector** example, you can write a move constructor to move data from one object to another.

For more information about the introduction of move semantics into the C++ Standard Library in Visual Studio 2010, see [C++ Standard Library](#).

## Perfect Forwarding

Perfect forwarding reduces the need for overloaded functions and helps avoid the forwarding problem. The *forwarding problem* can occur when you write a generic function that takes references as its parameters and it passes (or *forwards*) these parameters to another function. For example, if the generic function takes a parameter of type **const T&**, then the called function cannot modify the value of that parameter. If the generic function takes a parameter of type **T&**, then the function cannot be called by using an rvalue (such as a temporary object or integer literal).

Ordinarily, to solve this problem, you must provide overloaded versions of the generic function that take both **T&** and **const T&** for each of its parameters. As a result, the number of overloaded functions increases exponentially with the number of parameters. Rvalue references enable you to write one version of a function that accepts arbitrary arguments and forwards them to another function as if the other function had been called directly.

Consider the following example that declares four types, **W**, **X**, **Y**, and **Z**. The constructor for each type takes a different combination of **const** and non-**const** lvalue references as its parameters.



```

struct W
{
    W(int&, int&) {}
};

struct X
{
    X(const int&, int&) {}
};

struct Y
{
    Y(int&, const int&) {}
};

struct Z
{
    Z(const int&, const int&) {}
};

```

Suppose you want to write a generic function that generates objects. The following example shows one way to write this function:

```

template <typename T, typename A1, typename A2>
T* factory(A1& a1, A2& a2)
{
    return new T(a1, a2);
}

```

The following example shows a valid call to the `factory` function:

```

int a = 4, b = 5;
W* pw = factory<W>(a, b);

```

However, the following example does not contain a valid call to the `factory` function because `factory` takes lvalue references that are modifiable as its parameters, but it is called by using rvalues:

```

Z* pz = factory<Z>(2, 2);

```

Ordinarily, to solve this problem, you must create an overloaded version of the `factory` function for every combination of `A&` and `const A&` parameters. Rvalue references enable you to write one version of the `factory` function, as shown in the following example:

```
template <typename T, typename A1, typename A2>
T* factory(A1&& a1, A2&& a2)
{
    return new T(std::forward<A1>(a1), std::forward<A2>(a2));
}
```

This example uses rvalue references as the parameters to the `factory` function. The purpose of the `std::forward` function is to forward the parameters of the factory function to the constructor of the template class.

The following example shows the `main` function that uses the revised `factory` function to create instances of the `W`, `X`, `Y`, and `Z` classes. The revised `factory` function forwards its parameters (either lvalues or rvalues) to the appropriate class constructor.

```
int main()
{
    int a = 4, b = 5;
    W* pw = factory<W>(a, b);
    X* px = factory<X>(2, b);
    Y* py = factory<Y>(a, 2);
    Z* pz = factory<Z>(2, 2);

    delete pw;
    delete px;
    delete py;
    delete pz;
}
```

## Additional Properties of Rvalue References

**You can overload a function to take an lvalue reference and an rvalue reference.**

By overloading a function to take a **const** lvalue reference or an rvalue reference, you can write code that distinguishes between non-modifiable objects (lvalues) and modifiable temporary values (rvalues). You can pass an object to a function that takes an rvalue reference unless the object is marked as **const**. The following example shows the function `f`, which is overloaded to take an lvalue reference and an rvalue reference. The `main` function calls `f` with both lvalues and an rvalue.

```
// reference-overload.cpp
// Compile with: /EHsc
#include <iostream>
using namespace std;

// A class that contains a memory resource.
class MemoryBlock
{
    // TODO: Add resources for the class here.
```

```
};

void f(const MemoryBlock&)
{
    cout << "In f(const MemoryBlock&). This version cannot modify the parameter."
    << endl;
}

void f(MemoryBlock&&)
{
    cout << "In f(MemoryBlock&&). This version can modify the parameter." << endl;
}

int main()
{
    MemoryBlock block;
    f(block);
    f(MemoryBlock());
}
```

This example produces the following output:

```
In f(const MemoryBlock&). This version cannot modify the parameter.
In f(MemoryBlock&&). This version can modify the parameter.
```

In this example, the first call to `f` passes a local variable (an lvalue) as its argument. The second call to `f` passes a temporary object as its argument. Because the temporary object cannot be referenced elsewhere in the program, the call binds to the overloaded version of `f` that takes an rvalue reference, which is free to modify the object.

**The compiler treats a named rvalue reference as an lvalue and an unnamed rvalue reference as an rvalue.**

When you write a function that takes an rvalue reference as its parameter, that parameter is treated as an lvalue in the body of the function. The compiler treats a named rvalue reference as an lvalue because a named object can be referenced by several parts of a program; it would be dangerous to allow multiple parts of a program to modify or remove resources from that object. For example, if multiple parts of a program try to transfer resources from the same object, only the first part will successfully transfer the resource.

The following example shows the function `g`, which is overloaded to take an lvalue reference and an rvalue reference. The function `f` takes an rvalue reference as its parameter (a named rvalue reference) and returns an rvalue reference (an unnamed rvalue reference). In the call to `g` from `f`, overload resolution selects the version of `g` that takes an lvalue reference because the body of `f` treats its parameter as an lvalue. In the call to `g` from `main`, overload resolution selects the version of `g` that takes an rvalue reference because `f` returns an rvalue reference.

```
// named-reference.cpp
// Compile with: /EHsc
```

```

#include <iostream>
using namespace std;

// A class that contains a memory resource.
class MemoryBlock
{
    // TODO: Add resources for the class here.
};

void g(const MemoryBlock&)
{
    cout << "In g(const MemoryBlock&)." << endl;
}

void g(MemoryBlock&&)
{
    cout << "In g(MemoryBlock&&)." << endl;
}

MemoryBlock&& f(MemoryBlock&& block)
{
    g(block);
    return move(block);
}

int main()
{
    g(f(MemoryBlock()));
}

```

This example produces the following output:

```

In g(const MemoryBlock&).
In g(MemoryBlock&&).

```

In this example, the `main` function passes an rvalue to `f`. The body of `f` treats its named parameter as an lvalue. The call from `f` to `g` binds the parameter to an lvalue reference (the first overloaded version of `g`).

- **You can cast an lvalue to an rvalue reference.**

The C++ Standard Library `std::move` function enables you to convert an object to an rvalue reference to that object. Alternatively, you can use the `static_cast` keyword to cast an lvalue to an rvalue reference, as shown in the following example:

```

// cast-reference.cpp
// Compile with: /EHsc
#include <iostream>
using namespace std;

```

```
// A class that contains a memory resource.
class MemoryBlock
{
    // TODO: Add resources for the class here.
};

void g(const MemoryBlock&)
{
    cout << "In g(const MemoryBlock&)." << endl;
}

void g(MemoryBlock&&)
{
    cout << "In g(MemoryBlock&&)." << endl;
}

int main()
{
    MemoryBlock block;
    g(block);
    g(static_cast<MemoryBlock&&>(block));
}
```

This example produces the following output:

```
In g(const MemoryBlock&).
In g(MemoryBlock&&).
```

### Function templates deduce their template argument types and then use reference collapsing rules.

It is common to write a function template that passes (or *forwards*) its parameters to another function. It is important to understand how template type deduction works for function templates that take rvalue references.

If the function argument is an rvalue, the compiler deduces the argument to be an rvalue reference. For example, if you pass an rvalue reference to an object of type `X` to a template function that takes type `T&&` as its parameter, template argument deduction deduces `T` to be `X`. Therefore, the parameter has type `X&&`. If the function argument is an lvalue or **const** lvalue, the compiler deduces its type to be an lvalue reference or **const** lvalue reference of that type.

The following example declares one structure template and then specializes it for various reference types. The `print_type_and_value` function takes an rvalue reference as its parameter and forwards it to the appropriate specialized version of the `S::print` method. The `main` function demonstrates the various ways to call the `S::print` method.

```
// template-type-deduction.cpp
// Compile with: /EHsc
#include <iostream>
```

```

#include <string>
using namespace std;

template<typename T> struct S;

// The following structures specialize S by
// lvalue reference (T&), const lvalue reference (const T&),
// rvalue reference (T&&), and const rvalue reference (const T&&).
// Each structure provides a print method that prints the type of
// the structure and its parameter.

template<typename T> struct S<T&> {
    static void print(T& t)
    {
        cout << "print<T&>: " << t << endl;
    }
};

template<typename T> struct S<const T&> {
    static void print(const T& t)
    {
        cout << "print<const T&>: " << t << endl;
    }
};

template<typename T> struct S<T&&> {
    static void print(T&& t)
    {
        cout << "print<T&&>: " << t << endl;
    }
};

template<typename T> struct S<const T&&> {
    static void print(const T&& t)
    {
        cout << "print<const T&&>: " << t << endl;
    }
};

// This function forwards its parameter to a specialized
// version of the S type.
template <typename T> void print_type_and_value(T&& t)
{
    S<T&&>::print(std::forward<T>(t));
}

// This function returns the constant string "fourth".
const string fourth() { return string("fourth"); }

int main()
{
    // The following call resolves to:
    // print_type_and_value<string&>(string& && t)
    // Which collapses to:

```

```

// print_type_and_value<string&>(string& t)
string s1("first");
print_type_and_value(s1);

// The following call resolves to:
// print_type_and_value<const string&>(const string& && t)
// Which collapses to:
// print_type_and_value<const string&>(const string& t)
const string s2("second");
print_type_and_value(s2);

// The following call resolves to:
// print_type_and_value<string&&>(string&& t)
print_type_and_value(string("third"));

// The following call resolves to:
// print_type_and_value<const string&&>(const string&& t)
print_type_and_value(fourth());
}

```

This example produces the following output:

```

print<T&>: first
print<const T&>: second
print<T&&>: third
print<const T&&>: fourth

```

To resolve each call to the `print_type_and_value` function, the compiler first performs template argument deduction. The compiler then applies reference collapsing rules when it substitutes the deduced template arguments for the parameter types. For example, passing the local variable `s1` to the `print_type_and_value` function causes the compiler to produce the following function signature:

```

print_type_and_value<string&>(string& && t)

```

The compiler uses reference collapsing rules to reduce the signature to the following:

```

print_type_and_value<string&>(string& t)

```

This version of the `print_type_and_value` function then forwards its parameter to the correct specialized version of the `S::print` method.

The following table summarizes the reference collapsing rules for template argument type deduction:

Expanded type	Collapsed type
---------------	----------------

T& &	T&
T& &&	T&
T&& &	T&
T&& &&	T&&

Template argument deduction is an important element of implementing perfect forwarding. The section [Perfect Forwarding](#), which is presented earlier in this topic, describes perfect forwarding in more detail.

## Summary

Rvalue references distinguish lvalues from rvalues. They can help you improve the performance of your applications by eliminating the need for unnecessary memory allocations and copy operations. They also enable you to write one version of a function that accepts arbitrary arguments and forwards them to another function as if the other function had been called directly.

## See also

[Expressions with Unary Operators](#)

[Lvalue Reference Declarator: &](#)

[Lvalues and Rvalues](#)

[Move Constructors and Move Assignment Operators \(C++\)](#)

[C++ Standard Library](#)



# Reference-Type Function Arguments

---

It is often more efficient to pass references, rather than large objects, to functions. This allows the compiler to pass the address of the object while maintaining the syntax that would have been used to access the object. Consider the following example that uses the `Date` structure:

```
// reference_type_function_arguments.cpp
#include <iostream>

struct Date
{
    short Month;
    short Day;
    short Year;
};

// Create a date of the form DDDYYYY (day of year, year)
// from a Date.
long DateOfYear( Date& date )
{
    static int cDaysInMonth[] = {
        31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
    };
    long dateOfYear = 0;

    // Add in days for months already elapsed.
    for ( int i = 0; i < date.Month - 1; ++i )
        dateOfYear += cDaysInMonth[i];

    // Add in days for this month.
    dateOfYear += date.Day;

    // Check for leap year.
    if ( date.Month > 2 &&
        (( date.Year % 100 != 0 || date.Year % 400 == 0 ) &&
         date.Year % 4 == 0 ) )
        dateOfYear++;

    // Add in year.
    dateOfYear *= 10000;
    dateOfYear += date.Year;

    return dateOfYear;
}

int main()
{
    Date date{ 8, 27, 2018 };
    long dateOfYear = DateOfYear(date);
}
```

```
std::cout << dateOfYear << std::endl;  
}
```

The preceding code shows that members of a structure passed by reference are accessed using the member-selection operator (.) instead of the pointer member-selection operator (->).

Although arguments passed as reference types observe the syntax of non-pointer types, they retain one important characteristic of pointer types: they are modifiable unless declared as **const**. Because the intent of the preceding code is not to modify the object `date`, a more appropriate function prototype is:

```
long DateOfYear( const Date& date );
```

This prototype guarantees that the function `DateOfYear` will not change its argument.

Any function prototyped as taking a reference type can accept an object of the same type in its place because there is a standard conversion from *typename* to *typename&*.

## See also

### [References](#)

# Reference-Type Function Returns

---

Functions can be declared to return a reference type. There are two reasons to make such a declaration:

- The information being returned is a large enough object that returning a reference is more efficient than returning a copy.
- The type of the function must be an l-value.
- The referred-to object will not go out of scope when the function returns.

Just as it can be more efficient to pass large objects *to* functions by reference, it also can be more efficient to return large objects *from* functions by reference. Reference-return protocol eliminates the necessity of copying the object to a temporary location prior to returning.

Reference-return types can also be useful when the function must evaluate to an l-value. Most overloaded operators fall into this category, particularly the assignment operator. Overloaded operators are covered in [Overloaded Operators](#).

## Example

Consider the `Point` example:

```
// refType_function_returns.cpp
// compile with: /EHsc

#include <iostream>
using namespace std;

class Point
{
public:
    // Define "accessor" functions as
    // reference types.
    unsigned& x();
    unsigned& y();
private:
    // Note that these are declared at class scope:
    unsigned obj_x;
    unsigned obj_y;
};

unsigned& Point::x()
{
    return obj_x;
}

unsigned& Point::y()
{
    return obj_y;
}
```

```

int main()
{
    Point ThePoint;
    // Use x() and y() as l-values.
    ThePoint.x() = 7;
    ThePoint.y() = 9;

    // Use x() and y() as r-values.
    cout << "x = " << ThePoint.x() << "\n"
    << "y = " << ThePoint.y() << "\n";
}

```

## Output

```

x = 7
y = 9

```

Notice that the functions `x` and `y` are declared as returning reference types. These functions can be used on either side of an assignment statement.

Note also that in `main`, `ThePoint` object remains in scope, and therefore its reference members are still alive and can be safely accessed.

Declarations of reference types must contain initializers except in the following cases:

- Explicit **extern** declaration
- Declaration of a class member
- Declaration within a class
- Declaration of an argument to a function or the return type for a function

## Caution returning address of local

If you declare an object at local scope, that object will be destroyed when the function returns. If the function returns a reference to that object, that reference will probably cause an access violation at runtime if the caller attempts to use the null reference.

```

// C4172 means Don't do this!!!
Foo& GetFoo()
{
    Foo f;
    ...
    return f;
} // f is destroyed here

```

The compiler issues a warning in this case: `warning C4172: returning address of local variable or temporary`. In simple programs it is possible that occasionally no access violation will occur if the reference is accessed by the caller before the memory location is overwritten. This is due to sheer luck. Heed the warning.

## See also

### [References](#)

# References to pointers

---

References to pointers can be declared in much the same way as references to objects. A reference to a pointer is a modifiable value that's used like a normal pointer.

## Example

This code sample shows the difference between using a pointer to a pointer and a reference to a pointer.

Functions `Add1` and `Add2` are functionally equivalent, although they're not called the same way. The difference is that `Add1` uses double indirection, but `Add2` uses the convenience of a reference to a pointer.

```
// references_to_pointers.cpp
// compile with: /EHsc

#include <iostream>
#include <string>

// C++ Standard Library namespace
using namespace std;

enum {
    sizeofBuffer = 132
};

// Define a binary tree structure.
struct BTree {
    char *szText;
    BTree *Left;
    BTree *Right;
};

// Define a pointer to the root of the tree.
BTree *btRoot = 0;

int Add1( BTree **Root, char *szToAdd );
int Add2( BTree*& Root, char *szToAdd );
void PrintTree( BTree* btRoot );

int main( int argc, char *argv[] ) {
    // Usage message
    if( argc < 2 ) {
        cerr << "Usage: " << argv[0] << " [1 | 2]" << "\n";
        cerr << "\nwhere:\n";
        cerr << "1 uses double indirection\n";
        cerr << "2 uses a reference to a pointer.\n";
        cerr << "\nInput is from stdin. Use ^Z to terminate input.\n";
        return 1;
    }
}
```

```

char *szBuf = new char[sizeOfBuffer];
if (szBuf == NULL) {
    cerr << "Out of memory!\n";
    return -1;
}

// Read a text file from the standard input device and
// build a binary tree.
while( !cin.eof() )
{
    cin.get( szBuf, sizeOfBuffer, '\n' );
    cin.get();

    if ( strlen( szBuf ) ) {
        switch ( *argv[1] ) {
            // Method 1: Use double indirection.
            case '1':
                Add1( &btRoot, szBuf );
                break;
            // Method 2: Use reference to a pointer.
            case '2':
                Add2( btRoot, szBuf );
                break;
            default:
                cerr << "Illegal value '"
                    << *argv[1]
                    << "' supplied for add method.\n"
                    << "Choose 1 or 2.\n";
                return -1;
        }
    }
}

// Display the sorted list.
PrintTree( btRoot );
}

// PrintTree: Display the binary tree in order.
void PrintTree( BTree* MybtRoot ) {
    // Traverse the left branch of the tree recursively.
    if ( MybtRoot->Left )
        PrintTree( MybtRoot->Left );

    // Print the current node.
    cout << MybtRoot->szText << "\n";

    // Traverse the right branch of the tree recursively.
    if ( MybtRoot->Right )
        PrintTree( MybtRoot->Right );
}

// Add1: Add a node to the binary tree.
//      Uses double indirection.
int Add1( BTree **Root, char *szToAdd ) {
    if ( (*Root) == 0 ) {

```

```

    (*Root) = new BTree;
    (*Root)->Left = 0;
    (*Root)->Right = 0;
    (*Root)->szText = new char[strlen( szToAdd ) + 1];
    strcpy_s((*Root)->szText, (strlen( szToAdd ) + 1), szToAdd );
    return 1;
}
else {
    if ( strcmp( (*Root)->szText, szToAdd ) > 0 )
        return Add1( &((*Root)->Left), szToAdd );
    else
        return Add1( &((*Root)->Right), szToAdd );
}
}

// Add2: Add a node to the binary tree.
//      Uses reference to pointer
int Add2( BTree*& Root, char *szToAdd ) {
    if ( Root == 0 ) {
        Root = new BTree;
        Root->Left = 0;
        Root->Right = 0;
        Root->szText = new char[strlen( szToAdd ) + 1];
        strcpy_s( Root->szText, (strlen( szToAdd ) + 1), szToAdd );
        return 1;
    }
    else {
        if ( strcmp( Root->szText, szToAdd ) > 0 )
            return Add2( Root->Left, szToAdd );
        else
            return Add2( Root->Right, szToAdd );
    }
}
}

```

Usage: references\_to\_pointers.exe [1 | 2]

where:

- 1 uses double indirection
- 2 uses a reference to a pointer.

Input is from stdin. Use ^Z to terminate input.

## See also

### [References](#)



# Pointers (C++)

---

A pointer is a variable that stores the memory address of an object. Pointers are used extensively in both C and C++ for three main purposes:

- to allocate new objects on the heap,
- to pass functions to other functions
- to iterate over elements in arrays or other data structures.

In C-style programming, *raw pointers* are used for all these scenarios. However, raw pointers are the source of many serious programming errors. Therefore, their use is strongly discouraged except where they provide a significant performance benefit and there is no ambiguity as to which pointer is the *owning pointer* that is responsible for deleting the object. Modern C++ provides *smart pointers* for allocating objects, *iterators* for traversing data structures, and *lambda expressions* for passing functions. By using these language and library facilities instead of raw pointers, you will make your program safer, easier to debug, and simpler to understand and maintain. See [Smart pointers](#), [Iterators](#), and [Lambda expressions](#) for more information.

## In this section

- [Raw pointers](#)
- [Const and volatile pointers](#)
- [new and delete operators](#)
- [Smart pointers](#)
- [How to: Create and use unique\\_ptr instances](#)
- [How to: Create and use shared\\_ptr instances](#)
- [How to: Create and use weak\\_ptr instances](#)
- [How to: Create and use CComPtr and CComQIPtr instances](#)

## See also

[Iterators](#)

[Lambda expressions](#)

# Raw pointers (C++)

---

A pointer is a type of variable that stores the address of an object in memory and is used to access that object. A *raw pointer* is a pointer whose lifetime is not controlled by an encapsulating object such as a [smart pointer](#). A raw pointer can be assigned the address of another non-pointer variable, or it can be assigned a value of `nullptr`. A pointer that has not been assigned a value contains random data.

A pointer can also be *dereferenced* to retrieve the value of the object that it points at. The *member access operator* provides access to an object's members.

```
int* p = nullptr; // declare pointer and initialize it
                // so that it doesn't store a random address
int i = 5;
p = &i; // assign pointer to address of object
int j = *p; // dereference p to retrieve the value at its address
```

A pointer can point to a typed object or to **void**. When a program allocates a new object on the [heap](#) in memory, it receives the address of that object in the form of a pointer. Such pointers are called *owning pointers*; an owning pointer (or a copy of it) must be used to explicitly delete the heap-allocated object when it is no longer needed. Failure to delete the memory results in a *memory leak* and renders that memory location unavailable to any other program on the machine. For more information, see [new and delete operators](#).

```
MyClass* mc = new MyClass(); // allocate object on the heap
mc->print(); // access class member
delete mc; // delete object (please don't forget!)
```

A pointer (if it isn't declared as **const**) can be incremented or decremented so that it points to a new location in memory. This is called *pointer arithmetic* and is used in C-style programming to iterate over elements in arrays or other data structures. A **const** pointer can't be made to point to a different memory location, and in that sense is very similar to a [reference](#). For more information, see [const and volatile pointers](#).

```
// declare a C-style string. Compiler adds terminating '\0'.
const char* str = "Hello world";

const int c = 1;
const int* pconst = &c; // declare a non-const pointer to const int
const int c2 = 2;
pconst = &c2; // OK pconst itself isn't const
const int* const pconst2 = &c;
// pconst2 = &c2; // Error! pconst2 is const.
```

On 64-bit operating systems, a pointer has a size of 64 bits; a system's pointer size determines how much addressable memory it can have. All copies of a pointer point to the same memory location. Pointers (along with references) are used extensively in C++ to pass larger objects to and from functions because it is usually far more efficient to copy an object's 64-bit address than to copy an entire object. When defining a function, specify pointer parameters as **const** unless you intend for the function to modify the object. In general, **const** references are the preferred way to pass objects to functions unless the value of the object can possibly be **nullptr**.

[Pointers to functions](#) enable functions to be passed to other functions and are used for "callbacks" in C-style programming. Modern C++ uses [lambda expressions](#) for this purpose.

## Initialization and member access

The following example shows how to declare a raw pointer and initialize it with an object allocated on the heap, and then how to use it. It also shows a few of the dangers associated with raw pointers. (Remember, this is C-style programming and not modern C++!)

```
#include <iostream>
#include <string>

class MyClass
{
public:
    int num;
    std::string name;
    void print() { std::cout << name << ":" << num << std::endl; }
};

// Accepts a MyClass pointer
void func_A(MyClass* mc)
{
    // Modify the object that mc points to.
    // All copies of the pointer will point to
    // the same modified object.
    mc->num = 3;
}

// Accepts a MyClass object
void func_B(MyClass mc)
{
    // mc here is a regular object, not a pointer.
    // Use the "." operator to access members.
    // This statement modifies only the local copy of mc.
    mc.num = 21;
    std::cout << "Local copy of mc:";
    mc.print(); // "Erika, 21"
}

int main()
{
```

```

// Use the * operator to declare a pointer type
// Use new to allocate and initialize memory
MyClass* pmc = new MyClass{ 108, "Nick" };

// Prints the memory address. Usually not what you want.
std::cout << pmc << std::endl;

// Copy the pointed-to object by dereferencing the pointer
// to access the contents of the memory location.
// mc is a separate object, allocated here on the stack
MyClass mc = *pmc;

// Declare a pointer that points to mc using the addressof operator
MyClass* pcopy = &mc;

// Use the -> operator to access the object's public members
pmc->print(); // "Nick, 108"

// Copy the pointer. Now pmc and pmc2 point to same object!
MyClass* pmc2 = pmc;

// Use copied pointer to modify the original object
pmc2->name = "Erika";
pmc->print(); // "Erika, 108"
pmc2->print(); // "Erika, 108"

// Pass the pointer to a function.
func_A(mc);
pmc->print(); // "Erika, 3"
pmc2->print(); // "Erika, 3"

// Dereference the pointer and pass a copy
// of the pointed-to object to a function
func_B(*mc);
pmc->print(); // "Erika, 3" (original not modified by function)

delete(pmc); // don't forget to give memory back to operating system!
// delete(pmc2); //crash! memory location was already deleted
}

```

## Pointer arithmetic and arrays

Pointers and arrays are closely related. When an array is passed by-value to a function, it is passed as a pointer to the first element. The following example demonstrates the following important properties of pointers and arrays:

- the `sizeof` operator returns the total size in bytes of an array
- to determine the number of elements, divide total bytes by the size of one element
- when an array is passed to a function, it *decays* to a pointer type
- the `sizeof` operator when applied to a pointer returns the pointer size, 4 bytes on x86 or 8 bytes on x64

```

#include <iostream>

void func(int arr[], int length)
{
    // returns pointer size. not useful here.
    size_t test = sizeof(arr);

    for(int i = 0; i < length; ++i)
    {
        std::cout << arr[i] << " ";
    }
}

int main()
{
    int i[5]{ 1,2,3,4,5 };
    // sizeof(i) = total bytes
    int j = sizeof(i) / sizeof(i[0]);
    func(i,j);
}

```

Certain arithmetic operations can be performed on non-const pointers to make them point to a new memory location. A pointer can be incremented and decremented using the `++`, `+=`, `-=` and `--` operators. This technique can be used in arrays and is especially useful in buffers of untyped data. A **void\*** increments by the size of a **char** (1 byte). A typed pointer increments by size of the type it points to.

The following example demonstrates how pointer arithmetic can be used to access individual pixels in a bitmap on Windows. Note the use of **new** and **delete**, and the dereference operator.

```

#include <Windows.h>
#include <fstream>

using namespace std;

int main()
{
    BITMAPINFOHEADER header;
    header.biHeight = 100; // Multiple of 4 for simplicity.
    header.biWidth = 100;
    header.biBitCount = 24;
    header.biPlanes = 1;
    header.biCompression = BI_RGB;
    header.biSize = sizeof(BITMAPINFOHEADER);

    constexpr int bufferSize = 30000;
    unsigned char* buffer = new unsigned char[bufferSize];

    BITMAPFILEHEADER bf;
}

```

```

bf.bfType = 0x4D42;
bf.bfSize = header.biSize + 14 + bufferSize;
bf.bfReserved1 = 0;
bf.bfReserved2 = 0;
bf.bfOffBits = sizeof(BITMAPFILEHEADER) + sizeof(BITMAPINFOHEADER); //54

// Create a gray square with a 2-pixel wide outline.
unsigned char* begin = &buffer[0];
unsigned char* end = &buffer[0] + bufferSize;
unsigned char* p = begin;
constexpr int pixelWidth = 3;
constexpr int borderWidth = 2;

while (p < end)
{
    // Is top or bottom edge?
    if ((p < begin + header.biWidth * pixelWidth * borderWidth)
        || (p > end - header.biWidth * pixelWidth * borderWidth)
        // Is left or right edge?
        || (p - begin) % (header.biWidth * pixelWidth) < (borderWidth *
pixelWidth)
        || (p - begin) % (header.biWidth * pixelWidth) > ((header.biWidth -
borderWidth) * pixelWidth))
    {
        *p = 0x00; // Black
    }
    else
    {
        *p = 0xC3; // Gray
    }
    p++; // Increment one byte sizeof(unsigned char).
}

ofstream wf(R"(box.bmp)", ios::out | ios::binary);

wf.write(reinterpret_cast<char*>(&bf), sizeof(bf));
wf.write(reinterpret_cast<char*>(&header), sizeof(header));
wf.write(reinterpret_cast<char*>(begin), bufferSize);

delete[] buffer; // Return memory to the OS.
wf.close();
}

```

## void\* pointers

A pointer to **void** simply points to a raw memory location. Sometimes it is necessary to use **void\*** pointers, for example when passing between C++ code and C functions.

When a typed pointer is cast to a void pointer, the contents of the memory location are not changed, but the type information is lost, so that you can't perform increment or decrement operations. A memory location can be cast, for example, from `MyClass*` to `void*` and back again to `MyClass*`. Such operations are inherently

error-prone and require great care to avoid errors. Modern C++ discourages the use of void pointers unless absolutely necessary.

```
//func.c
void func(void* data, int length)
{
    char* c = (char*)(data);

    // fill in the buffer with data
    for (int i = 0; i < length; ++i)
    {
        *c = 0x41;
        ++c;
    }
}

// main.cpp
#include <iostream>

extern "C"
{
    void func(void* data, int length);
}

class MyClass
{
public:
    int num;
    std::string name;
    void print() { std::cout << name << ":" << num << std::endl; }
};

int main()
{
    MyClass* mc = new MyClass{10, "Marian"};
    void* p = static_cast<void*>(mc);
    MyClass* mc2 = static_cast<MyClass*>(p);
    std::cout << mc2->name << std::endl; // "Marian"

    // use operator new to allocate untyped memory block
    void* pvoid = operator new(1000);
    char* pchar = static_cast<char*>(pvoid);
    for(char* c = pchar; c < pchar + 1000; ++c)
    {
        *c = 0x00;
    }
    func(pvoid, 1000);
    char ch = static_cast<char*>(pvoid)[0];
    std::cout << ch << std::endl; // 'A'
    operator delete(p);
}
```

## Pointers to functions

In C-style programming, function pointers are used primarily to pass functions to other functions. In this scenario, the caller can customize the behavior of a function without modifying it. In modern C++, [lambda expressions](#) provide the same capability with greater type safety and other advantages.

A function pointer declaration specifies the signature that the pointed-to function must have:

```
// Declare pointer to any function that...

// ...accepts a string and returns a string
string (*g)(string a);

// has no return value and no parameters
void (*x)();

// ...returns an int and takes three parameters
// of the specified types
int (*i)(int i, string s, double d);
```

The following example shows a function `combine` that takes as a parameter any function that accepts a `std::string` and returns a `std::string`. Depending on the function that is passed to `combine` it will either prepend or append a string.

```
#include <iostream>
#include <string>

using namespace std;

string base {"hello world"};

string append(string s)
{
    return base.append(" ").append(s);
}

string prepend(string s)
{
    return s.append(" ").append(base);
}

string combine(string s, string(*g)(string a))
{
    return (*g)(s);
}

int main()
{
```



```
cout << combine("from MSVC", append) << "\n";  
cout << combine("Good morning and", prepend) << "\n";  
}
```

## See also

[Smart pointers Indirection Operator: \\*](#)

[Address-of Operator: &](#)

[Welcome back to C++](#)

# const and volatile pointers

---

The `const` and `volatile` keywords change how pointers are treated. The **const** keyword specifies that the pointer cannot be modified after initialization; the pointer is protected from modification thereafter.

The **volatile** keyword specifies that the value associated with the name that follows can be modified by actions other than those in the user application. Therefore, the **volatile** keyword is useful for declaring objects in shared memory that can be accessed by multiple processes or global data areas used for communication with interrupt service routines.

When a name is declared as **volatile**, the compiler reloads the value from memory each time it is accessed by the program. This dramatically reduces the possible optimizations. However, when the state of an object can change unexpectedly, it is the only way to ensure predictable program performance.

To declare the object pointed to by the pointer as **const** or **volatile**, use a declaration of the form:

```
const char *cpch;  
volatile char *vpch;
```

To declare the value of the pointer — that is, the actual address stored in the pointer — as **const** or **volatile**, use a declaration of the form:

```
char * const pchc;  
char * volatile pchv;
```

The C++ language prevents assignments that would allow modification of an object or pointer declared as **const**. Such assignments would remove the information that the object or pointer was declared with, thereby violating the intent of the original declaration. Consider the following declarations:

```
const char cch = 'A';  
char ch = 'B';
```

Given the preceding declarations of two objects (`cch`, of type **const char**, and `ch`, of type **char**), the following declaration/initializations are valid:

```
const char *pch1 = &cch;  
const char *const pch4 = &cch;  
const char *pch5 = &ch;  
char *pch6 = &ch;  
char *const pch7 = &ch;  
const char *const pch8 = &ch;
```

The following declaration/initializations are erroneous.

```
char *pch2 = &cch;    // Error
char *const pch3 = &cch;    // Error
```

The declaration of `pch2` declares a pointer through which a constant object might be modified and is therefore disallowed. The declaration of `pch3` specifies that the pointer is constant, not the object; the declaration is disallowed for the same reason the `pch2` declaration is disallowed.

The following eight assignments show assigning through pointer and changing of pointer value for the preceding declarations; for now, assume that the initialization was correct for `pch1` through `pch8`.

```
*pch1 = 'A';    // Error: object declared const
pch1 = &ch;     // OK: pointer not declared const
*pch2 = 'A';    // OK: normal pointer
pch2 = &ch;     // OK: normal pointer
*pch3 = 'A';    // OK: object not declared const
pch3 = &ch;     // Error: pointer declared const
*pch4 = 'A';    // Error: object declared const
pch4 = &ch;     // Error: pointer declared const
```

Pointers declared as **volatile**, or as a mixture of **const** and **volatile**, obey the same rules.

Pointers to **const** objects are often used in function declarations as follows:

```
errno_t strcpy_s( char *strDestination, size_t numberOfElements, const char
*strSource );
```

The preceding statement declares a function, `strcpy_s`, where two of the three arguments are of type pointer to **char**. Because the arguments are passed by reference and not by value, the function would be free to modify both `strDestination` and `strSource` if `strSource` were not declared as **const**. The declaration of `strSource` as **const** assures the caller that `strSource` cannot be changed by the called function.

[!NOTE] Because there is a standard conversion from *typename \** to **const** *typename \**, it is legal to pass an argument of type `char *` to `strcpy_s`. However, the reverse is not true; no implicit conversion exists to remove the **const** attribute from an object or pointer.

A **const** pointer of a given type can be assigned to a pointer of the same type. However, a pointer that is not **const** cannot be assigned to a **const** pointer. The following code shows correct and incorrect assignments:

```
// const_pointer.cpp
int *const cpObject = 0;
int *pObject;

int main() {
```

```
pObject = cpObject;
cpObject = pObject;    // C3892
}
```

The following sample shows how to declare an object as const if you have a pointer to a pointer to an object.

```
// const_pointer2.cpp
struct X {
    X(int i) : m_i(i) { }
    int m_i;
};

int main() {
    // correct
    const X cx(10);
    const X * pcx = &cx;
    const X ** ppcx = &pcx;

    // also correct
    X const cx2(20);
    X const * pcx2 = &cx2;
    X const ** ppcx2 = &pcx2;
}
```

## See also

[Pointers Raw pointers](#)

# new and delete operators

---

C++ supports dynamic allocation and deallocation of objects using the [new](#) and [delete](#) operators. These operators allocate memory for objects from a pool called the free store. The **new** operator calls the special function [operator new](#), and the **delete** operator calls the special function [operator delete](#).

The **new** function in the C++ Standard Library supports the behavior specified in the C++ standard, which is to throw a `std::bad_alloc` exception if the memory allocation fails. If you still want the non-throwing version of **new**, link your program with `nothrownew.obj`. However, when you link with `nothrownew.obj`, the default **operator new** in the C++ Standard Library no longer functions.

For a list of the library files that comprise the C Runtime Library and the C++ Standard Library, see [CRT Library Features](#).

## The new operator

When a statement such as the following is encountered in a program, it translates into a call to the function **operator new**:

```
char *pch = new char[BUFFER_SIZE];
```

If the request is for zero bytes of storage, **operator new** returns a pointer to a distinct object (that is, repeated calls to **operator new** return different pointers). If there is insufficient memory for the allocation request, **operator new** throws a `std::bad_alloc` exception, or returns `nullptr` if you have linked in non-throwing **operator new** support.

You can write a routine that attempts to free memory and retry the allocation; see [\\_set\\_new\\_handler](#) for more information. For more details on the recovery scheme, see the Handling insufficient memory section of this topic.

The two scopes for **operator new** functions are described in the following table.

Scope for operator new functions

Operator	Scope
<b>::operator new</b>	Global
<i>class-name</i> <b>::operator new</b>	Class

The first argument to **operator new** must be of type `size_t` (a type defined in `<stddef.h>`), and the return type is always `void *`.

The global **operator new** function is called when the **new** operator is used to allocate objects of built-in types, objects of class type that do not contain user-defined **operator new** functions, and arrays of any type. When the **new** operator is used to allocate objects of a class type where an **operator new** is defined, that class's **operator new** is called.

An **operator new** function defined for a class is a static member function (which cannot, therefore, be virtual) that hides the global **operator new** function for objects of that class type. Consider the case where **new** is used to allocate and set memory to a given value:

```
#include <malloc.h>
#include <memory.h>

class Blanks
{
public:
    Blanks(){}
    void *operator new( size_t stAllocateBlock, char chInit );
};
void *Blanks::operator new( size_t stAllocateBlock, char chInit )
{
    void *pvTemp = malloc( stAllocateBlock );
    if( pvTemp != 0 )
        memset( pvTemp, chInit, stAllocateBlock );
    return pvTemp;
}
// For discrete objects of type Blanks, the global operator new function
// is hidden. Therefore, the following code allocates an object of type
// Blanks and initializes it to 0xa5
int main()
{
    Blanks *a5 = new(0xa5) Blanks;
    return a5 != 0;
}
```

The argument supplied in parentheses to **new** is passed to **Blanks::operator new** as the **chInit** argument. However, the global **operator new** function is hidden, causing code such as the following to generate an error:

```
Blanks *SomeBlanks = new Blanks;
```

The compiler supports member array **new** and **delete** operators in a class declaration. For example:

```
class MyClass
{
public:
    void * operator new[] (size_t)
    {
        return 0;
    }
    void operator delete[] (void*)
    {
    }
};
```

```
int main()
{
    MyClass *pMyClass = new MyClass[5];
    delete [] pMyClass;
}
```

## Handling insufficient memory

Testing for failed memory allocation can be done as shown here:

```
#include <iostream>
using namespace std;
#define BIG_NUMBER 100000000
int main() {
    int *pI = new int[BIG_NUMBER];
    if( pI == 0x0 ) {
        cout << "Insufficient memory" << endl;
        return -1;
    }
}
```

There is another way to handle failed memory allocation requests. Write a custom recovery routine to handle such a failure, then register your function by calling the [\\_set\\_new\\_handler](#) run-time function.

## The delete operator

Memory that is dynamically allocated using the **new** operator can be freed using the **delete** operator. The delete operator calls the **operator delete** function, which frees memory back to the available pool. Using the **delete** operator also causes the class destructor (if there is one) to be called.

There are global and class-scoped **operator delete** functions. Only one **operator delete** function can be defined for a given class; if defined, it hides the global **operator delete** function. The global **operator delete** function is always called for arrays of any type.

The global **operator delete** function. Two forms exist for the global **operator delete** and class-member **operator delete** functions:

```
void operator delete( void * );
void operator delete( void *, size_t );
```

Only one of the preceding two forms can be present for a given class. The first form takes a single argument of type `void *`, which contains a pointer to the object to deallocate. The second form—sized deallocation—takes two arguments, the first of which is a pointer to the memory block to deallocate and the second of which is the number of bytes to deallocate. The return type of both forms is **void** (**operator delete** cannot return a value).

The intent of the second form is to speed up searching for the correct size category of the object to be deleted, which is often not stored near the allocation itself and likely uncached. The second form is useful when an **operator delete** function from a base class is used to delete an object of a derived class.

The **operator delete** function is static; therefore, it cannot be virtual. The **operator delete** function obeys access control, as described in [Member-Access Control](#).

The following example shows user-defined **operator new** and **operator delete** functions designed to log allocations and deallocations of memory:

```
#include <iostream>
using namespace std;

int fLogMemory = 0;      // Perform logging (0=no; nonzero=yes)?
int cBlocksAllocated = 0; // Count of blocks allocated.

// User-defined operator new.
void *operator new( size_t stAllocateBlock ) {
    static int fInOpNew = 0;    // Guard flag.

    if ( fLogMemory && !fInOpNew ) {
        fInOpNew = 1;
        clog << "Memory block " << ++cBlocksAllocated
              << " allocated for " << stAllocateBlock
              << " bytes\n";
        fInOpNew = 0;
    }
    return malloc( stAllocateBlock );
}

// User-defined operator delete.
void operator delete( void *pvMem ) {
    static int fInOpDelete = 0; // Guard flag.
    if ( fLogMemory && !fInOpDelete ) {
        fInOpDelete = 1;
        clog << "Memory block " << cBlocksAllocated--
              << " deallocated\n";
        fInOpDelete = 0;
    }

    free( pvMem );
}

int main( int argc, char *argv[] ) {
    fLogMemory = 1;    // Turn logging on
    if( argc > 1 )
        for( int i = 0; i < atoi( argv[1] ); ++i ) {
            char *pMem = new char[10];
            delete[] pMem;
        }
    fLogMemory = 0;    // Turn logging off.
```



```
    return cBlocksAllocated;
}
```

The preceding code can be used to detect "memory leakage" — that is, memory that is allocated on the free store but never freed. To perform this detection, the global **new** and **delete** operators are redefined to count allocation and deallocation of memory.

The compiler supports member array **new** and **delete** operators in a class declaration. For example:

```
// spec1_the_operator_delete_function2.cpp
// compile with: /c
class X {
public:
    void * operator new[] (size_t) {
        return 0;
    }
    void operator delete[] (void*) {}
};

void f() {
    X *pX = new X[5];
    delete [] pX;
}
```

# Smart pointers (Modern C++)

---

In modern C++ programming, the Standard Library includes *smart pointers*, which are used to help ensure that programs are free of memory and resource leaks and are exception-safe.

## Uses for smart pointers

Smart pointers are defined in the `std` namespace in the `<memory>` header file. They are crucial to the [RAII](#) or *Resource Acquisition Is Initialization* programming idiom. The main goal of this idiom is to ensure that resource acquisition occurs at the same time that the object is initialized, so that all resources for the object are created and made ready in one line of code. In practical terms, the main principle of RAII is to give ownership of any heap-allocated resource—for example, dynamically-allocated memory or system object handles—to a stack-allocated object whose destructor contains the code to delete or free the resource and also any associated cleanup code.

In most cases, when you initialize a raw pointer or resource handle to point to an actual resource, pass the pointer to a smart pointer immediately. In modern C++, raw pointers are only used in small code blocks of limited scope, loops, or helper functions where performance is critical and there is no chance of confusion about ownership.

The following example compares a raw pointer declaration to a smart pointer declaration.

[!code-cppsmart\_pointers\_intro#1]

As shown in the example, a smart pointer is a class template that you declare on the stack, and initialize by using a raw pointer that points to a heap-allocated object. After the smart pointer is initialized, it owns the raw pointer. This means that the smart pointer is responsible for deleting the memory that the raw pointer specifies. The smart pointer destructor contains the call to delete, and because the smart pointer is declared on the stack, its destructor is invoked when the smart pointer goes out of scope, even if an exception is thrown somewhere further up the stack.

Access the encapsulated pointer by using the familiar pointer operators, `->` and `*`, which the smart pointer class overloads to return the encapsulated raw pointer.

The C++ smart pointer idiom resembles object creation in languages such as C#: you create the object and then let the system take care of deleting it at the correct time. The difference is that no separate garbage collector runs in the background; memory is managed through the standard C++ scoping rules so that the runtime environment is faster and more efficient.

[!IMPORTANT] Always create smart pointers on a separate line of code, never in a parameter list, so that a subtle resource leak won't occur due to certain parameter list allocation rules.

The following example shows how a `unique_ptr` smart pointer type from the C++ Standard Library could be used to encapsulate a pointer to a large object.

[!code-cppsmart\_pointers\_intro#2]

The example demonstrates the following essential steps for using smart pointers.

1. Declare the smart pointer as an automatic (local) variable. (Do not use the **new** or **malloc** expression on the smart pointer itself.)
2. In the type parameter, specify the pointed-to type of the encapsulated pointer.
3. Pass a raw pointer to a **new**-ed object in the smart pointer constructor. (Some utility functions or smart pointer constructors do this for you.)
4. Use the overloaded **->** and **\*** operators to access the object.
5. Let the smart pointer delete the object.

Smart pointers are designed to be as efficient as possible both in terms of memory and performance. For example, the only data member in **unique\_ptr** is the encapsulated pointer. This means that **unique\_ptr** is exactly the same size as that pointer, either four bytes or eight bytes. Accessing the encapsulated pointer by using the smart pointer overloaded **\*** and **->** operators is not significantly slower than accessing the raw pointers directly.

Smart pointers have their own member functions, which are accessed by using "dot" notation. For example, some C++ Standard Library smart pointers have a reset member function that releases ownership of the pointer. This is useful when you want to free the memory owned by the smart pointer before the smart pointer goes out of scope, as shown in the following example.

[!code-cppsmart\_pointers\_intro#3]

Smart pointers usually provide a way to access their raw pointer directly. C++ Standard Library smart pointers have a **get** member function for this purpose, and **CComPtr** has a public **p** class member. By providing direct access to the underlying pointer, you can use the smart pointer to manage memory in your own code and still pass the raw pointer to code that does not support smart pointers.

[!code-cppsmart\_pointers\_intro#4]

## Kinds of smart pointers

The following section summarizes the different kinds of smart pointers that are available in the Windows programming environment and describes when to use them.

### C++ Standard Library smart pointers

Use these smart pointers as a first choice for encapsulating pointers to plain old C++ objects (POCO).

- **unique\_ptr**  
Allows exactly one owner of the underlying pointer. Use as the default choice for POCO unless you know for certain that you require a **shared\_ptr**. Can be moved to a new owner, but not copied or shared. Replaces **auto\_ptr**, which is deprecated. Compare to **boost::scoped\_ptr**. **unique\_ptr** is small and efficient; the size is one pointer and it supports rvalue references for fast insertion and retrieval from C++ Standard Library collections. Header file: **<memory>**. For more information, see [How to: Create and Use unique\\_ptr Instances](#) and [unique\\_ptr Class](#).
- **shared\_ptr**  
Reference-counted smart pointer. Use when you want to assign one raw pointer to multiple owners, for

example, when you return a copy of a pointer from a container but want to keep the original. The raw pointer is not deleted until all `shared_ptr` owners have gone out of scope or have otherwise given up ownership. The size is two pointers; one for the object and one for the shared control block that contains the reference count. Header file: `<memory>`. For more information, see [How to: Create and Use `shared\_ptr` Instances](#) and [`shared\_ptr` Class](#).

- `weak_ptr`

Special-case smart pointer for use in conjunction with `shared_ptr`. A `weak_ptr` provides access to an object that is owned by one or more `shared_ptr` instances, but does not participate in reference counting. Use when you want to observe an object, but do not require it to remain alive. Required in some cases to break circular references between `shared_ptr` instances. Header file: `<memory>`. For more information, see [How to: Create and Use `weak\_ptr` Instances](#) and [`weak\_ptr` Class](#).

## Smart pointers for COM objects (classic Windows programming)

When you work with COM objects, wrap the interface pointers in an appropriate smart pointer type. The Active Template Library (ATL) defines several smart pointers for various purposes. You can also use the `_com_ptr_t` smart pointer type, which the compiler uses when it creates wrapper classes from .tlb files. It's the best choice when you do not want to include the ATL header files.

### `CComPtr` Class

Use this unless you cannot use ATL. Performs reference counting by using the `AddRef` and `Release` methods. For more information, see [How to: Create and Use `CComPtr` and `CComQIPtr` Instances](#).

### `CComQIPtr` Class

Resembles `CComPtr` but also provides simplified syntax for calling `QueryInterface` on COM objects. For more information, see [How to: Create and Use `CComPtr` and `CComQIPtr` Instances](#).

### `CComHeapPtr` Class

Smart pointer to objects that use `CoTaskMemFree` to free memory.

### `CComGIPtr` Class

Smart pointer for interfaces that are obtained from the global interface table (GIT).

### `_com_ptr_t` Class

Resembles `CComQIPtr` in functionality but does not depend on ATL headers.

## ATL smart pointers for POCO objects

In addition to smart pointers for COM objects, ATL also defines smart pointers, and collections of smart pointers, for plain old C++ objects (POCO). In classic Windows programming, these types are useful alternatives to the C++ Standard Library collections, especially when code portability is not required or when you do not want to mix the programming models of the C++ Standard Library and ATL.

### `CAutoPtr` Class

Smart pointer that enforces unique ownership by transferring ownership on copy. Comparable to the deprecated `std::auto_ptr` Class.

### `CHeapPtr` Class

Smart pointer for objects that are allocated by using the C `malloc` function.

### [CAutoVectorPtr Class](#)

Smart pointer for arrays that are allocated by using `new[]`.

### [CAutoPtrArray Class](#)

Class that encapsulates an array of `CAutoPtr` elements.

### [CAutoPtrList Class](#)

Class that encapsulates methods for manipulating a list of `CAutoPtr` nodes.

## See also

[Pointers](#)

[C++ Language Reference](#)

[C++ Standard Library](#)

# How to: Create and use `unique_ptr` instances

---

A `unique_ptr` does not share its pointer. It cannot be copied to another `unique_ptr`, passed by value to a function, or used in any C++ Standard Library algorithm that requires copies to be made. A `unique_ptr` can only be moved. This means that the ownership of the memory resource is transferred to another `unique_ptr` and the original `unique_ptr` no longer owns it. We recommend that you restrict an object to one owner, because multiple ownership adds complexity to the program logic. Therefore, when you need a smart pointer for a plain C++ object, use `unique_ptr`, and when you construct a `unique_ptr`, use the `make_unique` helper function.

The following diagram illustrates the transfer of ownership between two `unique_ptr` instances.

 Moving the ownership of a `unique_ptr`

`unique_ptr` is defined in the `<memory>` header in the C++ Standard Library. It is exactly as efficient as a raw pointer and can be used in C++ Standard Library containers. The addition of `unique_ptr` instances to C++ Standard Library containers is efficient because the move constructor of the `unique_ptr` eliminates the need for a copy operation.

## Example 1

The following example shows how to create `unique_ptr` instances and pass them between functions.

[code-cppstl\_smart\_pointers#210]

These examples demonstrate this basic characteristic of `unique_ptr`: it can be moved, but not copied. "Moving" transfers ownership to a new `unique_ptr` and resets the old `unique_ptr`.

## Example 2

The following example shows how to create `unique_ptr` instances and use them in a vector.

[code-cppstl\_smart\_pointers#211]

In the range for loop, notice that the `unique_ptr` is passed by reference. If you try to pass by value here, the compiler will throw an error because the `unique_ptr` copy constructor is deleted.

## Example 3

The following example shows how to initialize a `unique_ptr` that is a class member.

[code-cppstl\_smart\_pointers#212]

## Example 4

You can use `make_unique` to create a `unique_ptr` to an array, but you cannot use `make_unique` to initialize the array elements.

[code-cppstl\_smart\_pointers#213]

For more examples, see [make\\_unique](#).

## See also

[Smart Pointers \(Modern C++\)](#)


[make\\_unique](#)

# How to: Create and Use `shared_ptr` instances

---

The `shared_ptr` type is a smart pointer in the C++ standard library that is designed for scenarios in which more than one owner might have to manage the lifetime of the object in memory. After you initialize a `shared_ptr` you can copy it, pass it by value in function arguments, and assign it to other `shared_ptr` instances. All the instances point to the same object, and share access to one "control block" that increments and decrements the reference count whenever a new `shared_ptr` is added, goes out of scope, or is reset. When the reference count reaches zero, the control block deletes the memory resource and itself.

The following illustration shows several `shared_ptr` instances that point to one memory location.

 Shared pointer diagram

## Example setup

The examples that follow all assume that you've included the required headers and declared the required types, as shown here:

```
// shared_ptr-examples.cpp
// The following examples assume these declarations:
#include <algorithm>
#include <iostream>
#include <memory>
#include <string>
#include <vector>

struct MediaAsset
{
    virtual ~MediaAsset() = default; // make it polymorphic
};

struct Song : public MediaAsset
{
    std::wstring artist;
    std::wstring title;
    Song(const std::wstring& artist_, const std::wstring& title_) :
        artist{ artist_ }, title{ title_ } {}
};

struct Photo : public MediaAsset
{
    std::wstring date;
    std::wstring location;
    std::wstring subject;
    Photo(
        const std::wstring& date_,
        const std::wstring& location_,
        const std::wstring& subject_) :
        date{ date_ }, location{ location_ }, subject{ subject_ } {}
};
```



```
};

using namespace std;

int main()
{
    // The examples go here, in order:
    // Example 1
    // Example 2
    // Example 3
    // Example 4
    // Example 6
}
```

## Example 1

Whenever possible, use the `make_shared` function to create a `shared_ptr` when the memory resource is created for the first time. `make_shared` is exception-safe. It uses the same call to allocate the memory for the control block and the resource, which reduces the construction overhead. If you don't use `make_shared`, then you have to use an explicit `new` expression to create the object before you pass it to the `shared_ptr` constructor. The following example shows various ways to declare and initialize a `shared_ptr` together with a new object.

[!code-cppstl\_smart\_pointers#1]

## Example 2

The following example shows how to declare and initialize `shared_ptr` instances that take on shared ownership of an object that has already been allocated by another `shared_ptr`. Assume that `sp2` is an initialized `shared_ptr`.

[!code-cppstl\_smart\_pointers#2]

## Example 3

`shared_ptr` is also helpful in C++ Standard Library containers when you're using algorithms that copy elements. You can wrap elements in a `shared_ptr`, and then copy it into other containers with the understanding that the underlying memory is valid as long as you need it, and no longer. The following example shows how to use the `remove_copy_if` algorithm on `shared_ptr` instances in a vector.

[!code-cppstl\_smart\_pointers#4]

## Example 4

You can use `dynamic_pointer_cast`, `static_pointer_cast`, and `const_pointer_cast` to cast a `shared_ptr`. These functions resemble the `dynamic_cast`, `static_cast`, and `const_cast` operators. The following example shows how to test the derived type of each element in a vector of `shared_ptr` of base classes, and then copy the elements and display information about them.

[!code-cppstl\_smart\_pointers#5]

## Example 5

You can pass a `shared_ptr` to another function in the following ways:

- Pass the `shared_ptr` by value. This invokes the copy constructor, increments the reference count, and makes the callee an owner. There's a small amount of overhead in this operation, which may be significant depending on how many `shared_ptr` objects you're passing. Use this option when the implied or explicit code contract between the caller and callee requires that the callee be an owner.
- Pass the `shared_ptr` by reference or const reference. In this case, the reference count isn't incremented, and the callee can access the pointer as long as the caller doesn't go out of scope. Or, the callee can decide to create a `shared_ptr` based on the reference, and become a shared owner. Use this option when the caller has no knowledge of the callee, or when you must pass a `shared_ptr` and want to avoid the copy operation for performance reasons.
- Pass the underlying pointer or a reference to the underlying object. This enables the callee to use the object, but doesn't enable it to share ownership or extend the lifetime. If the callee creates a `shared_ptr` from the raw pointer, the new `shared_ptr` is independent from the original, and doesn't control the underlying resource. Use this option when the contract between the caller and callee clearly specifies that the caller retains ownership of the `shared_ptr` lifetime.
- When you're deciding how to pass a `shared_ptr`, determine whether the callee has to share ownership of the underlying resource. An "owner" is an object or function that can keep the underlying resource alive for as long as it needs it. If the caller has to guarantee that the callee can extend the life of the pointer beyond its (the function's) lifetime, use the first option. If you don't care whether the callee extends the lifetime, then pass by reference and let the callee copy it or not.
- If you have to give a helper function access to the underlying pointer, and you know that the helper function will just use the pointer and return before the calling function returns, then that function doesn't have to share ownership of the underlying pointer. It just has to access the pointer within the lifetime of the caller's `shared_ptr`. In this case, it's safe to pass the `shared_ptr` by reference, or pass the raw pointer or a reference to the underlying object. Passing this way provides a small performance benefit, and may also help you express your programming intent.
- Sometimes, for example in a `std::vector<shared_ptr<T>>`, you may have to pass each `shared_ptr` to a lambda expression body or named function object. If the lambda or function doesn't store the pointer, then pass the `shared_ptr` by reference to avoid invoking the copy constructor for each element.

## Example 6

The following example shows how `shared_ptr` overloads various comparison operators to enable pointer comparisons on the memory that is owned by the `shared_ptr` instances.

[!code-cppstl\_smart\_pointers#3]

## See also

[Smart Pointers \(Modern C++\)](#)

# How to: Create and use `weak_ptr` instances

---

Sometimes an object must store a way to access the underlying object of a `shared_ptr` without causing the reference count to be incremented. Typically, this situation occurs when you have cyclic references between `shared_ptr` instances.

The best design is to avoid shared ownership of pointers whenever you can. However, if you must have shared ownership of `shared_ptr` instances, avoid cyclic references between them. When cyclic references are unavoidable, or even preferable for some reason, use `weak_ptr` to give one or more of the owners a weak reference to another `shared_ptr`. By using a `weak_ptr`, you can create a `shared_ptr` that joins to an existing set of related instances, but only if the underlying memory resource is still valid. A `weak_ptr` itself does not participate in the reference counting, and therefore, it cannot prevent the reference count from going to zero. However, you can use a `weak_ptr` to try to obtain a new copy of the `shared_ptr` with which it was initialized. If the memory has already been deleted, the `weak_ptr`'s `bool` operator returns `false`. If the memory is still valid, the new shared pointer increments the reference count and guarantees that the memory will be valid as long as the `shared_ptr` variable stays in scope.

## Example

The following code example shows a case where `weak_ptr` is used to ensure proper deletion of objects that have circular dependencies. As you examine the example, assume that it was created only after alternative solutions were considered. The `Controller` objects represent some aspect of a machine process, and they operate independently. Each controller must be able to query the status of the other controllers at any time, and each one contains a private `vector<weak_ptr<Controller>>` for this purpose. Each vector contains a circular reference, and therefore, `weak_ptr` instances are used instead of `shared_ptr`.

[[code-cppstl\\_smart\\_pointers#222](#)]

```
Creating Controller0
Creating Controller1
Creating Controller2
Creating Controller3
Creating Controller4
push_back to v[0]: 1
push_back to v[0]: 2
push_back to v[0]: 3
push_back to v[0]: 4
push_back to v[1]: 0
push_back to v[1]: 2
push_back to v[1]: 3
push_back to v[1]: 4
push_back to v[2]: 0
push_back to v[2]: 1
push_back to v[2]: 3
push_back to v[2]: 4
push_back to v[3]: 0
push_back to v[3]: 1
push_back to v[3]: 2
```

```
push_back to v[3]: 4
push_back to v[4]: 0
push_back to v[4]: 1
push_back to v[4]: 2
push_back to v[4]: 3
use_count = 1
Status of 1 = On
Status of 2 = On
Status of 3 = On
Status of 4 = On
use_count = 1
Status of 0 = On
Status of 2 = On
Status of 3 = On
Status of 4 = On
use_count = 1
Status of 0 = On
Status of 1 = On
Status of 3 = On
Status of 4 = On
use_count = 1
Status of 0 = On
Status of 1 = On
Status of 2 = On
Status of 4 = On
use_count = 1
Status of 0 = On
Status of 1 = On
Status of 2 = On
Status of 3 = On
Destroying Controller0
Destroying Controller1
Destroying Controller2
Destroying Controller3
Destroying Controller4
Press any key
```

As an experiment, modify the vector `others` to be a `vector<shared_ptr<Controller>>`, and then in the output, notice that no destructors are invoked when `TestRun` returns.

## See also

[Smart Pointers \(Modern C++\)](#)

# How to: Create and use CComPtr and CComQIPtr instances

---

In classic Windows programming, libraries are often implemented as COM objects (or more precisely, as COM servers). Many Windows operating system components are implemented as COM servers, and many contributors provide libraries in this form. For information about the basics of COM, see [Component Object Model \(COM\)](#).

When you instantiate a Component Object Model (COM) object, store the interface pointer in a COM smart pointer, which performs the reference counting by using calls to `AddRef` and `Release` in the destructor. If you are using the Active Template Library (ATL) or the Microsoft Foundation Class Library (MFC), then use the `CComPtr` smart pointer. If you are not using ATL or MFC, then use `_com_ptr_t`. Because there is no COM equivalent to `std::unique_ptr`, use these smart pointers for both single-owner and multiple-owner scenarios. Both `CComPtr` and `CComQIPtr` support move operations that have rvalue references.

## Example

The following example shows how to use `CComPtr` to instantiate a COM object and obtain pointers to its interfaces. Notice that the `CComPtr::CoCreateInstance` member function is used to create the COM object, instead of the Win32 function that has the same name.

[!code-cppCOM\_smart\_pointers#01]

`CComPtr` and its relatives are part of the ATL and are defined in `<atlcomcli.h>`. `_com_ptr_t` is declared in `<comip.h>`. The compiler creates specializations of `_com_ptr_t` when it generates wrapper classes for type libraries.

## Example

ATL also provides `CComQIPtr`, which has a simpler syntax for querying a COM object to retrieve an additional interface. However, we recommend `CComPtr` because it does everything that `CComQIPtr` can do and is semantically more consistent with raw COM interface pointers. If you use a `CComPtr` to query for an interface, the new interface pointer is placed in an out parameter. If the call fails, an HRESULT is returned, which is the typical COM pattern. With `CComQIPtr`, the return value is the pointer itself, and if the call fails, the internal HRESULT return value cannot be accessed. The following two lines show how the error handling mechanisms in `CComPtr` and `CComQIPtr` differ.

[!code-cppCOM\_smart\_pointers#02]

## Example

`CComPtr` provides a specialization for `IDispatch` that enables it to store pointers to COM automation components and invoke the methods on the interface by using late binding. `CComDispatchDriver` is a typedef for `CComQIPtr<IDispatch, &IIDIDispatch>`, which is implicitly convertible to `CComPtr<IDispatch>`. Therefore, when any of these three names appears in code, it is equivalent to `CComPtr<IDispatch>`. The following example shows how to obtain a pointer to the Microsoft Word object model by using a `CComPtr<IDispatch>`.

[!code-cppCOM\_smart\_pointers#03]

## See also

[Smart Pointers \(Modern C++\)](#)

# Pimpl For Compile-Time Encapsulation (Modern C++)

---

The *pimpl idiom* is a modern C++ technique to hide implementation, to minimize coupling, and to separate interfaces. Pimpl is short for "pointer to implementation." You may already be familiar with the concept but know it by other names like Cheshire Cat or Compiler Firewall idiom.

## Why use pimpl?

Here's how the pimpl idiom can improve the software development lifecycle:

- Minimization of compilation dependencies.
- Separation of interface and implementation.
- Portability.

## Pimpl header

```
// my_class.h
class my_class {
    // ... all public and protected stuff goes here ...
private:
    class impl; unique_ptr<impl> pimpl; // opaque type here
};
```

The pimpl idiom avoids rebuild cascades and brittle object layouts. It's well suited for (transitively) popular types.

## Pimpl implementation

Define the `impl` class in the `.cpp` file.

```
// my_class.cpp
class my_class::impl { // defined privately here
    // ... all private data and functions: all of these
    //      can now change without recompiling callers ...
};
my_class::my_class(): pimpl( new impl )
{
    // ... set impl values ...
}
```

## Best practices

Consider whether to add support for non-throwing swap specialization.

## See also

[Welcome back to C++](#)

[C++ Language Reference](#)

[C++ Standard Library](#)



# Exception handling in MSVC

---

An exception is an error condition, possibly outside the program's control, that prevents the program from continuing along its regular execution path. Certain operations, including object creation, file input/output, and function calls made from other modules, are all potential sources of exceptions even when your program is running correctly. Robust code anticipates and handles exceptions. To detect logic errors, use assertions rather than exceptions (see [Using Assertions](#)).

## Kinds of exceptions

The Microsoft C++ compiler (MSVC) supports three kinds of exception handling:

- [C++ exception handling](#)

For most C++ programs, you should use C++ exception handling, which is type-safe and ensures that object destructors are invoked during stack unwinding.

- [Structured exception handling](#)

Windows supplies its own exception mechanism, called SEH. It is not recommended for C++ or MFC programming. Use SEH only in non-MFC C programs.

- [MFC exceptions](#)

Use the `/EH` compiler option to specify the type of exception handling to use in a project; C++ exception handling is the default. Do not mix the error handling mechanisms; for example, do not use C++ exceptions with structured exception handling. Using C++ exception handling makes your code more portable, and it allows you to handle exceptions of any type. For more information about the drawbacks of structured exception handling, see [Structured Exception Handling](#). For advice about mixing MFC macros and C++ exceptions, see [Exceptions: Using MFC Macros and C++ Exceptions](#).

## In this section

- [Modern C++ best practices for exceptions and error handling](#)
- [How to design for exception safety](#)
- [How to interface between exceptional and non-exceptional code](#)
- [The try, catch, and throw Statements](#)
- [How Catch Blocks are Evaluated](#)
- [Exceptions and Stack Unwinding](#)
- [Exception Specifications](#)
- [noexcept](#)
- [Unhandled C++ Exceptions](#)

- [Mixing C \(Structured\) and C++ Exceptions](#)
- [Structured Exception Handling \(SEH\) \(C/C++\)](#)

## See also

[C++ Language Reference](#)

[x64 exception handling](#)

[Exception Handling \(C++/CLI and C++/CX\)](#)

# Modern C++ best practices for exceptions and error handling

---

In modern C++, in most scenarios, the preferred way to report and handle both logic errors and runtime errors is to use exceptions. This is especially true when the stack might contain several function calls between the function that detects the error and the function that has the context to know how to handle it. Exceptions provide a formal, well-defined way for code that detects errors to pass the information up the call stack.

Program errors are generally divided into two categories: logic errors that are caused by programming mistakes, for example, an "index out of range" error, and runtime errors that are beyond the control of programmer, for example, a "network service unavailable" error. In C-style programming and in COM, error reporting is managed either by returning a value that represents an error code or a status code for a particular function, or by setting a global variable that the caller may optionally retrieve after every function call to see whether errors were reported. For example, COM programming uses the HRESULT return value to communicate errors to the caller, and the Win32 API has the GetLastError function to retrieve the last error that was reported by the call stack. In both of these cases, it's up to the caller to recognize the code and respond to it appropriately. If the caller doesn't explicitly handle the error code, the program might crash without warning, or continue to execute with bad data and produce incorrect results.

Exceptions are preferred in modern C++ for the following reasons:

- An exception forces calling code to recognize an error condition and handle it. Unhandled exceptions stop program execution.
- An exception jumps to the point in the call stack that can handle the error. Intermediate functions can let the exception propagate. They do not have to coordinate with other layers.
- The exception stack-unwinding mechanism destroys all objects in scope according to well-defined rules after an exception is thrown.
- An exception enables a clean separation between the code that detects the error and the code that handles the error.

The following simplified example shows the necessary syntax for throwing and catching exceptions in C++.

```
#include <stdexcept>
#include <limits>
#include <iostream>

using namespace std;

void MyFunc(int c)
{
    if (c > numeric_limits< char> ::max())
        throw invalid_argument("MyFunc argument too large.");
    //...
}
```

```

int main()
{
    try
    {
        MyFunc(256); //cause an exception to throw
    }

    catch (invalid_argument& e)
    {
        cerr << e.what() << endl;
        return -1;
    }
    //...
    return 0;
}

```

Exceptions in C++ resemble those in languages such as C# and Java. In the **try** block, if an exception is *thrown* it will be *caught* by the first associated **catch** block whose type matches that of the exception. In other words, execution jumps from the **throw** statement to the **catch** statement. If no usable catch block is found, `std::terminate` is invoked and the program exits. In C++, any type may be thrown; however, we recommend that you throw a type that derives directly or indirectly from `std::exception`. In the previous example, the exception type, `invalid_argument`, is defined in the standard library in the `<stdexcept>` header file. C++ does not provide, and does not require, a **finally** block to make sure that all resources are released if an exception is thrown. The resource acquisition is initialization (RAII) idiom, which uses smart pointers, provides the required functionality for resource cleanup. For more information, see [How to: Design for Exception Safety](#). For information about the C++ stack-unwinding mechanism, see [Exceptions and Stack Unwinding](#).

## Basic guidelines

Robust error handling is challenging in any programming language. Although exceptions provide several features that support good error handling, they can't do all the work for you. To realize the benefits of the exception mechanism, keep exceptions in mind as you design your code.

- Use asserts to check for errors that should never occur. Use exceptions to check for errors that might occur, for example, errors in input validation on parameters of public functions. For more information, see the section titled **Exceptions vs. Assertions**.
- Use exceptions when the code that handles the error might be separated from the code that detects the error by one or more intervening function calls. Consider whether to use error codes instead in performance-critical loops when code that handles the error is tightly-coupled to the code that detects it.
- For every function that might throw or propagate an exception, provide one of the three exception guarantees: the strong guarantee, the basic guarantee, or the nothrow (noexcept) guarantee. For more information, see [How to: Design for Exception Safety](#).
- Throw exceptions by value, catch them by reference. Don't catch what you can't handle.

- Don't use exception specifications, which are deprecated in C++11. For more information, see the section titled **Exception specifications and noexcept**.
- Use standard library exception types when they apply. Derive custom exception types from the [exception Class](#) hierarchy.
- Don't allow exceptions to escape from destructors or memory-deallocation functions.

## Exceptions and performance

The exception mechanism has a very minimal performance cost if no exception is thrown. If an exception is thrown, the cost of the stack traversal and unwinding is roughly comparable to the cost of a function call. Additional data structures are required to track the call stack after a **try** block is entered, and additional instructions are required to unwind the stack if an exception is thrown. However, in most scenarios, the cost in performance and memory footprint is not significant. The adverse effect of exceptions on performance is likely to be significant only on very memory-constrained systems, or in performance-critical loops where an error is likely to occur regularly and the code to handle it is tightly coupled to the code that reports it. In any case, it's impossible to know the actual cost of exceptions without profiling and measuring. Even in those rare cases when the cost is significant, you can weigh it against the increased correctness, easier maintainability, and other advantages that are provided by a well-designed exception policy.

## Exceptions vs. assertions

Exceptions and asserts are two distinct mechanisms for detecting run-time errors in a program. Use asserts to test for conditions during development that should never be true if all your code is correct. There is no point in handling such an error by using an exception because the error indicates that something in the code has to be fixed, and doesn't represent a condition that the program has to recover from at run time. An assert stops execution at the statement so that you can inspect the program state in the debugger; an exception continues execution from the first appropriate catch handler. Use exceptions to check error conditions that might occur at run time even if your code is correct, for example, "file not found" or "out of memory." You might want to recover from these conditions, even if the recovery just outputs a message to a log and ends the program. Always check arguments to public functions by using exceptions. Even if your function is error-free, you might not have complete control over arguments that a user might pass to it.

## C++ exceptions versus Windows SEH exceptions

Both C and C++ programs can use the structured exception handling (SEH) mechanism in the Windows operating system. The concepts in SEH resemble those in C++ exceptions, except that SEH uses the **\_\_try**, **\_\_except**, and **\_\_finally** constructs instead of **try** and **catch**. In the Microsoft C++ compiler (MSVC), C++ exceptions are implemented for SEH. However, when you write C++ code, use the C++ exception syntax.

For more information about SEH, see [Structured Exception Handling \(C/C++\)](#).

## Exception specifications and noexcept

Exception specifications were introduced in C++ as a way to specify the exceptions that a function might throw. However, exception specifications proved problematic in practice, and are deprecated in the C++11 draft standard. We recommend that you do not use exception specifications except for **throw()**, which indicates that the function allows no exceptions to escape. If you must use exception specifications of the type

`throw(type)`, be aware that MSVC departs from the standard in certain ways. For more information, see [Exception Specifications \(throw\)](#). The `noexcept` specifier is introduced in C++11 as the preferred alternative to `throw()`.

## See also

[How to: Interface Between Exceptional and Non-Exceptional Code](#)

[C++ Language Reference](#)

[C++ Standard Library](#)

# How to: Design for exception safety

---

One of the advantages of the exception mechanism is that execution, together with data about the exception, jumps directly from the statement that throws the exception to the first catch statement that handles it. The handler may be any number of levels up in the call stack. Functions that are called between the try statement and the throw statement are not required to know anything about the exception that is thrown. However, they have to be designed so that they can go out of scope "unexpectedly" at any point where an exception might propagate up from below, and do so without leaving behind partially created objects, leaked memory, or data structures that are in unusable states.

## Basic techniques

A robust exception-handling policy requires careful thought and should be part of the design process. In general, most exceptions are detected and thrown at the lower layers of a software module, but typically these layers do not have enough context to handle the error or expose a message to end users. In the middle layers, functions can catch and rethrow an exception when they have to inspect the exception object, or they have additional useful information to provide for the upper layer that ultimately catches the exception. A function should catch and "swallow" an exception only if it is able to completely recover from it. In many cases, the correct behavior in the middle layers is to let an exception propagate up the call stack. Even at the highest layer, it might be appropriate to let an unhandled exception terminate a program if the exception leaves the program in a state in which its correctness cannot be guaranteed.

No matter how a function handles an exception, to help guarantee that it is "exception-safe," it must be designed according to the following basic rules.

### Keep resource classes simple

When you encapsulate manual resource management in classes, use a class that does nothing except manage a single resource. By keeping the class simple, you reduce the risk of introducing resource leaks. Use [smart pointers](#) when possible, as shown in the following example. This example is intentionally artificial and simplistic to highlight the differences when [shared\\_ptr](#) is used.

```
// old-style new/delete version
class NDResourceClass {
private:
    int*    m_p;
    float* m_q;
public:
    NDResourceClass() : m_p(0), m_q(0) {
        m_p = new int;
        m_q = new float;
    }

    ~NDResourceClass() {
        delete m_p;
        delete m_q;
    }
}
```

```

    // Potential leak! When a constructor emits an exception,
    // the destructor will not be invoked.
};

// shared_ptr version
#include <memory>

using namespace std;

class SPResourceClass {
private:
    shared_ptr<int> m_p;
    shared_ptr<float> m_q;
public:
    SPResourceClass() : m_p(new int), m_q(new float) { }
    // Implicitly defined dtor is OK for these members,
    // shared_ptr will clean up and avoid leaks regardless.
};

// A more powerful case for shared_ptr

class Shape {
    // ...
};

class Circle : public Shape {
    // ...
};

class Triangle : public Shape {
    // ...
};

class SPShapeResourceClass {
private:
    shared_ptr<Shape> m_p;
    shared_ptr<Shape> m_q;
public:
    SPShapeResourceClass() : m_p(new Circle), m_q(new Triangle) { }
};

```

## Use the RAII idiom to manage resources

To be exception-safe, a function must ensure that objects that it has allocated by using `malloc` or `new` are destroyed, and all resources such as file handles are closed or released even if an exception is thrown. The *Resource Acquisition Is Initialization* (RAII) idiom ties management of such resources to the lifespan of automatic variables. When a function goes out of scope, either by returning normally or because of an exception, the destructors for all fully-constructed automatic variables are invoked. An RAII wrapper object such as a smart pointer calls the appropriate delete or close function in its destructor. In exception-safe code, it is critically important to pass ownership of each resource immediately to some kind of RAII object. Note that the `vector`, `string`, `make_shared`, `fstream`, and similar classes handle acquisition of the resource for you.



However, `unique_ptr` and traditional `shared_ptr` constructions are special because resource acquisition is performed by the user instead of the object; therefore, they count as *Resource Release Is Destruction* but are questionable as RAI.

## The three exception guarantees

Typically, exception safety is discussed in terms of the three exception guarantees that a function can provide: the *no-fail guarantee*, the *strong guarantee*, and the *basic guarantee*.

### No-fail guarantee

The no-fail (or, "no-throw") guarantee is the strongest guarantee that a function can provide. It states that the function will not throw an exception or allow one to propagate. However, you cannot reliably provide such a guarantee unless (a) you know that all the functions that this function calls are also no-fail, or (b) you know that any exceptions that are thrown are caught before they reach this function, or (c) you know how to catch and correctly handle all exceptions that might reach this function.

Both the strong guarantee and the basic guarantee rely on the assumption that the destructors are no-fail. All containers and types in the Standard Library guarantee that their destructors do not throw. There is also a converse requirement: The Standard Library requires that user-defined types that are given to it—for example, as template arguments—must have non-throwing destructors.

### Strong guarantee

The strong guarantee states that if a function goes out of scope because of an exception, it will not leak memory and program state will not be modified. A function that provides a strong guarantee is essentially a transaction that has commit or rollback semantics: either it completely succeeds or it has no effect.

### Basic guarantee

The basic guarantee is the weakest of the three. However, it might be the best choice when a strong guarantee is too expensive in memory consumption or in performance. The basic guarantee states that if an exception occurs, no memory is leaked and the object is still in a usable state even though the data might have been modified.

## Exception-safe classes

A class can help ensure its own exception safety, even when it is consumed by unsafe functions, by preventing itself from being partially constructed or partially destroyed. If a class constructor exits before completion, then the object is never created and its destructor will never be called. Although automatic variables that are initialized prior to the exception will have their destructors invoked, dynamically allocated memory or resources that are not managed by a smart pointer or similar automatic variable will be leaked.

The built-in types are all no-fail, and the Standard Library types support the basic guarantee at a minimum. Follow these guidelines for any user-defined type that must be exception-safe:

- Use smart pointers or other RAI-type wrappers to manage all resources. Avoid resource management functionality in your class destructor, because the destructor will not be invoked if the constructor throws an exception. However, if the class is a dedicated resource manager that controls just one resource, then it's acceptable to use the destructor to manage resources.

- Understand that an exception thrown in a base class constructor cannot be swallowed in a derived class constructor. If you want to translate and re-throw the base class exception in a derived constructor, use a function try block.
- Consider whether to store all class state in a data member that is wrapped in a smart pointer, especially if a class has a concept of "initialization that is permitted to fail." Although C++ allows for uninitialized data members, it does not support uninitialized or partially initialized class instances. A constructor must either succeed or fail; no object is created if the constructor does not run to completion.
- Do not allow any exceptions to escape from a destructor. A basic axiom of C++ is that destructors should never allow an exception to propagate up the call stack. If a destructor must perform a potentially exception-throwing operation, it must do so in a try catch block and swallow the exception. The standard library provides this guarantee on all destructors it defines.

## See also

[Modern C++ best practices for exceptions and error handling](#)

[How to: Interface Between Exceptional and Non-Exceptional Code](#)

# How to: Interface between exceptional and non-exceptional code

---

This article describes how to implement consistent exception-handling in a C++ module, and also how to translate those exceptions to and from error codes at the exception boundaries.

Sometimes a C++ module has to interface with code that doesn't use exceptions (non-exceptional code). Such an interface is known as an *exception boundary*. For example, you may want to call the Win32 function `CreateFile` in your C++ program. `CreateFile` doesn't throw exceptions; instead it sets error codes that can be retrieved by the `GetLastError` function. If your C++ program is non-trivial, then in it you probably prefer to have a consistent exception-based error-handling policy. And you probably don't want to abandon exceptions just because you interface with non-exceptional code, and neither do you want to mix exception-based and non-exception-based error policies in your C++ module.

## Calling non-exceptional functions from C++

When you call a non-exceptional function from C++, the idea is to wrap that function in a C++ function that detects any errors and then possibly throws an exception. When you design such a wrapper function, first decide which type of exception guarantee to provide: no-throw, strong, or basic. Second, design the function so that all resources, for example, file handles, are correctly released if an exception is thrown. Typically, this means that you use smart pointers or similar resource managers to own the resources. For more information about design considerations, see [How to: Design for Exception Safety](#).

### Example

The following example shows C++ functions that use the Win32 `CreateFile` and `ReadFile` functions internally to open and read two files. The `File` class is a resource acquisition is initialization (RAII) wrapper for the file handles. Its constructor detects a "file not found" condition and throws an exception to propagate the error up the call stack of the C++ module (in this example, the `main()` function). If an exception is thrown after a `File` object is fully constructed, the destructor automatically calls `CloseHandle` to release the file handle. (If you prefer, you can use the Active Template Library (ATL) `CHandle` class for this same purpose, or a `unique_ptr` together with a custom deleter.) The functions that call Win32 and CRT APIs detect errors and then throw C++ exceptions using the locally-defined `ThrowLastErrorIf` function, which in turn uses the `Win32Exception` class, derived from the `runtime_error` class. All functions in this example provide a strong exception guarantee; if an exception is thrown at any point in these functions, no resources are leaked and no program state is modified.

```
// compile with: /EHsc
#include <Windows.h>
#include <stdlib.h>
#include <vector>
#include <iostream>
#include <string>
#include <limits>
#include <stdexcept>
```

```

using namespace std;

string FormatErrorMessage(DWORD error, const string& msg)
{
    static const int BUFFERLENGTH = 1024;
    vector<char> buf(BUFFERLENGTH);
    FormatMessageA(FORMAT_MESSAGE_FROM_SYSTEM, 0, error, 0, buf.data(),
        BUFFERLENGTH - 1, 0);
    return string(buf.data()) + " (" + msg + ")";
}

class Win32Exception : public runtime_error
{
private:
    DWORD m_error;
public:
    Win32Exception(DWORD error, const string& msg)
        : runtime_error(FormatErrorMessage(error, msg)), m_error(error) { }

    DWORD GetErrorCode() const { return m_error; }
};

void ThrowLastErrorIf(bool expression, const string& msg)
{
    if (expression) {
        throw Win32Exception(GetLastError(), msg);
    }
}

class File
{
private:
    HANDLE m_handle;

    // Declared but not defined, to avoid double closing.
    File& operator=(const File&);
    File(const File&);
public:
    explicit File(const string& filename)
    {
        m_handle = CreateFileA(filename.c_str(), GENERIC_READ, FILE_SHARE_READ,
            nullptr, OPEN_EXISTING, FILE_ATTRIBUTE_READONLY, nullptr);
        ThrowLastErrorIf(m_handle == INVALID_HANDLE_VALUE,
            "CreateFile call failed on file named " + filename);
    }

    ~File() { CloseHandle(m_handle); }

    HANDLE GetHandle() { return m_handle; }
};

size_t GetFileSizeSafe(const string& filename)
{
    File fobj(filename);

```

```

    LARGE_INTEGER filesize;

    BOOL result = GetFileSizeEx(fobj.GetHandle(), &filesize);
    ThrowLastErrorIf(result == FALSE, "GetFileSizeEx failed: " + filename);

    if (filesize.QuadPart < (numeric_limits<size_t>::max)()) {
        return filesize.QuadPart;
    } else {
        throw;
    }
}

vector<char> ReadFileVector(const string& filename)
{
    File fobj(filename);
    size_t filesize = GetFileSizeSafe(filename);
    DWORD bytesRead = 0;

    vector<char> readbuffer(filesize);

    BOOL result = ReadFile(fobj.GetHandle(), readbuffer.data(), readbuffer.size(),
        &bytesRead, nullptr);
    ThrowLastErrorIf(result == FALSE, "ReadFile failed: " + filename);

    cout << filename << " file size: " << filesize << ", bytesRead: "
        << bytesRead << endl;

    return readbuffer;
}

bool IsFileDiff(const string& filename1, const string& filename2)
{
    return ReadFileVector(filename1) != ReadFileVector(filename2);
}

#include <iomanip>

int main ( int argc, char* argv[] )
{
    string filename1("file1.txt");
    string filename2("file2.txt");

    try
    {
        if(argc > 2) {
            filename1 = argv[1];
            filename2 = argv[2];
        }

        cout << "Using file names " << filename1 << " and " << filename2 << endl;

        if (IsFileDiff(filename1, filename2)) {
            cout << "+++ Files are different." << endl;
        } else {

```

```

        cout<< "=== Files match." << endl;
    }
}
catch(const Win32Exception& e)
{
    ios state(nullptr);
    state.copyfmt(cout);
    cout << e.what() << endl;
    cout << "Error code: 0x" << hex << uppercase << setw(8) << setfill('0')
        << e.GetErrorCode() << endl;
    cout.copyfmt(state); // restore previous formatting
}
}

```

## Calling exceptional code from non-exceptional code

C++ functions that are declared as "extern C" can be called by C programs. C++ COM servers can be consumed by code written in any of a number of different languages. When you implement public exception-aware functions in C++ to be called by non-exceptional code, the C++ function must not allow any exceptions to propagate back to the caller. Therefore, the C++ function must specifically catch every exception that it knows how to handle and, if appropriate, convert the exception to an error code that the caller understands. If not all potential exceptions are known, the C++ function should have a `catch(...)` block as the last handler. In such a case, it's best to report a fatal error to the caller, because your program might be in an unknown state.

The following example shows a function that assumes that any exception that might be thrown is either a `Win32Exception` or an exception type derived from `std::exception`. The function catches any exception of these types and propagates the error information as a Win32 error code to the caller.

```

BOOL DiffFiles2(const string& file1, const string& file2)
{
    try
    {
        File f1(file1);
        File f2(file2);
        if (IsTextFileDiff(f1, f2))
        {
            SetLastError(MY_APPLICATION_ERROR_FILE_MISMATCH);
            return FALSE;
        }
        return TRUE;
    }
    catch(Win32Exception& e)
    {
        SetLastError(e.GetErrorCode());
    }

    catch(std::exception& e)
    {
        SetLastError(MY_APPLICATION_GENERAL_ERROR);
    }
}

```

```

    }
    return FALSE;
}

```

When you convert from exceptions to error codes, one potential issue is that error codes often don't contain the richness of information that an exception can store. To address this, you can provide a **catch** block for each specific exception type that might be thrown, and perform logging to record the details of the exception before it is converted to an error code. This approach can create a lot of code repetition if multiple functions all use the same set of **catch** blocks. A good way to avoid code repetition is by refactoring those blocks into one private utility function that implements the **try** and **catch** blocks and accepts a function object that is invoked in the **try** block. In each public function, pass the code to the utility function as a lambda expression.

```

template<typename Func>
bool Win32ExceptionBoundary(Func&& f)
{
    try
    {
        return f();
    }
    catch(Win32Exception& e)
    {
        SetLastError(e.GetErrorCode());
    }
    catch(const std::exception& e)
    {
        SetLastError(MY_APPLICATION_GENERAL_ERROR);
    }
    return false;
}

```

The following example shows how to write the lambda expression that defines the functor. When a functor is defined "inline" by using a lambda expression, it is often easier to read than it would be if it were written as a named function object.

```

bool DiffFiles3(const string& file1, const string& file2)
{
    return Win32ExceptionBoundary([&]() -> bool
    {
        File f1(file1);
        File f2(file2);
        if (IsTextFileDiff(f1, f2))
        {
            SetLastError(MY_APPLICATION_ERROR_FILE_MISMATCH);
            return false;
        }
        return true;
    });
}

```

---

For more information about lambda expressions, see [Lambda Expressions](#).

## See also

[Modern C++ best practices for exceptions and error handling](#)

[How to: Design for Exception Safety](#)



# try, throw, and catch Statements (C++)

---

To implement exception handling in C++, you use **try**, **throw**, and **catch** expressions.

First, use a **try** block to enclose one or more statements that might throw an exception.

A **throw** expression signals that an exceptional condition—often, an error—has occurred in a **try** block. You can use an object of any type as the operand of a **throw** expression. Typically, this object is used to communicate information about the error. In most cases, we recommend that you use the `std::exception` class or one of the derived classes that are defined in the standard library. If one of those is not appropriate, we recommend that you derive your own exception class from `std::exception`.

To handle exceptions that may be thrown, implement one or more **catch** blocks immediately following a **try** block. Each **catch** block specifies the type of exception it can handle.

This example shows a **try** block and its handlers. Assume that `GetNetworkResource()` acquires data over a network connection and that the two exception types are user-defined classes that derive from `std::exception`. Notice that the exceptions are caught by **const** reference in the **catch** statement. We recommend that you throw exceptions by value and catch them by const reference.

## Example

```
MyData md;
try {
    // Code that could throw an exception
    md = GetNetworkResource();
}
catch (const networkIOException& e) {
    // Code that executes when an exception of type
    // networkIOException is thrown in the try block
    // ...
    // Log error message in the exception object
    cerr << e.what();
}
catch (const myDataFormatException& e) {
    // Code that handles another exception type
    // ...
    cerr << e.what();
}

// The following syntax shows a throw expression
MyData GetNetworkResource()
{
    // ...
    if (IOSuccess == false)
        throw networkIOException("Unable to connect");
    // ...
    if (readError)
        throw myDataFormatException("Format error");
}
```

```
// ...  
}
```

## Remarks

The code after the **try** clause is the guarded section of code. The **throw** expression *throws*—that is, raises—an exception. The code block after the **catch** clause is the exception handler. This is the handler that *catches* the exception that's thrown if the types in the **throw** and **catch** expressions are compatible. For a list of rules that govern type-matching in **catch** blocks, see [How Catch Blocks are Evaluated](#). If the **catch** statement specifies an ellipsis (...) instead of a type, the **catch** block handles every type of exception. When you compile with the `/EHa` option, these can include C structured exceptions and system-generated or application-generated asynchronous exceptions such as memory protection, divide-by-zero, and floating-point violations. Because **catch** blocks are processed in program order to find a matching type, an ellipsis handler must be the last handler for the associated **try** block. Use `catch(...)` with caution; do not allow a program to continue unless the catch block knows how to handle the specific exception that is caught. Typically, a `catch(...)` block is used to log errors and perform special cleanup before program execution is stopped.

A **throw** expression that has no operand re-throws the exception currently being handled. We recommend this form when re-throwing the exception, because this preserves the original exception's polymorphic type information. Such an expression should only be used in a **catch** handler or in a function that's called from a **catch** handler. The re-thrown exception object is the original exception object, not a copy.

```
try {  
    throw CSomeOtherException();  
}  
catch(...) {  
    // Catch all exceptions - dangerous!!!  
    // Respond (perhaps only partially) to the exception, then  
    // re-throw to pass the exception to some other handler  
    // ...  
    throw;  
}
```

## See also

[Modern C++ best practices for exceptions and error handling](#)

[Keywords](#)

[Unhandled C++ Exceptions](#)

[\\_\\_uncaught\\_exception](#)

# How Catch Blocks are Evaluated (C++)

---

C++ enables you to throw exceptions of any type, although in general it is recommended to throw types that are derived from `std::exception`. A C++ exception can be caught by a **catch** handler that specifies the same type as the thrown exception, or by a handler that can catch any type of exception.

If the type of thrown exception is a class, which also has a base class (or classes), it can be caught by handlers that accept base classes of the exception's type, as well as references to bases of the exception's type. Note that when an exception is caught by a reference, it is bound to the actual thrown exception object; otherwise, it is a copy (much the same as an argument to a function).

When an exception is thrown, it may be caught by the following types of **catch** handlers:

- A handler that can accept any type (using the ellipsis syntax).
- A handler that accepts the same type as the exception object; because it is a copy, **const** and **volatile** modifiers are ignored.
- A handler that accepts a reference to the same type as the exception object.
- A handler that accepts a reference to a **const** or **volatile** form of the same type as the exception object.
- A handler that accepts a base class of the same type as the exception object; since it is a copy, **const** and **volatile** modifiers are ignored. The **catch** handler for a base class must not precede the **catch** handler for the derived class.
- A handler that accepts a reference to a base class of the same type as the exception object.
- A handler that accepts a reference to a **const** or **volatile** form of a base class of the same type as the exception object.
- A handler that accepts a pointer to which a thrown pointer object can be converted via standard pointer conversion rules.

The order in which **catch** handlers appear is significant, because handlers for a given **try** block are examined in order of their appearance. For example, it is an error to place the handler for a base class before the handler for a derived class. After a matching **catch** handler is found, subsequent handlers are not examined. As a result, an ellipsis **catch** handler must be the last handler for its **try** block. For example:

```
// ...
try
{
    // ...
}
catch( ... )
{
    // Handle exception here.
}
// Error: the next two handlers are never examined.
```

```
catch( const char * str )
{
    cout << "Caught exception: " << str << endl;
}
catch( CExcptClass E )
{
    // Handle CExcptClass exception here.
}
```

In this example, the ellipsis **catch** handler is the only handler that is examined.

## See also

[Modern C++ best practices for exceptions and error handling](#)

# Exceptions and Stack Unwinding in C++

---

In the C++ exception mechanism, control moves from the throw statement to the first catch statement that can handle the thrown type. When the catch statement is reached, all of the automatic variables that are in scope between the throw and catch statements are destroyed in a process that is known as *stack unwinding*. In stack unwinding, execution proceeds as follows:

1. Control reaches the **try** statement by normal sequential execution. The guarded section in the **try** block is executed.
2. If no exception is thrown during execution of the guarded section, the **catch** clauses that follow the **try** block are not executed. Execution continues at the statement after the last **catch** clause that follows the associated **try** block.
3. If an exception is thrown during execution of the guarded section or in any routine that the guarded section calls either directly or indirectly, an exception object is created from the object that is created by the **throw** operand. (This implies that a copy constructor may be involved.) At this point, the compiler looks for a **catch** clause in a higher execution context that can handle an exception of the type that is thrown, or for a **catch** handler that can handle any type of exception. The **catch** handlers are examined in order of their appearance after the **try** block. If no appropriate handler is found, the next dynamically enclosing **try** block is examined. This process continues until the outermost enclosing **try** block is examined.
4. If a matching handler is still not found, or if an exception occurs during the unwinding process but before the handler gets control, the predefined run-time function **terminate** is called. If an exception occurs after the exception is thrown but before the unwind begins, **terminate** is called.
5. If a matching **catch** handler is found, and it catches by value, its formal parameter is initialized by copying the exception object. If it catches by reference, the parameter is initialized to refer to the exception object. After the formal parameter is initialized, the process of unwinding the stack begins. This involves the destruction of all automatic objects that were fully constructed—but not yet destructed—between the beginning of the **try** block that is associated with the **catch** handler and the throw site of the exception. Destruction occurs in reverse order of construction. The **catch** handler is executed and the program resumes execution after the last handler—that is, at the first statement or construct that is not a **catch** handler. Control can only enter a **catch** handler through a thrown exception, never through a **goto** statement or a **case** label in a **switch** statement.

## Stack unwinding example

The following example demonstrates how the stack is unwound when an exception is thrown. Execution on the thread jumps from the throw statement in **C** to the catch statement in **main**, and unwinds each function along the way. Notice the order in which the **Dummy** objects are created and then destroyed as they go out of scope. Also notice that no function completes except **main**, which contains the catch statement. Function **A** never returns from its call to **B()**, and **B** never returns from its call to **C()**. If you uncomment the definition of the **Dummy** pointer and the corresponding delete statement, and then run the program, notice that the pointer is never deleted. This shows what can happen when functions do not provide an exception guarantee. For

more information, see [How to: Design for Exceptions](#). If you comment out the catch statement, you can observe what happens when a program terminates because of an unhandled exception.

```
#include <string>
#include <iostream>
using namespace std;

class MyException{};
class Dummy
{
public:
    Dummy(string s) : MyName(s) { PrintMsg("Created Dummy:"); }
    Dummy(const Dummy& other) : MyName(other.MyName){ PrintMsg("Copy created
Dummy:"); }
    ~Dummy(){ PrintMsg("Destroyed Dummy:"); }
    void PrintMsg(string s) { cout << s << MyName << endl; }
    string MyName;
    int level;
};

void C(Dummy d, int i)
{
    cout << "Entering FunctionC" << endl;
    d.MyName = " C";
    throw MyException();

    cout << "Exiting FunctionC" << endl;
}

void B(Dummy d, int i)
{
    cout << "Entering FunctionB" << endl;
    d.MyName = "B";
    C(d, i + 1);
    cout << "Exiting FunctionB" << endl;
}

void A(Dummy d, int i)
{
    cout << "Entering FunctionA" << endl;
    d.MyName = " A" ;
    // Dummy* pd = new Dummy("new Dummy"); //Not exception safe!!!
    B(d, i + 1);
    // delete pd;
    cout << "Exiting FunctionA" << endl;
}

int main()
{
    cout << "Entering main" << endl;
    try
    {
```

```

        Dummy d(" M");
        A(d,1);
    }
    catch (MyException& e)
    {
        cout << "Caught an exception of type: " << typeid(e).name() << endl;
    }

    cout << "Exiting main." << endl;
    char c;
    cin >> c;
}

```

```

/* Output:
    Entering main
    Created Dummy: M
    Copy created Dummy: M
    Entering FunctionA
    Copy created Dummy: A
    Entering FunctionB
    Copy created Dummy: B
    Entering FunctionC
    Destroyed Dummy: C
    Destroyed Dummy: B
    Destroyed Dummy: A
    Destroyed Dummy: M
    Caught an exception of type: class MyException
    Exiting main.

*/

```

# Exception specifications (throw, noexcept) (C++)

---

Exception specifications are a C++ language feature that indicate the programmer's intent about the exception types that can be propagated by a function. You can specify that a function may or may not exit by an exception by using an *exception specification*. The compiler can use this information to optimize calls to the function, and to terminate the program if an unexpected exception escapes the function.

Prior to C++17 there were two kinds of exception specification. The *noexcept specification* was new in C++11. It specifies whether the set of potential exceptions that can escape the function is empty. The *dynamic exception specification*, or `throw(optional_type_list)` specification, was deprecated in C++11 and removed in C++17, except for `throw()`, which is an alias for `noexcept(true)`. This exception specification was designed to provide summary information about what exceptions can be thrown out of a function, but in practice it was found to be problematic. The one dynamic exception specification that did prove to be somewhat useful was the unconditional `throw()` specification. For example, the function declaration:

```
void MyFunction(int i) throw();
```

tells the compiler that the function does not throw any exceptions. However, in `/std:c++14` mode this could lead to undefined behavior if the function does throw an exception. Therefore we recommend using the `noexcept` operator instead of the one above:

```
void MyFunction(int i) noexcept;
```

The following table summarizes the Microsoft C++ implementation of exception specifications:

Exception specification	Meaning
<code>noexcept</code> <code>noexcept(true)</code> <code>throw()</code>	<p>The function does not throw an exception. In <code>/std:c++14</code> mode (which is the default), <code>noexcept</code> and <code>noexcept(true)</code> are equivalent. When an exception is thrown from a function that is declared <code>noexcept</code> or <code>noexcept(true)</code>, <code>std::terminate</code> is invoked. When an exception is thrown from a function declared as <code>throw()</code> in <code>/std:c++14</code> mode, the result is undefined behavior. No specific function is invoked. This is a divergence from the C++14 standard, which required the compiler to invoke <code>std::unexpected</code>.</p> <p><b>Visual Studio 2017 version 15.5 and later:</b> In <code>/std:c++17</code> mode, <code>noexcept</code>, <code>noexcept(true)</code>, and <code>throw()</code> are all equivalent. In <code>/std:c++17</code> mode, <code>throw()</code> is an alias for <code>noexcept(true)</code>. In <code>/std:c++17</code> mode, when an exception is thrown from a function declared with any of these specifications, <code>std::terminate</code> is invoked as required by the C++17 standard.</p>



Exception specification	Meaning
<code>noexcept(false)</code> <code>throw(...)</code> No specification	The function can throw an exception of any type.
<code>throw(type)</code>	(C++14 and earlier) The function can throw an exception of type <code>type</code> . The compiler accepts the syntax, but interprets it as <code>noexcept(false)</code> . In <b>/std:c++17</b> mode the compiler issues warning C5040.

If exception handling is used in an application, there must be a function in the call stack that handles thrown exceptions before they exit the outer scope of a function marked `noexcept`, `noexcept(true)`, or `throw()`. If any functions called between the one that throws an exception and the one that handles the exception are specified as `noexcept`, `noexcept(true)` (or `throw()` in **/std:c++17** mode), the program is terminated when the `noexcept` function propagates the exception.

The exception behavior of a function depends on the following factors:

- Which [language standard compilation mode](#) is set.
- Whether you are compiling the function under C or C++.
- Which [/EH](#) compiler option you use.
- Whether you explicitly specify the exception specification.

Explicit exception specifications are not allowed on C functions. A C function is assumed not to throw exceptions under **/EHsc**, and may throw structured exceptions under **/EHs**, **/EHa**, or **/EHac**.

The following table summarizes whether a C++ function may potentially throw under various compiler exception handling options:

Function	/EHsc	/EHs	/EHa	/EHac
C++ function with no exception specification	Yes	Yes	Yes	Yes
C++ function with <code>noexcept</code> , <code>noexcept(true)</code> , or <code>throw()</code> exception specification	No	No	Yes	Yes
C++ function with <code>noexcept(false)</code> , <code>throw(...)</code> , or <code>throw(type)</code> exception specification	Yes	Yes	Yes	Yes

## Example

```
// exception_specification.cpp
// compile with: /EHs
#include <stdio.h>

void handler() {
    printf_s("in handler\n");
}
```

```

}

void f1(void) throw(int) {
    printf_s("About to throw 1\n");
    if (1)
        throw 1;
}

void f5(void) throw() {
    try {
        f1();
    }
    catch(...) {
        handler();
    }
}

// invalid, doesn't handle the int exception thrown from f1()
// void f3(void) throw() {
//     f1();
// }

void __declspec(nothrow) f2(void) {
    try {
        f1();
    }
    catch(int) {
        handler();
    }
}

// only valid if compiled without /EHc
// /EHc means assume extern "C" functions don't throw exceptions
extern "C" void f4(void);
void f4(void) {
    f1();
}

int main() {
    f2();

    try {
        f4();
    }
    catch(...) {
        printf_s("Caught exception from f4\n");
    }
    f5();
}

```

```
About to throw 1  
in handler  
About to throw 1  
Caught exception from f4  
About to throw 1  
in handler
```

## See also

[try, throw, and catch Statements \(C++\)](#)

[Modern C++ best practices for exceptions and error handling](#)

# noexcept (C++)

---

**C++11:** Specifies whether a function might throw exceptions.

## Syntax

```
noexcept-expression:   noexcept   noexcept( constant-expression )
```

## Parameters

*constant-expression*

A constant expression of type **bool** that represents whether the set of potential exception types is empty. The unconditional version is equivalent to **noexcept(true)**.

## Remarks

A *noexcept expression* is a kind of *exception specification*, a suffix to a function declaration that represents a set of types that might be matched by an exception handler for any exception that exits a function. Unary conditional operator **noexcept(constant\_expression)** where *constant\_expression* yields **true**, and its unconditional synonym **noexcept**, specify that the set of potential exception types that can exit a function is empty. That is, the function never throws an exception and never allows an exception to be propagated outside its scope. The operator **noexcept(constant\_expression)** where *constant\_expression* yields **false**, or the absence of an exception specification (other than for a destructor or deallocation function), indicates that the set of potential exceptions that can exit the function is the set of all types.

Mark a function as **noexcept** only if all the functions that it calls, either directly or indirectly, are also **noexcept** or **const**. The compiler does not necessarily check every code path for exceptions that might bubble up to a **noexcept** function. If an exception does exit the outer scope of a function marked **noexcept**, **std::terminate** is invoked immediately, and there is no guarantee that destructors of any in-scope objects will be invoked. Use **noexcept** instead of the dynamic exception specifier **throw()**, which is now deprecated in the standard. We recommended you apply **noexcept** to any function that never allows an exception to propagate up the call stack. When a function is declared **noexcept**, it enables the compiler to generate more efficient code in several different contexts. For more information, see [Exception specifications](#).

## Example

A template function that copies its argument might be declared **noexcept** on the condition that the object being copied is a plain old data type (POD). Such a function could be declared like this:

```
#include <type_traits>

template <typename T>
T copy_object(const T& obj) noexcept(std::is_pod<T>)
{
    // ...
}
```

## See also

[Modern C++ best practices for exceptions and error handling](#)

[Exception Specifications \(throw, noexcept\)](#)

# Unhandled C++ exceptions

---

If a matching handler (or ellipsis **catch** handler) cannot be found for the current exception, the predefined **terminate** run-time function is called. (You can also explicitly call **terminate** in any of your handlers.) The default action of **terminate** is to call **abort**. If you want **terminate** to call some other function in your program before exiting the application, call the **set\_terminate** function with the name of the function to be called as its single argument. You can call **set\_terminate** at any point in your program. The **terminate** routine always calls the last function given as an argument to **set\_terminate**.

## Example

The following example throws a **char \*** exception, but does not contain a handler designated to catch exceptions of type **char \***. The call to **set\_terminate** instructs **terminate** to call **term\_func**.

```
// exceptions_Unhandled_Exceptions.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;
void term_func() {
    cout << "term_func was called by terminate." << endl;
    exit( -1 );
}
int main() {
    try
    {
        set_terminate( term_func );
        throw "Out of memory!"; // No catch handler for this exception
    }
    catch( int )
    {
        cout << "Integer exception raised." << endl;
    }
    return 0;
}
```

## Output

```
term_func was called by terminate.
```

The **term\_func** function should terminate the program or current thread, ideally by calling **exit**. If it doesn't, and instead returns to its caller, **abort** is called.

## See also

[Modern C++ best practices for exceptions and error handling](#)

# Mixing C (structured) and C++ exceptions

---

If you want to write portable code, the use of structured exception handling (SEH) in a C++ program isn't recommended. However, you may sometimes want to compile using `/EHa` and mix structured exceptions and C++ source code, and need some facility for handling both kinds of exceptions. Because a structured exception handler has no concept of objects or typed exceptions, it can't handle exceptions thrown by C++ code. However, C++ **catch** handlers can handle structured exceptions. C++ exception handling syntax (**try**, **throw**, **catch**) isn't accepted by the C compiler, but structured exception handling syntax (**\_\_try**, **\_\_except**, **\_\_finally**) is supported by the C++ compiler.

See [\\_set\\_se\\_translator](#) for information on how to handle structured exceptions as C++ exceptions.

If you mix structured and C++ exceptions, be aware of these potential issues:

- C++ exceptions and structured exceptions cannot be mixed within the same function.
- Termination handlers (**\_\_finally** blocks) are always executed, even during unwinding after an exception is thrown.
- C++ exception handling can catch and preserve unwind semantics in all modules compiled with the `/EH` compiler options, which enable unwind semantics.
- There may be some situations in which destructor functions are not called for all objects. For example, if a structured exception occurs while attempting to make a function call through an uninitialized function pointer, and that function takes as parameters objects that were constructed before the call, the destructors of those objects are not called during stack unwind.

## Next steps

- [Using `setjmp` or `longjmp` in C++ programs](#)

See more information on the use of `setjmp` and `longjmp` in C++ programs.

- [Handle structured exceptions in C++](#)

See examples of the ways you can use C++ to handle structured exceptions.

## See also

[Modern C++ best practices for exceptions and error handling](#)

# Using `setjmp` and `longjmp`

---

When `setjmp` and `longjmp` are used together, they provide a way to execute a non-local **goto**. They are typically used in C code to pass execution control to error-handling or recovery code in a previously called routine without using the standard calling or return conventions.

[!CAUTION] Because `setjmp` and `longjmp` don't support correct destruction of stack frame objects portably between C++ compilers, and because they might degrade performance by preventing optimization on local variables, we don't recommend their use in C++ programs. We recommend you use **try** and **catch** constructs instead.

If you decide to use `setjmp` and `longjmp` in a C++ program, also include `<setjmp.h>` or `<setjmpex.h>` to assure correct interaction between the functions and Structured Exception Handling (SEH) or C++ exception handling.

## Microsoft Specific

If you use an `/EH` option to compile C++ code, destructors for local objects are called during the stack unwind. However, if you use `/EHs` or `/EHsc` to compile, and one of your functions that uses `noexcept` calls `longjmp`, then the destructor unwind for that function might not occur, depending on the optimizer state.

In portable code, when a `longjmp` call is executed, correct destruction of frame-based objects is explicitly not guaranteed by the standard, and may not be supported by other compilers. To let you know, at warning level 4, a call to `setjmp` causes warning C4611: interaction between '`_setjmp`' and C++ object destruction is non-portable.

## END Microsoft Specific

## See also

[Mixing C \(Structured\) and C++ Exceptions](#)



# Handle structured exceptions in C++

---

The major difference between C structured exception handling (SEH) and C++ exception handling is that the C++ exception handling model deals in types, while the C structured exception handling model deals with exceptions of one type; specifically, **unsigned int**. That is, C exceptions are identified by an unsigned integer value, whereas C++ exceptions are identified by data type. When a structured exception is raised in C, each possible handler executes a filter that examines the C exception context and determines whether to accept the exception, pass it to some other handler, or ignore it. When an exception is thrown in C++, it may be of any type.

A second difference is that the C structured exception handling model is referred to as *asynchronous*, because exceptions occur secondary to the normal flow of control. The C++ exception handling mechanism is fully *synchronous*, which means that exceptions occur only when they are thrown.

When you use the [/EHs or /EHsc](#) compiler option, no C++ exception handlers handle structured exceptions. These exceptions are handled only by **\_\_except** structured exception handlers or **\_\_finally** structured termination handlers. For information, see [Structured Exception Handling \(C/C++\)](#).

Under the [/EHa](#) compiler option, if a C exception is raised in a C++ program, it can be handled by a structured exception handler with its associated filter or by a C++ **catch** handler, whichever is dynamically nearer to the exception context. For example, this sample C++ program raises a C exception inside a C++ **try** context:

## Example - Catch a C exception in a C++ catch block

```
// exceptions_Exception_Handling_Differences.cpp
// compile with: /EHa
#include <iostream>

using namespace std;
void SEHFunc( void );

int main() {
    try {
        SEHFunc();
    }
    catch( ... ) {
        cout << "Caught a C exception."<< endl;
    }
}

void SEHFunc() {
    __try {
        int x, y = 0;
        x = 5 / y;
    }
    __finally {
        cout << "In finally." << endl;
    }
}
```

```
}  
}
```

In finally.  
Caught a C exception.

## C exception wrapper classes

In a simple example like the above, the C exception can be caught only by an ellipsis (...) **catch** handler. No information about the type or nature of the exception is communicated to the handler. While this method works, in some cases you may want to define a transformation between the two exception handling models so that each C exception is associated with a specific class. To transform one, you can define a C exception "wrapper" class, which can be used or derived from in order to attribute a specific class type to a C exception. By doing so, each C exception can be handled separately by a specific C++ **catch** handler, instead of all of them in a single handler.

Your wrapper class might have an interface consisting of some member functions that determine the value of the exception, and that access the extended exception context information provided by the C exception model. You might also want to define a default constructor and a constructor that accepts an **unsigned int** argument (to provide for the underlying C exception representation), and a bitwise copy constructor. Here is a possible implementation of a C exception wrapper class:

```
// exceptions_Exception_Handling_Differences2.cpp  
// compile with: /c  
class SE_Exception {  
private:  
    SE_Exception() {}  
    SE_Exception( SE_Exception& ) {}  
    unsigned int nSE;  
public:  
    SE_Exception( unsigned int n ) : nSE( n ) {}  
    ~SE_Exception() {}  
    unsigned int getSeNumber() {  
        return nSE;  
    }  
};
```

To use this class, install a custom C exception translation function that is called by the internal exception handling mechanism each time a C exception is thrown. Within your translation function, you can throw any typed exception (perhaps an `SE_Exception` type, or a class type derived from `SE_Exception`) that can be caught by an appropriate matching C++ **catch** handler. The translation function can instead return, which indicates that it did not handle the exception. If the translation function itself raises a C exception, `terminate` is called.

To specify a custom translation function, call the `_set_se_translator` function with the name of your translation function as its single argument. The translation function that you write is called once for each function

invocation on the stack that has **try** blocks. There is no default translation function; if you do not specify one by calling `_set_se_translator`, the C exception can only be caught by an ellipsis **catch** handler.

## Example - Use a custom translation function

For example, the following code installs a custom translation function, and then raises a C exception that is wrapped by the `SE_Exception` class:

```
// exceptions_Exception_Handling_Differences3.cpp
// compile with: /EHa
#include <stdio.h>
#include <eh.h>
#include <windows.h>

class SE_Exception {
private:
    SE_Exception() {}
    unsigned int nSE;
public:
    SE_Exception( SE_Exception& e ) : nSE(e.nSE) {}
    SE_Exception(unsigned int n) : nSE(n) {}
    ~SE_Exception() {}
    unsigned int getSeNumber() { return nSE; }
};

void SEFunc() {
    __try {
        int x, y = 0;
        x = 5 / y;
    }
    __finally {
        printf_s( "In finally\n" );
    }
}

void trans_func( unsigned int u, _EXCEPTION_POINTERS* pExp ) {
    printf_s( "In trans_func.\n" );
    throw SE_Exception( u );
}

int main() {
    _set_se_translator( trans_func );
    try {
        SEFunc();
    }
    catch( SE_Exception e ) {
        printf_s( "Caught a __try exception with SE_Exception.\n" );
        printf_s( "nSE = 0x%x\n", e.getSeNumber() );
    }
}
```

```
In trans_func.  
In finally  
Caught a __try exception with SE_Exception.  
nSE = 0xc0000094
```

## See also

[Mixing C \(Structured\) and C++ exceptions](#)

# Structured Exception Handling (C/C++)

---

Structured exception handling (SEH) is a Microsoft extension to C to handle certain exceptional code situations, such as hardware faults, gracefully. Although Windows and Microsoft C++ support SEH, we recommend that you use ISO-standard C++ exception handling because it makes your code more portable and flexible. Nevertheless, to maintain existing code or for particular kinds of programs, you still might have to use SEH.

## Microsoft-specific:

## Grammar

*try-except-statement :*

**\_\_try** *compound-statement* **\_\_except** ( *expression* ) *compound-statement*

*try-finally-statement :*

**\_\_try** *compound-statement* **\_\_finally** *compound-statement*

## Remarks

With SEH, you can ensure that resources such as memory blocks and files are released correctly if execution unexpectedly terminates. You can also handle specific problems—for example, insufficient memory—by using concise structured code that does not rely on **goto** statements or elaborate testing of return codes.

The try-except and try-finally statements referred to in this article are Microsoft extensions to the C language. They support SEH by enabling applications to gain control of a program after events that would otherwise terminate execution. Although SEH works with C++ source files, it's not specifically designed for C++. If you use SEH in a C++ program that you compile by using the [/EHa](#) or [/EHsc](#) option, destructors for local objects are called but other execution behavior might not be what you expect. For an illustration, see the example later in this article. In most cases, instead of SEH we recommend that you use ISO-standard [C++ exception handling](#), which the Microsoft C++ compiler also supports. By using C++ exception handling, you can ensure that your code is more portable, and you can handle exceptions of any type.

If you have C code that uses SEH, you can mix it with C++ code that uses C++ exception handling. For information, see [Handle structured exceptions in C++](#).

There are two SEH mechanisms:

- [Exception handlers](#), or **\_\_except** blocks, which can respond to or dismiss the exception.
- [Termination handlers](#), or **\_\_finally** blocks, which are always called, whether an exception causes termination or not.

These two kinds of handlers are distinct, but are closely related through a process known as "unwinding the stack." When a structured exception occurs, Windows looks for the most recently installed exception handler that is currently active. The handler can do one of three things:

- Fail to recognize the exception and pass control to other handlers.

- Recognize the exception but dismiss it.
- Recognize the exception and handle it.

The exception handler that recognizes the exception may not be in the function that was running when the exception occurred. In some cases, it may be in a function much higher on the stack. The currently running function and all other functions on the stack frame are terminated. During this process, the stack is "unwound;" that is, local non-static variables of terminated functions are cleared from the stack.

As it unwinds the stack, the operating system calls any termination handlers that you've written for each function. By using a termination handler, you can clean up resources that otherwise would remain open because of an abnormal termination. If you've entered a critical section, you can exit it in the termination handler. If the program is going to shut down, you can perform other housekeeping tasks such as closing and removing temporary files.

## Next steps

- [Writing an exception handler](#)
- [Writing a termination handler](#)
- [Handle structured exceptions in C++](#)

## Example

As stated earlier, destructors for local objects are called if you use SEH in a C++ program and compile it by using the `/EHa` or `/EHsc` option. However, the behavior during execution may not be what you expect if you are also using C++ exceptions. This example demonstrates these behavioral differences.

```
#include <stdio.h>
#include <Windows.h>
#include <exception>

class TestClass
{
public:
    ~TestClass()
    {
        printf("Destroying TestClass!\r\n");
    }
};

__declspec(noinline) void TestCPPEX()
{
#ifdef CPPEX
    printf("Throwing C++ exception\r\n");
    throw std::exception("");
#else
    printf("Triggering SEH exception\r\n");
    volatile int *pInt = 0x00000000;
    *pInt = 20;
#endif
}
```

```

#endif
}

__declspec(noinline) void TestExceptions()
{
    TestClass d;
    TestCPPEX();
}

int main()
{
    __try
    {
        TestExceptions();
    }
    __except(EXCEPTION_EXECUTE_HANDLER)
    {
        printf("Executing SEH __except block\r\n");
    }

    return 0;
}

```

If you use **/EHsc** to compile this code but the local test control macro **CPPEX** is undefined, there is no execution of the **TestClass** destructor and the output looks like this:

```

Triggering SEH exception
Executing SEH __except block

```

If you use **/EHsc** to compile the code and **CPPEX** is defined by using **/DCPPEX** (so that a C++ exception is thrown), the **TestClass** destructor executes and the output looks like this:

```

Throwing C++ exception
Destroying TestClass!
Executing SEH __except block

```

If you use **/EHa** to compile the code, the **TestClass** destructor executes regardless of whether the exception was thrown by using **std::throw** or by using SEH to trigger the exception, that is, whether **CPPEX** defined or not. The output looks like this:

```

Throwing C++ exception
Destroying TestClass!
Executing SEH __except block

```

For more information, see [/EH \(Exception Handling Model\)](#).

**END Microsoft-specific**

## See also

[Exception Handling](#)

[Keywords](#)

[<exception>](#)

[Errors and Exception Handling](#)

[Structured Exception Handling \(Windows\)](#)



# Writing an Exception Handler

---

Exception handlers are typically used to respond to specific errors. You can use the exception-handling syntax to filter out all exceptions other than those you know how to handle. Other exceptions should be passed to other handlers (possibly in the run-time library or the operating system) written to look for those specific exceptions.

Exception handlers use the try-except statement.

## What do you want to know more about?

- [The try-except statement](#)
- [Writing an exception filter](#)
- [Raising software exceptions](#)
- [Hardware exceptions](#)
- [Restrictions on exception handlers](#)

## See also

[Structured Exception Handling \(C/C++\)](#)

# try-except Statement

---

## Microsoft Specific

The **try-except** statement is a Microsoft extension to the C and C++ languages that supports structured exception handling.

## Syntax

```
__try
{
    // guarded code
}
__except ( expression )
{
    // exception handler code
}
```

## Remarks

The **try-except** statement is a Microsoft extension to the C and C++ languages that enables target applications to gain control when events that normally terminate program execution occur. Such events are called *exceptions*, and the mechanism that deals with exceptions is called *structured exception handling* (SEH).

For related information, see the [try-finally statement](#).

Exceptions can be either hardware-based or software-based. Even when applications cannot completely recover from hardware or software exceptions, structured exception handling makes it possible to display error information and trap the internal state of the application to help diagnose the problem. This is especially useful for intermittent problems that cannot be reproduced easily.

[!NOTE] Structured exception handling works with Win32 for both C and C++ source files. However, it is not specifically designed for C++. You can ensure that your code is more portable by using C++ exception handling. Also, C++ exception handling is more flexible, in that it can handle exceptions of any type. For C++ programs, it is recommended that you use the C++ exception-handling mechanism ([try](#), [catch](#), and [throw](#) statements).

The compound statement after the **\_\_try** clause is the body or guarded section. The compound statement after the **\_\_except** clause is the exception handler. The handler specifies a set of actions to be taken if an exception is raised during execution of the body of the guarded section. Execution proceeds as follows:

1. The guarded section is executed.
2. If no exception occurs during execution of the guarded section, execution continues at the statement after the **\_\_except** clause.
3. If an exception occurs during execution of the guarded section or in any routine the guarded section calls, the **\_\_except expression** (called the *filter expression*) is evaluated and the value determines how

the exception is handled. There are three possible values:

- EXCEPTION\_CONTINUE\_EXECUTION (-1) Exception is dismissed. Continue execution at the point where the exception occurred.
- EXCEPTION\_CONTINUE\_SEARCH (0) Exception is not recognized. Continue to search up the stack for a handler, first for containing **try-except** statements, then for handlers with the next highest precedence.
- EXCEPTION\_EXECUTE\_HANDLER (1) Exception is recognized. Transfer control to the exception handler by executing the **\_\_except** compound statement, then continue execution after the **\_\_except** block.

Because the **\_\_except** expression is evaluated as a C expression, it is limited to a single value, the conditional-expression operator, or the comma operator. If more extensive processing is required, the expression can call a routine that returns one of the three values listed above.

Each application can have its own exception handler.

It is not valid to jump into a **\_\_try** statement, but valid to jump out of one. The exception handler is not called if a process is terminated in the middle of executing a **try-except** statement.

For compatibility with previous versions, **\_\_try**, **\_\_except**, and **\_\_leave** are synonyms for **\_\_try**, **\_\_except**, and **\_\_leave** unless compiler option [/Za \(Disable language extensions\)](#) is specified.

### The **\_\_leave** Keyword

The **\_\_leave** keyword is valid only within the guarded section of a **try-except** statement, and its effect is to jump to the end of the guarded section. Execution continues at the first statement after the exception handler.

A **goto** statement can also jump out of the guarded section, and it does not degrade performance as it does in a **try-finally** statement because stack unwinding does not occur. However, we recommend that you use the **\_\_leave** keyword rather than a **goto** statement because you are less likely to make a programming mistake if the guarded section is large or complex.

### Structured Exception Handling Intrinsic Functions

Structured exception handling provides two intrinsic functions that are available to use with the **try-except** statement: **GetExceptionCode** and **GetExceptionInformation**.

**GetExceptionCode** returns the code (a 32-bit integer) of the exception.

The intrinsic function **GetExceptionInformation** returns a pointer to a structure containing additional information about the exception. Through this pointer, you can access the machine state that existed at the time of a hardware exception. The structure is as follows:

```
typedef struct _EXCEPTION_POINTERS {
    PEXCEPTION_RECORD ExceptionRecord;
    PCONTEXT ContextRecord;
} EXCEPTION_POINTERS, *PEXCEPTION_POINTERS;
```

The pointer types `PEXCEPTION_RECORD` and `PCONTEXT` are defined in the include file `<winnt.h>`, and `_EXCEPTION_RECORD` and `_CONTEXT` are defined in the include file `<excpt.h>`

You can use `GetExceptionCode` within the exception handler. However, you can use `GetExceptionInformation` only within the exception filter expression. The information it points to is generally on the stack and is no longer available when control is transferred to the exception handler.

The intrinsic function `AbnormalTermination` is available within a termination handler. It returns 0 if the body of the **try-finally** statement terminates sequentially. In all other cases, it returns 1.

`excpt.h` defines some alternate names for these intrinsics:

`GetExceptionCode` is equivalent to `_exception_code`

`GetExceptionInformation` is equivalent to `_exception_info`

`AbnormalTermination` is equivalent to `_abnormal_termination`

## Example

```
// exceptions_try_except_Statement.cpp
// Example of try-except and try-finally statements
#include <stdio.h>
#include <windows.h> // for EXCEPTION_ACCESS_VIOLATION
#include <excpt.h>

int filter(unsigned int code, struct _EXCEPTION_POINTERS *ep)
{
    puts("in filter.");
    if (code == EXCEPTION_ACCESS_VIOLATION)
    {
        puts("caught AV as expected.");
        return EXCEPTION_EXECUTE_HANDLER;
    }
    else
    {
        puts("didn't catch AV, unexpected.");
        return EXCEPTION_CONTINUE_SEARCH;
    }
};

int main()
{
    int* p = 0x00000000; // pointer to NULL
    puts("hello");
    __try
    {
        puts("in try");
        __try
        {
            puts("in try");
            *p = 13; // causes an access violation exception;
        }
    }
}
```

```

    }
    __finally
    {
        puts("in finally. termination: ");
        puts(AbnormalTermination() ? "\tabnormal" : "\tnormal");
    }
}
__except(filter(GetExceptionCode(), GetExceptionInformation()))
{
    puts("in except");
}
puts("world");
}

```

## Output

```

hello
in try
in try
in filter.
caught AV as expected.
in finally. termination:
    abnormal
in except
world

```

## END Microsoft Specific

## See also

[Writing an exception handler](#)

[Structured Exception Handling \(C/C++\)](#)

[Keywords](#)

# Writing an exception filter

You can handle an exception either by jumping to the level of the exception handler or by continuing execution. Instead of using the exception handler code to handle the exception and falling through, you can use *filter* to clean up the problem and then, by returning -1, resume normal flow without clearing the stack.

[!NOTE] Some exceptions cannot be continued. If *filter* evaluates to -1 for such an exception, the system raises a new exception. When you call [RaiseException](#), you determine whether the exception will continue.

For example, the following code uses a function call in the *filter* expression: this function handles the problem and then returns -1 to resume normal flow of control:

```
// exceptions_Writing_an_Exception_Filter.cpp
#include <windows.h>
int main() {
    int Eval_Exception( int );

    __try {}

    __except ( Eval_Exception( GetExceptionCode( ))) {
        ;
    }

}

void ResetVars( int ) {}
int Eval_Exception ( int n_except ) {
    if ( n_except != STATUS_INTEGER_OVERFLOW &&
        n_except != STATUS_FLOAT_OVERFLOW )    // Pass on most exceptions
        return EXCEPTION_CONTINUE_SEARCH;

    // Execute some code to clean up problem
    ResetVars( 0 );    // initializes data to 0
    return EXCEPTION_CONTINUE_EXECUTION;
}
```

It is a good idea to use a function call in the *filter* expression whenever *filter* needs to do anything complex. Evaluating the expression causes execution of the function, in this case, [Eval\\_Exception](#).

Note the use of [GetExceptionCode](#) to determine the exception. You must call this function inside the filter itself. [Eval\\_Exception](#) cannot call [GetExceptionCode](#), but it must have the exception code passed to it.

This handler passes control to another handler unless the exception is an integer or floating-point overflow. If it is, the handler calls a function ([ResetVars](#) is only an example, not an API function) to reset some global variables. *Statement-block-2*, which in this example is empty, can never be executed because [Eval\\_Exception](#) never returns `EXCEPTION_EXECUTE_HANDLER` (1).

Using a function call is a good general-purpose technique for dealing with complex filter expressions. Two other C language features that are useful are:

- The conditional operator
- The comma operator

The conditional operator is frequently useful, because it can be used to check for a specific return code and then return one of two different values. For example, the filter in the following code recognizes the exception only if the exception is `STATUS_INTEGER_OVERFLOW`:

```
__except( GetExceptionCode() == STATUS_INTEGER_OVERFLOW ? 1 : 0 ) {
```

The purpose of the conditional operator in this case is mainly to provide clarity, because the following code produces the same results:

```
__except( GetExceptionCode() == STATUS_INTEGER_OVERFLOW ) {
```

The conditional operator is more useful in situations where you might want the filter to evaluate to -1, `EXCEPTION_CONTINUE_EXECUTION`.

The comma operator enables you to perform multiple, independent operations inside a single expression. The effect is roughly that of executing multiple statements and then returning the value of the last expression. For example, the following code stores the exception code in a variable and then tests it:

```
__except( nCode = GetExceptionCode(), nCode == STATUS_INTEGER_OVERFLOW )
```

## See also

[Writing an exception handler](#)

[Structured Exception Handling \(C/C++\)](#)

# Raising software exceptions

Some of the most common sources of program errors are not flagged as exceptions by the system. For example, if you attempt to allocate a memory block but there is insufficient memory, the run-time or API function does not raise an exception but returns an error code.

However, you can treat any condition as an exception by detecting that condition in your code and then reporting it by calling the `RaiseException` function. By flagging errors this way, you can bring the advantages of structured exception handling to any kind of run-time error.

To use structured exception handling with errors:

- Define your own exception code for the event.
- Call `RaiseException` when you detect a problem.
- Use exception-handling filters to test for the exception code you defined.

The `<winerror.h>` file shows the format for exception codes. To make sure that you do not define a code that conflicts with an existing exception code, set the third most significant bit to 1. The four most-significant bits should be set as shown in the following table.

Bits	Recommended binary setting	Description
31-30	11	These two bits describe the basic status of the code: 11 = error, 00 = success, 01 = informational, 10 = warning.
29	1	Client bit. Set to 1 for user-defined codes.
28	0	Reserved bit. (Leave set to 0.)

You can set the first two bits to a setting other than 11 binary if you want, although the "error" setting is appropriate for most exceptions. The important thing to remember is to set bits 29 and 28 as shown in the previous table.

The resulting error code should therefore have the highest four bits set to hexadecimal E. For example, the following definitions define exception codes that do not conflict with any Windows exception codes. (You may, however, need to check which codes are used by third-party DLLs.)

```
#define STATUS_INSUFFICIENT_MEM      0xE0000001
#define STATUS_FILE_BAD_FORMAT      0xE0000002
```

After you have defined an exception code, you can use it to raise an exception. For example, the following code raises the `STATUS_INSUFFICIENT_MEM` exception in response to a memory allocation problem:



```
lpstr = _malloc( nBufferSize );  
if (lpstr == NULL)  
    RaiseException( STATUS_INSUFFICIENT_MEM, 0, 0, 0 );
```

If you want to simply raise an exception, you can set the last three parameters to 0. The three last parameters are useful for passing additional information and setting a flag that prevents handlers from continuing execution. See the [RaiseException](#) function in the Windows SDK for more information.

In your exception-handling filters, you can then test for the codes you've defined. For example:

```
__try {  
    ...  
}  
__except (GetExceptionCode() == STATUS_INSUFFICIENT_MEM ||  
         GetExceptionCode() == STATUS_FILE_BAD_FORMAT )
```

## See also

[Writing an exception handler](#)

[Structured exception handling \(C/C++\)](#)

# Hardware exceptions

---

Most of the standard exceptions recognized by the operating system are hardware-defined exceptions. Windows recognizes a few low-level software exceptions, but these are usually best handled by the operating system.

Windows maps the hardware errors of different processors to the exception codes in this section. In some cases, a processor may generate only a subset of these exceptions. Windows preprocesses information about the exception and issues the appropriate exception code.

The hardware exceptions recognized by Windows are summarized in the following table:

Exception code	Cause of exception
STATUS_ACCESS_VIOLATION	Reading or writing to an inaccessible memory location.
STATUS_BREAKPOINT	Encountering a hardware-defined breakpoint; used only by debuggers.
STATUS_DATATYPE_MISALIGNMENT	Reading or writing to data at an address that is not properly aligned; for example, 16-bit entities must be aligned on 2-byte boundaries. (Not applicable to Intel 80x86 processors.)
STATUS_FLOAT_DIVIDE_BY_ZERO	Dividing floating-point type by 0.0.
STATUS_FLOAT_OVERFLOW	Exceeding maximum positive exponent of floating-point type.
STATUS_FLOAT_UNDERFLOW	Exceeding magnitude of lowest negative exponent of floating-point type.
STATUS_FLOATING_RESEVERED_OPERAND	Using a reserved floating-point format (invalid use of format).
STATUS_ILLEGAL_INSTRUCTION	Attempting to execute an instruction code not defined by the processor.
STATUS_PRIVILEGED_INSTRUCTION	Executing an instruction not allowed in current machine mode.
STATUS_INTEGER_DIVIDE_BY_ZERO	Dividing an integer type by 0.
STATUS_INTEGER_OVERFLOW	Attempting an operation that exceeds the range of the integer.
STATUS_SINGLE_STEP	Executing one instruction in single-step mode; used only by debuggers.

Many of the exceptions listed in the previous table are intended to be handled by debuggers, the operating system, or other low-level code. With the exception of integer and floating-point errors, your code should not handle these errors. Thus, you should usually use the exception-handling filter to ignore exceptions (evaluate to 0). Otherwise, you may prevent lower-level mechanisms from responding appropriately. You can, however,

take appropriate precautions against the potential effect of these low-level errors by [writing termination handlers](#).

## See also

[Writing an exception handler](#)

[Structured Exception Handling \(C/C++\)](#)

# Restrictions on exception handlers

---

The principal limitation to using exception handlers in code is that you cannot use a **goto** statement to jump into a **\_\_try** statement block. Instead, you must enter the statement block through normal flow of control. You can jump out of a **\_\_try** statement block and nest exception handlers as you choose.

## See also

[Writing an Exception Handler](#)

[Structured Exception Handling \(C/C++\)](#)

# Writing a Termination Handler

---

Unlike an exception handler, a termination handler is always executed, regardless of whether the protected block of code terminated normally. The sole purpose of the termination handler should be to ensure that resources, such as memory, handles, and files, are properly closed regardless of how a section of code finishes executing.

Termination handlers use the try-finally statement.

## What do you want to know more about?

- [The try-finally statement](#)
- [Cleaning up resources](#)
- [Timing of actions in exception handling](#)
- [Restrictions on termination handlers](#)

## See also

[Structured Exception Handling \(C/C++\)](#)

# try-finally Statement

---

## Microsoft Specific

The following syntax describes the **try-finally** statement:

```
__try
{
    // guarded code
}
__finally
{
    // termination code
}
```

## Grammar

*try-finally-statement:*

**\_\_try** *compound-statement* **\_\_finally** *compound-statement*

The **try-finally** statement is a Microsoft extension to the C and C++ languages that enables target applications to guarantee execution of cleanup code when execution of a block of code is interrupted. Cleanup consists of such tasks as deallocating memory, closing files, and releasing file handles. The **try-finally** statement is especially useful for routines that have several places where a check is made for an error that could cause premature return from the routine.

For related information and a code sample, see [try-except Statement](#). For more information on structured exception handling in general, see [Structured Exception Handling](#). For more information on handling exceptions in managed applications with C++/CLI, see [Exception Handling under /clr](#).

[!NOTE] Structured exception handling works with Win32 for both C and C++ source files. However, it is not specifically designed for C++. You can ensure that your code is more portable by using C++ exception handling. Also, C++ exception handling is more flexible, in that it can handle exceptions of any type. For C++ programs, it is recommended that you use the C++ exception-handling mechanism ([try](#), [catch](#), and [throw](#) statements).

The compound statement after the **\_\_try** clause is the guarded section. The compound statement after the **\_\_finally** clause is the termination handler. The handler specifies a set of actions that execute when the guarded section is exited, regardless of whether the guarded section is exited by an exception (abnormal termination), or by standard fall through (normal termination).

Control reaches a **\_\_try** statement by simple sequential execution (fall through). When control enters the **\_\_try**, its associated handler becomes active. If the flow of control reaches the end of the try block, execution proceeds as follows:

1. The termination handler is invoked.

2. When the termination handler completes, execution continues after the **\_\_finally** statement. Regardless of how the guarded section ends (for example, via a **goto** out of the guarded body or a **return** statement), the termination handler is executed *before* the flow of control moves out of the guarded section.

A **\_\_finally** statement does not block searching for an appropriate exception handler.

If an exception occurs in the **\_\_try** block, the operating system must find a handler for the exception or the program will fail. If a handler is found, any and all **\_\_finally** blocks are executed and execution resumes in the handler.

For example, suppose a series of function calls links function A to function D, as shown in the following figure. Each function has one termination handler. If an exception is raised in function D and handled in A, the termination handlers are called in this order as the system unwinds the stack: D, C, B.



Order of termination-handler execution

Order of Termination-Handler Execution

[!NOTE] The behavior of try-finally is different from some other languages that support the use of **finally**, such as C#. A single **\_\_try** may have either, but not both, of **\_\_finally** and **\_\_except**. If both are to be used together, an outer try-except statement must enclose the inner try-finally statement. The rules specifying when each block executes are also different.

For compatibility with previous versions, **\_try**, **\_finally**, and **\_leave** are synonyms for **\_\_try**, **\_\_finally**, and **\_\_leave** unless compiler option [/Za \(Disable language extensions\)](#) is specified.

## The **\_\_leave** Keyword

The **\_\_leave** keyword is valid only within the guarded section of a **try-finally** statement, and its effect is to jump to the end of the guarded section. Execution continues at the first statement in the termination handler.

A **goto** statement can also jump out of the guarded section, but it degrades performance because it invokes stack unwinding. The **\_\_leave** statement is more efficient because it does not cause stack unwinding.

## Abnormal Termination

Exiting a **try-finally** statement using the [longjmp](#) run-time function is considered abnormal termination. It is illegal to jump into a **\_\_try** statement, but legal to jump out of one. All **\_\_finally** statements that are active between the point of departure (normal termination of the **\_\_try** block) and the destination (the **\_\_except** block that handles the exception) must be run. This is called a local unwind.

If a **try** block is prematurely terminated for any reason, including a jump out of the block, the system executes the associated **finally** block as a part of the process of unwinding the stack. In such cases, the [AbnormalTermination](#) function returns **true** if called from within the **finally** block; otherwise, it returns **false**.

The termination handler is not called if a process is killed in the middle of executing a **try-finally** statement.

**END Microsoft Specific**

See also

Writing a termination handler

Structured Exception Handling (C/C++)

Keywords

Termination-Handler Syntax



# Cleaning up Resources

---

During termination-handler execution, you may not know which resources are actually allocated before the termination handler was called. It is possible that the **\_\_try** statement block was interrupted before all resources were allocated, so that not all resources were opened.

Therefore, to be safe, you should check to see which resources are actually open before proceeding with termination-handling cleanup. A recommended procedure is to:

1. Initialize handles to NULL.
2. In the **\_\_try** statement block, allocate resources. Handles are set to positive values as the resource is allocated.
3. In the **\_\_finally** statement block, release each resource whose corresponding handle or flag variable is nonzero or not NULL.

## Example

For example, the following code uses a termination handler to close three files and a memory block that were allocated in the **\_\_try** statement block. Before cleaning up a resource, the code first checks to see if the resource was allocated.

```
// exceptions_Cleaning_up_Resources.cpp
#include <stdlib.h>
#include <malloc.h>
#include <stdio.h>
#include <windows.h>

void fileOps() {
    FILE  *fp1 = NULL,
          *fp2 = NULL,
          *fp3 = NULL;
    LPVOID lpvoid = NULL;
    errno_t err;

    __try {
        lpvoid = malloc( BUFSIZ );

        err = fopen_s(&fp1, "ADDRESS.DAT", "w+" );
        err = fopen_s(&fp2, "NAMES.DAT", "w+" );
        err = fopen_s(&fp3, "CARS.DAT", "w+" );
    }
    __finally {
        if ( fp1 )
            fclose( fp1 );
        if ( fp2 )
            fclose( fp2 );
        if ( fp3 )
```

```
        fclose( fp3 );
        if ( lpvoid )
            free( lpvoid );
    }
}

int main() {
    fileOps();
}
```

## See also

[Writing a termination handler](#)

[Structured Exception Handling \(C/C++\)](#)

# Timing of exception handling: A summary

---

A termination handler is executed no matter how the `__try` statement block is terminated. Causes include jumping out of the `__try` block, a `longjmp` statement that transfers control out of the block, and unwinding the stack due to exception handling.

[!NOTE] The Microsoft C++ compiler supports two forms of the `setjmp` and `longjmp` statements. The fast version bypasses termination handling but is more efficient. To use this version, include the file `<setjmp.h>`. The other version supports termination handling as described in the previous paragraph. To use this version, include the file `<setjmpex.h>`. The increase in performance of the fast version depends on hardware configuration.

The operating system executes all termination handlers in the proper order before any other code can be executed, including the body of an exception handler.

When the cause for interruption is an exception, the system must first execute the filter portion of one or more exception handlers before deciding what to terminate. The order of events is:

1. An exception is raised.
2. The system looks at the hierarchy of active exception handlers and executes the filter of the handler with highest precedence; this is the exception handler most recently installed and most deeply nested, in terms of blocks and function calls.
3. If this filter passes control (returns 0), the process continues until a filter is found that does not pass control.
4. If this filter returns -1, execution continues where the exception was raised, and no termination takes place.
5. If the filter returns 1, the following events occur:
  - The system unwinds the stack, clearing all stack frames between the currently executing code (where the exception was raised) and the stack frame that contains the exception handler gaining control.
  - As the stack is unwound, each termination handler on the stack is executed.
  - The exception handler itself is executed.
  - Control passes to the line of code after the end of this exception handler.

## See also

[Writing a termination handler](#)

[Structured Exception Handling \(C/C++\)](#)

# Restrictions on Termination Handlers

---

You cannot use a **goto** statement to jump into a **\_\_try** statement block or a **\_\_finally** statement block. Instead, you must enter the statement block through normal flow of control. (You can, however, jump out of a **\_\_try** statement block.) Also, you cannot nest an exception handler or termination handler inside a **\_\_finally** block.

In addition, some kinds of code permitted in a termination handler produce questionable results, so you should use them with caution, if at all. One is a **goto** statement that jumps out of a **\_\_finally** statement block. If the block is executing as part of normal termination, nothing unusual happens. But if the system is unwinding the stack, that unwinding stops, and the current function gains control as if there were no abnormal termination.

A **return** statement inside a **\_\_finally** statement block presents roughly the same situation. Control returns to the immediate caller of the function containing the termination handler. If the system was unwinding the stack, this process is halted, and the program proceeds as if there had been no exception raised.

## See also

[Writing a termination handler](#)

[Structured Exception Handling \(C/C++\)](#)

# Transporting exceptions between threads

The Microsoft C++ compiler (MSVC) supports *transporting an exception* from one thread to another. Transporting exceptions enables you to catch an exception in one thread and then make the exception appear to be thrown in a different thread. For example, you can use this feature to write a multithreaded application where the primary thread handles all the exceptions thrown by its secondary threads. Transporting exceptions is useful mostly to developers who create parallel programming libraries or systems. To implement transporting exceptions, MSVC provides the `exception_ptr` type and the `current_exception`, `rethrow_exception`, and `make_exception_ptr` functions.

## Syntax

```
namespace std
{
    typedef unspecified exception_ptr;
    exception_ptr current_exception();
    void rethrow_exception(exception_ptr p);
    template<class E>
        exception_ptr make_exception_ptr(E e) noexcept;
}
```

## Parameters

Parameter	Description
<i>unspecified</i>	An unspecified internal class that is used to implement the <code>exception_ptr</code> type.
<i>p</i>	An <code>exception_ptr</code> object that references an exception.
<i>E</i>	A class that represents an exception.
<i>e</i>	An instance of the parameter <code>E</code> class.

## Return value

The `current_exception` function returns an `exception_ptr` object that references the exception that is currently in progress. If no exception is in progress, the function returns an `exception_ptr` object that is not associated with any exception.

The `make_exception_ptr` function returns an `exception_ptr` object that references the exception specified by the `e` parameter.

## Remarks

### Scenario

Imagine that you want to create an application that can scale to handle a variable amount of work. To achieve this objective, you design a multithreaded application where an initial, primary thread creates as many secondary threads as it needs in order to do the job. The secondary threads help the primary thread to manage resources, to balance loads, and to improve throughput. By distributing the work, the multithreaded application performs better than a single-threaded application.

However, if a secondary thread throws an exception, you want the primary thread to handle it. This is because you want your application to handle exceptions in a consistent, unified manner regardless of the number of secondary threads.

## Solution

To handle the previous scenario, the C++ Standard supports transporting an exception between threads. If a secondary thread throws an exception, that exception becomes the *current exception*. By analogy to the real world, the current exception is said to be *in flight*. The current exception is in flight from the time it is thrown until the exception handler that catches it returns.

The secondary thread can catch the current exception in a **catch** block, and then call the `current_exception` function to store the exception in an `exception_ptr` object. The `exception_ptr` object must be available to the secondary thread and to the primary thread. For example, the `exception_ptr` object can be a global variable whose access is controlled by a mutex. The term *transport an exception* means an exception in one thread can be converted to a form that can be accessed by another thread.

Next, the primary thread calls the `rethrow_exception` function, which extracts and then throws the exception from the `exception_ptr` object. When the exception is thrown, it becomes the current exception in the primary thread. That is, the exception appears to originate in the primary thread.

Finally, the primary thread can catch the current exception in a **catch** block and then process it or throw it to a higher level exception handler. Or, the primary thread can ignore the exception and allow the process to end.

Most applications do not have to transport exceptions between threads. However, this feature is useful in a parallel computing system because the system can divide work among secondary threads, processors, or cores. In a parallel computing environment, a single, dedicated thread can handle all the exceptions from the secondary threads and can present a consistent exception-handling model to any application.

For more information about the C++ Standards committee proposal, search the Internet for document number N2179, titled "Language Support for Transporting Exceptions between Threads".

## Exception-handling models and compiler options

Your application's exception-handling model determines whether it can catch and transport an exception. Visual C++ supports three models that can handle C++ exceptions, structured exception handling (SEH) exceptions, and common language runtime (CLR) exceptions. Use the `/EH` and `/clr` compiler options to specify your application's exception-handling model.

Only the following combination of compiler options and programming statements can transport an exception. Other combinations either cannot catch exceptions, or can catch but cannot transport exceptions.

- The `/EHa` compiler option and the **catch** statement can transport SEH and C++ exceptions.
- The `/EHa`, `/EHs`, and `/EHsc` compiler options and the **catch** statement can transport C++ exceptions.

- The **/CLR** compiler option and the **catch** statement can transport C++ exceptions. The **/CLR** compiler option implies specification of the **/EHa** option. Note that the compiler does not support transporting managed exceptions. This is because managed exceptions, which are derived from the [System.Exception class](#), are already objects that you can move between threads by using the facilities of the common language runtime.

[!IMPORTANT] We recommend that you specify the **/EHsc** compiler option and catch only C++ exceptions. You expose yourself to a security threat if you use the **/EHa** or **/CLR** compiler option and a **catch** statement with an ellipsis *exception-declaration* (**catch(...)**). You probably intend to use the **catch** statement to capture a few specific exceptions. However, the **catch(...)** statement captures all C++ and SEH exceptions, including unexpected ones that should be fatal. If you ignore or mishandle an unexpected exception, malicious code can use that opportunity to undermine the security of your program.

## Usage

The following sections describe how to transport exceptions by using the [exception\\_ptr](#) type, and the [current\\_exception](#), [rethrow\\_exception](#), and [make\\_exception\\_ptr](#) functions.

### exception\_ptr type

Use an [exception\\_ptr](#) object to reference the current exception or an instance of a user-specified exception. In the Microsoft implementation, an exception is represented by an [EXCEPTION\\_RECORD](#) structure. Each [exception\\_ptr](#) object includes an exception reference field that points to a copy of the [EXCEPTION\\_RECORD](#) structure that represents the exception.

When you declare an [exception\\_ptr](#) variable, the variable is not associated with any exception. That is, its exception reference field is NULL. Such an [exception\\_ptr](#) object is called a *null exception\_ptr*.

Use the [current\\_exception](#) or [make\\_exception\\_ptr](#) function to assign an exception to an [exception\\_ptr](#) object. When you assign an exception to an [exception\\_ptr](#) variable, the variable's exception reference field points to a copy of the exception. If there is insufficient memory to copy the exception, the exception reference field points to a copy of a [std::bad\\_alloc](#) exception. If the [current\\_exception](#) or [make\\_exception\\_ptr](#) function cannot copy the exception for any other reason, the function calls the [terminate](#) function to exit the current process.

Despite its name, an [exception\\_ptr](#) object is not itself a pointer. It does not obey pointer semantics and cannot be used with the pointer member access ([->](#)) or indirection ([\\*](#)) operators. The [exception\\_ptr](#) object has no public data members or member functions.

### Comparisons

You can use the equal ([==](#)) and not-equal ([!=](#)) operators to compare two [exception\\_ptr](#) objects. The operators do not compare the binary value (bit pattern) of the [EXCEPTION\\_RECORD](#) structures that represent the exceptions. Instead, the operators compare the addresses in the exception reference field of the [exception\\_ptr](#) objects. Consequently, a null [exception\\_ptr](#) and the NULL value compare as equal.

### current\_exception function

Call the `current_exception` function in a **catch** block. If an exception is in flight and the **catch** block can catch the exception, the `current_exception` function returns an `exception_ptr` object that references the exception. Otherwise, the function returns a null `exception_ptr` object.

## Details

The `current_exception` function captures the exception that is in flight regardless of whether the **catch** statement specifies an `exception-declaration` statement.

The destructor for the current exception is called at the end of the **catch** block if you do not rethrow the exception. However, even if you call the `current_exception` function in the destructor, the function returns an `exception_ptr` object that references the current exception.

Successive calls to the `current_exception` function return `exception_ptr` objects that refer to different copies of the current exception. Consequently, the objects compare as unequal because they refer to different copies, even though the copies have the same binary value.

## SEH exceptions

If you use the `/EHa` compiler option, you can catch an SEH exception in a C++ **catch** block. The `current_exception` function returns an `exception_ptr` object that references the SEH exception. And the `rethrow_exception` function throws the SEH exception if you call it with the transported `exception_ptr` object as its argument.

The `current_exception` function returns a null `exception_ptr` if you call it in an SEH **\_\_finally** termination handler, an **\_\_except** exception handler, or the **\_\_except** filter expression.

A transported exception does not support nested exceptions. A nested exception occurs if another exception is thrown while an exception is being handled. If you catch a nested exception, the `EXCEPTION_RECORD.ExceptionRecord` data member points to a chain of `EXCEPTION_RECORD` structures that describe the associated exceptions. The `current_exception` function does not support nested exceptions because it returns an `exception_ptr` object whose `ExceptionRecord` data member is zeroed out.

If you catch an SEH exception, you must manage the memory referenced by any pointer in the `EXCEPTION_RECORD.ExceptionInformation` data member array. You must guarantee that the memory is valid during the lifetime of the corresponding `exception_ptr` object, and that the memory is freed when the `exception_ptr` object is deleted.

You can use structured exception (SE) translator functions together with the transport exceptions feature. If an SEH exception is translated to a C++ exception, the `current_exception` function returns an `exception_ptr` that references the translated exception instead of the original SEH exception. The `rethrow_exception` function subsequently throws the translated exception, not the original exception. For more information about SE translator functions, see [\\_set\\_se\\_translator](#).

## rethrow\_exception function

After you store a caught exception in an `exception_ptr` object, the primary thread can process the object. In your primary thread, call the `rethrow_exception` function together with the `exception_ptr` object as its argument. The `rethrow_exception` function extracts the exception from the `exception_ptr` object and



then throws the exception in the context of the primary thread. If the *p* parameter of the `rethrow_exception` function is a null `exception_ptr`, the function throws `std::bad_exception`.

The extracted exception is now the current exception in the primary thread, and you can handle it as you would any other exception. If you catch the exception, you can handle it immediately or use a **throw** statement to send it to a higher level exception handler. Otherwise, do nothing and let the default system exception handler terminate your process.

## make\_exception\_ptr function

The `make_exception_ptr` function takes an instance of a class as its argument and then returns an `exception_ptr` that references the instance. Usually, you specify an `exception class` object as the argument to the `make_exception_ptr` function, although any class object can be the argument.

Calling the `make_exception_ptr` function is equivalent to throwing a C++ exception, catching it in a **catch** block, and then calling the `current_exception` function to return an `exception_ptr` object that references the exception. The Microsoft implementation of the `make_exception_ptr` function is more efficient than throwing and then catching an exception.

An application typically does not require the `make_exception_ptr` function, and we discourage its use.

## Example

The following example transports a standard C++ exception and a custom C++ exception from one thread to another.

```
// transport_exception.cpp
// compile with: /EHsc /MD
#include <windows.h>
#include <stdio.h>
#include <exception>
#include <stdexcept>

using namespace std;

// Define thread-specific information.
#define THREADCOUNT 2
exception_ptr aException[THREADCOUNT];
int          aArg[THREADCOUNT];

DWORD WINAPI ThrowExceptions( LPVOID );

// Specify a user-defined, custom exception.
// As a best practice, derive your exception
// directly or indirectly from std::exception.
class myException : public std::exception {
};
int main()
{
    HANDLE aThread[THREADCOUNT];
    DWORD ThreadID;
```

```

// Create secondary threads.
for( int i=0; i < THREADCOUNT; i++ )
{
    aArg[i] = i;
    aThread[i] = CreateThread(
        NULL,          // Default security attributes.
        0,             // Default stack size.
        (LPTHREAD_START_ROUTINE) ThrowExceptions,
        (LPVOID) &aArg[i], // Thread function argument.
        0,             // Default creation flags.
        &ThreadID); // Receives thread identifier.
    if( aThread[i] == NULL )
    {
        printf("CreateThread error: %d\n", GetLastError());
        return -1;
    }
}

// Wait for all threads to terminate.
WaitForMultipleObjects(THREADCOUNT, aThread, TRUE, INFINITE);
// Close thread handles.
for( int i=0; i < THREADCOUNT; i++ ) {
    CloseHandle(aThread[i]);
}

// Rethrow and catch the transported exceptions.
for ( int i = 0; i < THREADCOUNT; i++ ) {
    try {
        if (aException[i] == NULL) {
            printf("exception_ptr %d: No exception was transported.\n", i);
        }
        else {
            rethrow_exception( aException[i] );
        }
    }
    catch( const invalid_argument & ) {
        printf("exception_ptr %d: Caught an invalid_argument exception.\n",
i);
    }
    catch( const myException & ) {
        printf("exception_ptr %d: Caught a myException exception.\n", i);
    }
}

// Each thread throws an exception depending on its thread
// function argument, and then ends.
DWORD WINAPI ThrowExceptions( LPVOID lpParam )
{
    int x = *((int*)lpParam);
    if (x == 0) {
        try {
            // Standard C++ exception.
            // This example explicitly throws invalid_argument exception.

```

```

        // In practice, your application performs an operation that
        // implicitly throws an exception.
        throw invalid_argument("A C++ exception.");
    }
    catch ( const invalid_argument & ) {
        aException[x] = current_exception();
    }
}
else {
    // User-defined exception.
    aException[x] = make_exception_ptr( myException() );
}
return TRUE;
}

```

```

exception_ptr 0: Caught an invalid_argument exception.
exception_ptr 1: Caught a myException exception.

```

## Requirements

**Header:** <exception>

## See also

[Exception Handling](#)

[/EH \(Exception Handling Model\)](#)

[/clr \(Common Language Runtime Compilation\)](#)

# Assertion and User-Supplied Messages (C++)

---

The C++ language supports three error handling mechanisms that help you debug your application: the [#error directive](#), the [static\\_assert](#) keyword, and the [assert Macro](#), [\\_assert](#), [\\_wassert](#) macro. All three mechanisms issue error messages, and two also test software assertions. A software assertion specifies a condition that you expect to be true at a particular point in your program. If a compile time assertion fails, the compiler issues a diagnostic message and a compilation error. If a run-time assertion fails, the operating system issues a diagnostic message and closes your application.

## Remarks

The lifetime of your application consists of a preprocessing, compile, and run time phase. Each error handling mechanism accesses debug information that is available during one of these phases. To debug effectively, select the mechanism that provides appropriate information about that phase:

- The [#error directive](#) is in effect at preprocessing time. It unconditionally emits a user-specified message and causes the compilation to fail with an error. The message can contain text that is manipulated by preprocessor directives but any resulting expression is not evaluated.
- The [static\\_assert](#) declaration is in effect at compile time. It tests a software assertion that is represented by a user-specified integral expression that can be converted to a Boolean. If the expression evaluates to zero (false), the compiler issues the user-specified message and the compilation fails with an error.

The [static\\_assert](#) declaration is especially useful for debugging templates because template arguments can be included in the user-specified expression.

- The [assert Macro](#), [\\_assert](#), [\\_wassert](#) macro is in effect at run time. It evaluates a user-specified expression, and if the result is zero, the system issues a diagnostic message and closes your application. Many other macros, such as [\\_ASSERT](#) and [\\_ASSERTE](#), resemble this macro but issue different system-defined or user-defined diagnostic messages.

## See also

[#error Directive \(C/C++\)](#)

[assert Macro, \\_assert, \\_wassert](#)

[\\_ASSERT, \\_ASSERTE, \\_ASSERT\\_EXPR Macros](#)

[static\\_assert](#)

[\\_STATIC\\_ASSERT Macro](#)

[Templates](#)

# static\_assert

---

Tests a software assertion at compile time. If the specified constant expression is FALSE, the compiler displays the specified message, if one is provided, and the compilation fails with error C2338; otherwise, the declaration has no effect.

## Syntax

```
static_assert( constant-expression, string-literal );  
  
static_assert( constant-expression ); // C++17 (Visual Studio 2017 and later)
```

## Parameters

Parameter	Description
	An integral constant expression that can be converted to a Boolean.
<i>constant-expression</i>	If the evaluated expression is zero (false), the <i>string-literal</i> parameter is displayed and the compilation fails with an error. If the expression is nonzero (true), the <b>static_assert</b> declaration has no effect.
<i>string-literal</i>	An message that is displayed if the <i>constant-expression</i> parameter is zero. The message is a string of characters in the <a href="#">base character set</a> of the compiler; that is, not <a href="#">multibyte or wide characters</a> .

## Remarks

The *constant-expression* parameter of a **static\_assert** declaration represents a *software assertion*. A software assertion specifies a condition that you expect to be true at a particular point in your program. If the condition is true, the **static\_assert** declaration has no effect. If the condition is false, the assertion fails, the compiler displays the message in *string-literal* parameter, and the compilation fails with an error. In Visual Studio 2017 and later, the string-literal parameter is optional.

The **static\_assert** declaration tests a software assertion at compile time. In contrast, the [assert Macro and \\_assert and \\_wassert functions](#) test a software assertion at run time and incur a run time cost in space or time. The **static\_assert** declaration is especially useful for debugging templates because template arguments can be included in the *constant-expression* parameter.

The compiler examines the **static\_assert** declaration for syntax errors when the declaration is encountered. The compiler evaluates the *constant-expression* parameter immediately if it does not depend on a template parameter. Otherwise, the compiler evaluates the *constant-expression* parameter when the template is instantiated. Consequently, the compiler might issue a diagnostic message once when the declaration is encountered, and again when the template is instantiated.

You can use the **static\_assert** keyword at namespace, class, or block scope. (The **static\_assert** keyword is technically a declaration, even though it does not introduce new name into your program, because it can be used at namespace scope.)

## Description

In the following example, the **static\_assert** declaration has namespace scope. Because the compiler knows the size of type `void *`, the expression is evaluated immediately.

## Example

```
static_assert(sizeof(void *) == 4, "64-bit code generation is not supported.");
```

## Description

In the following example, the **static\_assert** declaration has class scope. The **static\_assert** verifies that a template parameter is a *plain old data* (POD) type. The compiler examines the **static\_assert** declaration when it is declared, but does not evaluate the *constant-expression* parameter until the `basic_string` class template is instantiated in `main()`.

## Example

```
#include <type_traits>
#include <iosfwd>
namespace std {
template <class CharT, class Traits = std::char_traits<CharT> >
class basic_string {
    static_assert(std::is_pod<CharT>::value,
                  "Template argument CharT must be a POD type in class template
basic_string");
    // ...
};
}

struct NonPOD {
    NonPOD(const NonPOD &) {}
    virtual ~NonPOD() {}
};

int main()
{
    std::basic_string<char> bs;
}
```

## Description

In the following example, the **static\_assert** declaration has block scope. The **static\_assert** verifies that the size of the VMPage structure is equal to the virtual memory pagesize of the system.

## Example

```
#include <sys/param.h> // defines PAGE_SIZE
class VMClient {
public:
    struct VMPage { // ...
        };
    int check_pagesize() {
        static_assert(sizeof(VMPage) == PAGE_SIZE,
            "Struct VMPage must be the same size as a system virtual memory page.");
        // ...
    }
    // ...
};
```

## See also

[Assertion and User-Supplied Messages \(C++\)](#)

[#error Directive \(C/C++\)](#)

[assert Macro, \\_assert, \\_wassert](#)

[Templates](#)

[ASCII Character Set](#)

[Declarations and Definitions](#)

# Overview of modules in C++

---

C++20 introduces *modules*, a modern solution for componentization of C++ libraries and programs. A module is a set of source code files that are compiled independently of the [translation units](#) that import them. Modules eliminate or greatly reduce many of the problems associated with the use of header files, and also potentially reduce compilation times. Macros, preprocessor directives, and non-exported names declared in a module are not visible and therefore have no effect on the compilation of the translation unit that imports the module. You can import modules in any order without concern for macro redefinitions. Declarations in the importing translation unit do not participate in overload resolution or name lookup in the imported module. After a module is compiled once, the results are stored in a binary file that describes all the exported types, functions and templates. That file can be processed much faster than a header file, and can be reused by the compiler every place where the module is imported in a project.

Modules can be used side by side with header files. A C++ source file can import modules and also `#include` header files. In some cases, a header file can be imported as a module rather than textually `#included` by the preprocessor. We recommend that new projects use modules rather than header files as much as possible. For larger existing projects under active development, we suggest that you experiment with converting legacy headers to modules to see whether you get a meaningful reduction in compilation times.

## Enable modules in the Microsoft C++ compiler

As of Visual Studio 2019 version 16.2, modules are not fully implemented in the Microsoft C++ compiler. You can use the modules feature to create single-partition modules and to import the Standard Library modules provided by Microsoft. To enable support for modules, compile with [/experimental:module](#) and [/std:c++latest](#). In a Visual Studio project, right-click the project node in **Solution Explorer** and choose **Properties**. Set the **Configuration** drop-down to **All Configurations**, then choose **Configuration Properties > C/C++ > Language > Enable C++ Modules (experimental)**.

A module and the code that consumes it must be compiled with the same compiler options.

## Consume the C++ Standard Library as modules

Although not specified by the C++20 standard, Microsoft enables its implementation of the C++ Standard Library to be imported as modules. By importing the C++ Standard Library as modules rather than `#including` it through header files, you can potentially speed up compilation times depending on the size of your project. The library is componentized into the following modules:

- `std.regex` provides the content of header `<regex>`
- `std.filesystem` provides the content of header `<filesystem>`
- `std.memory` provides the content of header `<memory>`
- `std.threading` provides the contents of headers `<atomic>`, `<condition_variable>`, `<future>`, `<mutex>`, `<shared_mutex>`, and `<thread>`
- `std.core` provides everything else in the C++ Standard Library

To consume these modules, just add an import declaration to the top of the source code file. For example:



```
import std.core;
import std.regex;
```

To consume the Microsoft Standard Library module, compile your program with `/EHsc` and `/MD` options.

## Basic example

The following example shows a simple module definition in a source file called **Foo.ixx**. The **.ixx** extension is required for module interface files in Visual Studio. In this example, the interface file contains the function definition as well as the declaration. However, the definitions can be also placed in one or more separate files (as shown in a later example). The **export module Foo** statement indicates that this file is the primary interface for a module called **Foo**. The **export** modifier on **f()** indicates that this function will be visible when **Foo** is imported by another program or module. Note that the module references a namespace **Bar**.

```
export module Foo;

#define ANSWER 42

namespace Bar
{
    int f_internal() {
        return ANSWER;
    }

    export int f() {
        return f_internal();
    }
}
```

The file **MyProgram.cpp** uses the **import** declaration to access the name that is exported by **Foo**. Note that the name **Bar** is visible here, but not all of its members. Also note that the macro **ANSWER** is not visible.

```
import Foo;
import std.core;

using namespace std;

int main()
{
    cout << "The result of f() is " << Bar::f() << endl; // 42
    // int i = Bar::f_internal(); // C2039
    // int j = ANSWER; //C2065
}
```

The import declaration can appear only at global scope.

## Implementing modules

You can create a module with a single interface file (.ixx) that exports names and includes implementations of all functions and types. You can also put the implementations in one or more separate implementation files, similar to how .h and .cpp files are used. The **export** keyword is used in the interface file only. An implementation file can **import** another module, but cannot **export** any names. Implementation files may be named with any extension. An interface file and the set of implementation files that back it are treated as a special kind of translation unit called a *module unit*. A name that is declared in any implementation file is automatically visible in all other files within the same module unit.

For larger modules, you can split the module into multiple module units called *partitions*. Each partition consists of an interface file backed by one or more implementation files. (As of Visual Studio 2019 version 16.2, partitions are not yet fully implemented.)

## Modules, namespaces, and argument-dependent lookup

The rules for namespaces in modules are the same as in any other code. If a declaration within a namespace is exported, the enclosing namespace (excluding non-exported members) is also implicitly exported. If a namespace is explicitly exported, all declarations within that namespace definition are exported.

When performing argument-dependent lookup for overload resolutions in the importing translation unit, the compiler considers functions which are declared in the same translation unit (including module interfaces) as where the type of the function's arguments are defined.

### Module partitions

[!NOTE] This section is provided for completeness. Partitions are not yet implemented in the Microsoft C++ compiler.

A module can be componentized into *partitions*, each consisting of an interface file and zero or more implementation files. A module partition is similar to a module, except that it shares ownership of all declarations in the entire module. All names that are exported by partition interface files are imported and re-exported by the primary interface file. A partition's name must begin with the module name followed by a colon. Declarations in any of the partitions are visible within the entire module. No special precautions are needed to avoid one-definition-rule (ODR) errors. You can declare a name (function, class, etc.) in one partition and define it in another. A partition implementation file begins like this:

```
module Foo:part1
```

and the partition interface file begins like this:

```
export module Foo:part1
```

To access declarations in another partition, a partition must import it, but it can only use the partition name, not the module name:

```
module Foo:part2;  
import :part1;
```

The primary interface unit must import and re-export all the module's interface partition files like this:

```
export import :part1  
export import :part2  
...
```

The primary interface unit can import partition implementation files, but cannot export them because those files are not allowed to export any names. This enables a module to keep implementation details internal to the module.

## Modules and header files

You can include header files in a module source file by putting the `#include` directive before the module declaration. These files are considered to be in the *global module fragment*. A module can only see the names in the *global module fragment* that are in headers it explicitly includes. The global module fragment only contains symbols that are actually used.

```
// MyModuleA.cpp  
  
#include "customlib.h"  
#include "anotherlib.h"  
  
import std.core;  
import MyModuleB;  
  
//... rest of file
```

You can use a traditional header file to control which modules are imported:

```
// MyProgram.h  
import std.core;  
#ifdef DEBUG_LOGGING  
import std.filesystem;  
#endif
```

### Imported header files

[!NOTE] This section is informational only. Legacy imports are not yet implemented in the Microsoft C++ compiler.

Some headers are sufficiently self-contained that they are allowed to be brought in using the **import** keyword. The main difference between an imported header and an imported module is that any preprocessor definitions in the header are visible in the importing program immediately after the import statement. (Preprocessor definitions in any files included by that header are *not* visible.)

```
import <vector>
import "myheader.h"
```

## See also

[module](#), [import](#), [export](#)

# module, import, export

---

The **module**, **import**, and **export** declarations are available in C++20 and require the [/experimental:module](#) compiler switch along with [/std:c++latest](#). For more information, see [Overview of modules in C++](#).

## module

Place a **module** declaration at the beginning of a module implementation file to specify that the file contents belong to the named module.

```
module ModuleA;
```

## export

Use an **export module** declaration for the module's primary interface file, which must have extension **.ixx**:

```
export module ModuleA;
```

In an interface file, use the **export** modifier on names that are intended to be part of the public interface:

```
// ModuleA.ixx

export module ModuleA;

namespace Bar
{
    export int f();
    export double d();
    double internal_f(); // not exported
}
```

Non-exported names are not visible to code that imports the module:

```
//MyProgram.cpp

import module ModuleA;

int main() {
    Bar::f(); // OK
    Bar::d(); // OK
    Bar::internal_f(); // Ill-formed: error C2065: 'internal_f': undeclared
    identifier
}
```

The **export** keyword may not appear in a module implementation file. When **export** is applied to a namespace name, all names in the namespace are exported.

## import

Use an **import** declaration to make a module's names visible in your program. The import declaration must appear after the module declaration and after any `#include` directives, but before any declarations in the file.

```
module ModuleA;

#include "custom-lib.h"
import std.core;
import std.regex;
import ModuleB;

// begin declarations here:
template <class T>
class Baz
{...};
```

## Remarks

Both **import** and **module** are treated as keywords only when they appear at the start of a logical line:

```
// OK:
module ;
module module-name
import :
import <
import "
import module-name
export module ;
export module module-name
export import :
export import <
export import "
export import module-name

// Error:
int i; module ;
```

### Microsoft Specific

In Microsoft C++, the tokens **import** and **module** are always identifiers and never keywords when they are used as arguments to a macro.

## Example

```
#define foo(...) __VA_ARGS__  
foo(  
import // Always an identifier, never a keyword  
)
```

**End Microsoft Specific**

## See Also

[Overview of modules in C++](#)

# Templates (C++)

---

Templates are the basis for generic programming in C++. As a strongly-typed language, C++ requires all variables to have a specific type, either explicitly declared by the programmer or deduced by the compiler. However, many data structures and algorithms look the same no matter what type they are operating on. Templates enable you to define the operations of a class or function, and let the user specify what concrete types those operations should work on.

## Defining and using templates

A template is a construct that generates an ordinary type or function at compile time based on arguments the user supplies for the template parameters. For example, you can define a function template like this:

```
template <typename T>
T minimum(const T& lhs, const T& rhs)
{
    return lhs < rhs ? lhs : rhs;
}
```

The above code describes a template for a generic function with a single type parameter *T*, whose return value and call parameters (lhs and rhs) are all of this type. You can name a type parameter anything you like, but by convention single upper case letters are most commonly used. *T* is a template parameter; the **typename** keyword says that this parameter is a placeholder for a type. When the function is called, the compiler will replace every instance of *T* with the concrete type argument that is either specified by the user or deduced by the compiler. The process in which the compiler generates a class or function from a template is referred to as *template instantiation*; `minimum<int>` is an instantiation of the template `minimum<T>`.

Elsewhere, a user can declare an instance of the template that is specialized for `int`. Assume that `get_a()` and `get_b()` are functions that return an `int`:

```
int a = get_a();
int b = get_b();
int i = minimum<int>(a, b);
```

However, because this is a function template and the compiler can deduce the type of *T* from the arguments *a* and *b*, you can call it just like an ordinary function:

```
int i = minimum(a, b);
```

When the compiler encounters that last statement, it generates a new function in which every occurrence of *T* in the template is replaced with **int**:



```
int minimum(const int& lhs, const int& rhs)
{
    return lhs < rhs ? lhs : rhs;
}
```

The rules for how the compiler performs type deduction in function templates are based on the rules for ordinary functions. For more information, see [Overload Resolution of Function Template Calls](#).

## Type parameters

In the `minimum` template above, note that the type parameter `T` is not qualified in any way until it is used in the function call parameters, where the `const` and reference qualifiers are added.

There is no practical limit to the number of type parameters. Separate multiple parameters by commas:

```
template <typename T, typename U, typename V> class Foo{};
```

The keyword **`class`** is equivalent to **`typename`** in this context. You can express the previous example as:

```
template <class T, class U, class V> class Foo{};
```

You can use the ellipses operator (`...`) to define a template that takes an arbitrary number of zero or more type parameters:

```
template<typename... Arguments> class vtclass;

vtclass< > vtinstance1;
vtclass<int> vtinstance2;
vtclass<float, bool> vtinstance3;
```

Any built-in or user-defined type can be used as a type argument. For example, you can use `std::vector` in the Standard Library to store variables of type `int`, `double`, `std::string`, `MyClass`, `const MyClass*`, `MyClass&`, and so on. The primary restriction when using templates is that a type argument must support any operations that are applied to the type parameters. For example, if we call `minimum` using `MyClass` as in this example:

```
class MyClass
{
public:
    int num;
    std::wstring description;
};

int main()
```

```
{
    MyClass mc1 {1, L"hello"};
    MyClass mc2 {2, L"goodbye"};
    auto result = minimum(mc1, mc2); // Error! C2678
}
```

A compiler error will be generated because `MyClass` does not provide an overload for the `<` operator.

There is no inherent requirement that the type arguments for any particular template all belong to the same object hierarchy, although you can define a template that enforces such a restriction. You can combine object-oriented techniques with templates; for example, you can store a `Derived*` in a `vector<Base*>`. Note that the arguments must be pointers

```
vector<MyClass*> vec;
MyDerived d(3, L"back again", time(0));
vec.push_back(&d);

// or more realistically:
vector<shared_ptr<MyClass>> vec2;
vec2.push_back(make_shared<MyDerived>());
```

The basic requirements that `std::vector` and other standard library containers impose on elements of `T` is that `T` be copy-assignable and copy-constructible.

## Non-type parameters

Unlike generic types in other languages such as C# and Java, C++ templates support *non-type parameters*, also called value parameters. For example, you can provide a constant integral value to specify the length of an array, as with this example that is similar to the `std::array` class in the Standard Library:

```
template<typename T, size_t L>
class MyArray
{
    T arr[L];
public:
    MyArray() { ... }
};
```

Note the syntax in the template declaration. The `size_t` value is passed in as a template argument at compile time and must be **const** or a **constexpr** expression. You use it like this:

```
MyArray<MyClass*, 10> arr;
```

Other kinds of values including pointers and references can be passed in as non-type parameters. For example, you can pass in a pointer to a function or function object to customize some operation inside the

template code.

## Type deduction for non-type template parameters

In Visual Studio 2017 and later, in **/std:c++17** mode the compiler deduces the type of a non-type template argument that is declared with **auto**:

```
template <auto x> constexpr auto constant = x;

auto v1 = constant<5>;           // v1 == 5, decltype(v1) is int
auto v2 = constant<true>;        // v2 == true, decltype(v2) is bool
auto v3 = constant<'a'>;         // v3 == 'a', decltype(v3) is char
```

## Templates as template parameters

A template can be a template parameter. In this example, `MyClass2` has two template parameters: a typename parameter `T` and a template parameter `Arr`:

```
template<typename T, template<typename U, int I> class Arr>
class MyClass2
{
    T t; //OK
    Arr<T, 10> a;
    U u; //Error. U not in scope
};
```

Because the `Arr` parameter itself has no body, its parameter names are not needed. In fact, it is an error to refer to `Arr`'s typename or class parameter names from within the body of `MyClass2`. For this reason, `Arr`'s type parameter names can be omitted, as shown in this example:

```
template<typename T, template<typename, int> class Arr>
class MyClass2
{
    T t; //OK
    Arr<T, 10> a;
};
```

## Default template arguments

Class and function templates can have default arguments. When a template has a default argument you can leave it unspecified when you use it. For example, the `std::vector` template has a default argument for the allocator:

```
template <class T, class Allocator = allocator<T>> class vector;
```

In most cases the default `std::allocator` class is acceptable, so you use a vector like this:

```
vector<int> myInts;
```

But if necessary you can specify a custom allocator like this:

```
vector<int, MyAllocator> ints;
```

For multiple template arguments, all arguments after the first default argument must have default arguments.

When using a template whose parameters are all defaulted, use empty angle brackets:

```
template<typename A = int, typename B = double>
class Bar
{
    //...
};
...
int main()
{
    Bar<> bar; // use all default type arguments
}
```

## Template specialization

In some cases, it isn't possible or desirable for a template to define exactly the same code for any type. For example, you might wish to define a code path to be executed only if the type argument is a pointer, or a `std::wstring`, or a type derived from a particular base class. In such cases you can define a *specialization* of the template for that particular type. When a user instantiates the template with that type, the compiler uses the specialization to generate the class, and for all other types, the compiler chooses the more general template. Specializations in which all parameters are specialized are *complete specializations*. If only some of the parameters are specialized, it is called a *partial specialization*.

```
template <typename K, typename V>
class MyMap{/*...*/};

// partial specialization for string keys
template<typename V>
class MyMap<string, V> {/*...*/};
...
MyMap<int, MyClass> classes; // uses original template
MyMap<string, MyClass> classes2; // uses the partial specialization
```

A template can have any number of specializations as long as each specialized type parameter is unique. Only class templates may be partially specialized. All complete and partial specializations of a template must be declared in the same namespace as the original template.

For more information, see [Template Specialization](#).

# typename

---

In template definitions, provides a hint to the compiler that an unknown identifier is a type. In template parameter lists, is used to specify a type parameter.

## Syntax

```
typename identifier;
```

## Remarks

This keyword must be used if a name in a template definition is a qualified name that is dependent on a template argument; it is optional if the qualified name is not dependent. For more information, see [Templates and Name Resolution](#).

**typename** can be used by any type anywhere in a template declaration or definition. It is not allowed in the base class list, unless as a template argument to a template base class.

```
template <class T>
class C1 : typename T::InnerType // Error - typename not allowed.
{};
template <class T>
class C2 : A<typename T::InnerType> // typename OK.
{};
```

The **typename** keyword can also be used in place of **class** in template parameter lists. For example, the following statements are semantically equivalent:

```
template<class T1, class T2>...
template<typename T1, typename T2>...
```

## Example

```
// typename.cpp
template<class T> class X
{
    typename T::Y m_y;    // treat Y as a type
};

int main()
{
}
```

---

## See also

[Templates](#)

[Keywords](#)

# Class Templates

---

This topic describes rules that are specific to C++ class templates.

## Member functions of class templates

Member functions can be defined inside or outside of a class template. They are defined like function templates if defined outside the class template.

```
// member_function_templates1.cpp
template<class T, int i> class MyStack
{
    T* pStack;
    T StackBuffer[i];
    static const int cItems = i * sizeof(T);
public:
    MyStack( void );
    void push( const T item );
    T& pop( void );
};

template< class T, int i > MyStack< T, i >::MyStack( void )
{
};

template< class T, int i > void MyStack< T, i >::push( const T item )
{
};

template< class T, int i > T& MyStack< T, i >::pop( void )
{
};

int main()
{
}
```

Note that just as with any template class member function, the definition of the class's constructor member function includes the template argument list twice.

Member functions can themselves be function templates, specifying additional parameters, as in the following example.

```
// member_templates.cpp
template<typename T>
class X
{
public:
```



```

    template<typename U>
    void mf(const U &u);
};

template<typename T> template <typename U>
void X<T>::mf(const U &u)
{
}

int main()
{
}

```

## Nested class templates

Templates can be defined within classes or class templates, in which case they are referred to as member templates. Member templates that are classes are referred to as nested class templates. Member templates that are functions are discussed in [Member Function Templates](#).

Nested class templates are declared as class templates inside the scope of the outer class. They can be defined inside or outside of the enclosing class.

The following code demonstrates a nested class template inside an ordinary class.

```

// nested_class_template1.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

class X
{
    template <class T>
    struct Y
    {
        T m_t;
        Y(T t): m_t(t) { }
    };

    Y<int> yInt;
    Y<char> yChar;

public:
    X(int i, char c) : yInt(i), yChar(c) { }
    void print()
    {
        cout << yInt.m_t << " " << yChar.m_t << endl;
    }
};

int main()

```

```

{
    X x(1, 'a');
    x.print();
}

```

```

// nested_class_template2.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

template <class T>
class X
{
    template <class U> class Y
    {
        U* u;
    public:
        Y();
        U& Value();
        void print();
        ~Y();
    };

    Y<int> y;
public:
    X(T t) { y.Value() = t; }
    void print() { y.print(); }
};

template <class T>
template <class U>
X<T>::Y<U>::Y()
{
    cout << "X<T>::Y<U>::Y()" << endl;
    u = new U();
}

template <class T>
template <class U>
U& X<T>::Y<U>::Value()
{
    return *u;
}

template <class T>
template <class U>
void X<T>::Y<U>::print()
{
    cout << this->Value() << endl;
}

```

```

template <class T>
template <class U>
X<T>::Y<U>::~~Y()
{
    cout << "X<T>::Y<U>::~~Y()" << endl;
    delete u;
}

int main()
{
    X<int>* xi = new X<int>(10);
    X<char>* xc = new X<char>('c');
    xi->print();
    xc->print();
    delete xi;
    delete xc;
}

//Output:
X<T>::Y<U>::~Y()
X<T>::Y<U>::~Y()
10
99
X<T>::Y<U>::~~Y()
X<T>::Y<U>::~~Y()

```

Local classes are not allowed to have member templates.

## Template friends

Class templates can have [friends](#). A class or class template, function, or function template can be a friend to a template class. Friends can also be specializations of a class template or function template, but not partial specializations.

In the following example, a friend function is defined as a function template within the class template. This code produces a version of the friend function for every instantiation of the template. This construct is useful if your friend function depends on the same template parameters as the class does.

```

// template_friend1.cpp
// compile with: /EHsc

#include <iostream>
using namespace std;

template <class T> class Array {
    T* array;
    int size;

public:
    Array(int sz): size(sz) {
        array = new T[size];
    }
};

```

```

    memset(array, 0, size * sizeof(T));
}

Array(const Array& a) {
    size = a.size;
    array = new T[size];
    memcpy_s(array, a.array, sizeof(T));
}

T& operator[](int i) {
    return *(array + i);
}

int Length() { return size; }

void print() {
    for (int i = 0; i < size; i++)
        cout << *(array + i) << " ";

    cout << endl;
}

template<class T>
friend Array<T>* combine(Array<T>& a1, Array<T>& a2);
};

template<class T>
Array<T>* combine(Array<T>& a1, Array<T>& a2) {
    Array<T>* a = new Array<T>(a1.size + a2.size);
    for (int i = 0; i < a1.size; i++)
        (*a)[i] = *(a1.array + i);

    for (int i = 0; i < a2.size; i++)
        (*a)[i + a1.size] = *(a2.array + i);

    return a;
}

int main() {
    Array<char> alpha1(26);
    for (int i = 0 ; i < alpha1.Length() ; i++)
        alpha1[i] = 'A' + i;

    alpha1.print();

    Array<char> alpha2(26);
    for (int i = 0 ; i < alpha2.Length() ; i++)
        alpha2[i] = 'a' + i;

    alpha2.print();
    Array<char>*alpha3 = combine(alpha1, alpha2);
    alpha3->print();
    delete alpha3;
}

```

```
//Output:
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z a b c d e f g h i j k l m n o
p q r s t u v w x y z
```

The next example involves a friend that has a template specialization. A function template specialization is automatically a friend if the original function template is a friend.

It is also possible to declare only the specialized version of the template as the friend, as the comment before the friend declaration in the following code indicates. If you do this, you must put the definition of the friend template specialization outside of the template class.

```
// template_friend2.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

template <class T>
class Array;

template <class T>
void f(Array<T>& a);

template <class T> class Array
{
    T* array;
    int size;

public:
    Array(int sz): size(sz)
    {
        array = new T[size];
        memset(array, 0, size * sizeof(T));
    }
    Array(const Array& a)
    {
        size = a.size;
        array = new T[size];
        memcpy_s(array, a.array, sizeof(T));
    }
    T& operator[](int i)
    {
        return *(array + i);
    }
    int Length()
    {
        return size;
    }
    void print()
    {
```

```

        for (int i = 0; i < size; i++)
        {
            cout << *(array + i) << " ";
        }
        cout << endl;
    }
    // If you replace the friend declaration with the int-specific
    // version, only the int specialization will be a friend.
    // The code in the generic f will fail
    // with C2248: 'Array<T>::size' :
    // cannot access private member declared in class 'Array<T>'.
    //friend void f<int>(Array<int>& a);

    friend void f<>(Array<T>& a);
};

// f function template, friend of Array<T>
template <class T>
void f(Array<T>& a)
{
    cout << a.size << " generic" << endl;
}

// Specialization of f for int arrays
// will be a friend because the template f is a friend.
template<> void f(Array<int>& a)
{
    cout << a.size << " int" << endl;
}

int main()
{
    Array<char> ac(10);
    f(ac);

    Array<int> a(10);
    f(a);
}
//Output:
10 generic
10 int

```

The next example shows a friend class template declared within a class template. The class template is then used as the template argument for the friend class. Friend class templates must be defined outside of the class template in which they are declared. Any specializations or partial specializations of the friend template are also friends of the original class template.

```

// template_friend3.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

```

```

template <class T>
class X
{
private:
    T* data;
    void InitData(int seed) { data = new T(seed); }
public:
    void print() { cout << *data << endl; }
    template <class U> friend class Factory;
};

template <class U>
class Factory
{
public:
    U* GetNewObject(int seed)
    {
        U* pu = new U;
        pu->InitData(seed);
        return pu;
    }
};

int main()
{
    Factory< X<int> > XintFactory;
    X<int>* x1 = XintFactory.GetNewObject(65);
    X<int>* x2 = XintFactory.GetNewObject(97);

    Factory< X<char> > XcharFactory;
    X<char>* x3 = XcharFactory.GetNewObject(65);
    X<char>* x4 = XcharFactory.GetNewObject(97);
    x1->print();
    x2->print();
    x3->print();
    x4->print();
}
//Output:
65
97
A
a

```

## Reuse of Template Parameters

Template parameters can be reused in the template parameter list. For example, the following code is allowed:

```

// template_specifications2.cpp

class Y

```

```
{
};
template<class T, T* pT> class X1
{
};
template<class T1, class T2 = T1> class X2
{
};

Y aY;

X1<Y, &aY> x1;
X2<int> x2;

int main()
{
}
```

See also

[Templates](#)



# Function Templates

---

Class templates define a family of related classes that are based on the type arguments passed to the class upon instantiation. Function templates are similar to class templates but define a family of functions. With function templates, you can specify a set of functions that are based on the same code but act on different types or classes. The following function template swaps two items:

```
// function_templates1.cpp
template< class T > void MySwap( T& a, T& b ) {
    T c(a);
    a = b;
    b = c;
}
int main() {
}
```

This code defines a family of functions that swap the values of the arguments. From this template, you can generate functions that will swap **int** and **long** types and also user-defined types. **MySwap** will even swap classes if the class's copy constructor and assignment operator are properly defined.

In addition, the function template will prevent you from swapping objects of different types, because the compiler knows the types of the *a* and *b* parameters at compile time.

Although this function could be performed by a nontemplated function, using void pointers, the template version is typesafe. Consider the following calls:

```
int j = 10;
int k = 18;
CString Hello = "Hello, Windows!";
MySwap( j, k );           //OK
MySwap( j, Hello );       //error
```

The second **MySwap** call triggers a compile-time error, because the compiler cannot generate a **MySwap** function with parameters of different types. If void pointers were used, both function calls would compile correctly, but the function would not work properly at run time.

Explicit specification of the template arguments for a function template is allowed. For example:

```
// function_templates2.cpp
template<class T> void f(T) {}
int main(int j) {
    f<char>(j);    // Generate the specialization f(char).
    // If not explicitly specified, f(int) would be deduced.
}
```

When the template argument is explicitly specified, normal implicit conversions are done to convert the function argument to the type of the corresponding function template parameters. In the above example, the compiler will convert `j` to type `char`.

## See also

[Templates](#)

[Function Template Instantiation](#)

[Explicit Instantiation](#)

[Explicit Specialization of Function Templates](#)

# Function Template Instantiation

---

When a function template is first called for each type, the compiler creates an instantiation. Each instantiation is a version of the templated function specialized for the type. This instantiation will be called every time the function is used for the type. If you have several identical instantiations, even in different modules, only one copy of the instantiation will end up in the executable file.

Conversion of function arguments is allowed in function templates for any argument and parameter pair where the parameter does not depend on a template argument.

Function templates can be explicitly instantiated by declaring the template with a particular type as an argument. For example, the following code is allowed:

```
// function_template_instantiation.cpp
template<class T> void f(T) { }
```

```
// Instantiate f with the explicitly specified template.
// argument 'int'
//
template void f<int> (int);
```

```
// Instantiate f with the deduced template argument 'char'.
template void f(char);
int main()
{
}
```

See also

[Function Templates](#)

# Explicit Instantiation

---

You can use explicit instantiation to create an instantiation of a templated class or function without actually using it in your code. Because this is useful when you are creating library (.lib) files that use templates for distribution, uninstantiated template definitions are not put into object (.obj) files.

This code explicitly instantiates `MyStack` for `int` variables and six items:

```
template class MyStack<int, 6>;
```

This statement creates an instantiation of `MyStack` without reserving any storage for an object. Code is generated for all members.

The next line explicitly instantiates only the constructor member function:

```
template MyStack<int, 6>::MyStack( void );
```

You can explicitly instantiate function templates by using a specific type argument to re-declare them, as shown in the example in [Function Template Instantiation](#).

You can use the **extern** keyword to prevent the automatic instantiation of members. For example:

```
extern template class MyStack<int, 6>;
```

Similarly, you can mark specific members as being external and not instantiated:

```
extern template MyStack<int, 6>::MyStack( void );
```

You can use the **extern** keyword to keep the compiler from generating the same instantiation code in more than one object module. You must instantiate the template function by using the specified explicit template parameters in at least one linked module if the function is called, or you will get a linker error when the program is built.

[!NOTE] The **extern** keyword in the specialization only applies to member functions defined outside of the body of the class. Functions defined inside the class declaration are considered inline functions and are always instantiated.

## See also

[Function Templates](#)

# Explicit Specialization of Function Templates

---

With a function template, you can define special behavior for a specific type by providing an explicit specialization (override) of the function template for that type. For example:

```
template<> void MySwap(double a, double b);
```

This declaration enables you to define a different function for **double** variables. Like non-template functions, standard type conversions (such as promoting a variable of type **float** to **double**) are applied.

## Example

```
// explicit_specialization.cpp
template<class T> void f(T t)
{
};

// Explicit specialization of f with 'char' with the
// template argument explicitly specified:
//
template<> void f<char>(char c)
{
}

// Explicit specialization of f with 'double' with the
// template argument deduced:
//
template<> void f(double d)
{
}
int main()
{
}
```

## See also

[Function Templates](#)

# Partial Ordering of Function Templates (C++)

---

Multiple function templates that match the argument list of a function call can be available. C++ defines a partial ordering of function templates to specify which function should be called. The ordering is partial because there can be some templates that are considered equally specialized.

The compiler chooses the most specialized template function available from the possible matches. For example, if a function template takes a type `T` and another function template that takes `T*` is available, the `T*` version is said to be more specialized. It's preferred over the generic `T` version whenever the argument is a pointer type, even though both would be allowable matches.

Use the following process to determine if one function template candidate is more specialized:

1. Consider two function templates, T1 and T2.
2. Replace the parameters in T1 with a hypothetical unique type X.
3. With the parameter list in T1, see if T2 is a valid template for that parameter list. Ignore any implicit conversions.
4. Repeat the same process with T1 and T2 reversed.
5. If one template is a valid template argument list for the other template, but the converse isn't true, then that template is considered to be less specialized than the other template. If by using the previous step, both templates form valid arguments for each other, then they're considered to be equally specialized, and an ambiguous call results when you attempt to use them.
6. Using these rules:
  1. A template specialization for a specific type is more specialized than one taking a generic type argument.
  2. A template taking only `T*` is more specialized than one taking only `T`, because a hypothetical type `X*` is a valid argument for a `T` template argument, but `X` is not a valid argument for a `T*` template argument.
  3. `const T` is more specialized than `T`, because `const X` is a valid argument for a `T` template argument, but `X` is not a valid argument for a `const T` template argument.
  4. `const T*` is more specialized than `T*`, because `const X*` is a valid argument for a `T*` template argument, but `X*` is not a valid argument for a `const T*` template argument.

## Example

The following sample works as specified in the standard:

```
// partial_ordering_of_function_templates.cpp
// compile with: /EHsc
#include <iostream>
```

```

template <class T> void f(T) {
    printf_s("Less specialized function called\n");
}

template <class T> void f(T*) {
    printf_s("More specialized function called\n");
}

template <class T> void f(const T*) {
    printf_s("Even more specialized function for const T*\n");
}

int main() {
    int i = 0;
    const int j = 0;
    int *pi = &i;
    const int *cpi = &j;

    f(i);    // Calls less specialized function.
    f(pi);   // Calls more specialized function.
    f(cpi);  // Calls even more specialized function.
    // Without partial ordering, these calls would be ambiguous.
}

```

## Output

```

Less specialized function called
More specialized function called
Even more specialized function for const T*

```

## See also

[Function Templates](#)

# Member Function Templates

---

The term member template refers to both member function templates and nested class templates. Member function templates are template functions that are members of a class or class template.

Member functions can be function templates in several contexts. All functions of class templates are generic but are not referred to as member templates or member function templates. If these member functions take their own template arguments, they are considered to be member function templates.

## Example

Member function templates of nontemplate or template classes are declared as function templates with their template parameters.

```
// member_function_templates.cpp
struct X
{
    template <class T> void mf(T* t) {}
};

int main()
{
    int i;
    X* x = new X();
    x->mf(&i);
}
```

## Example

The following example shows a member function template of a template class.

```
// member_function_templates2.cpp
template<typename T>
class X
{
public:
    template<typename U>
    void mf(const U &u)
    {
    }
};

int main()
{
}
```



## Example

```
// defining_member_templates_outside_class.cpp
template<typename T>
class X
{
public:
    template<typename U>
    void mf(const U &u);
};

template<typename T> template <typename U>
void X<T>::mf(const U &u)
{
}

int main()
{
}
```

## Example

Local classes are not allowed to have member templates.

Member template functions cannot be virtual functions and cannot override virtual functions from a base class when they are declared with the same name as a base class virtual function.

The following example shows a templated user-defined conversion:

```
// templated_user_defined_conversions.cpp
template <class T>
struct S
{
    template <class U> operator S<U>()
    {
        return S<U>();
    }
};

int main()
{
    S<int> s1;
    S<long> s2 = s1; // Convert s1 using UDC and copy constructs S<long>.
}
```

## See also

[Function Templates](#)

# Template Specialization (C++)

---

Class templates can be partially specialized, and the resulting class is still a template. Partial specialization allows template code to be partially customized for specific types in situations, such as:

- A template has multiple types and only some of them need to be specialized. The result is a template parameterized on the remaining types.
- A template has only one type, but a specialization is needed for pointer, reference, pointer to member, or function pointer types. The specialization itself is still a template on the type pointed to or referenced.

## Example

```
// partial_specialization_of_class_templates.cpp
template <class T> struct PTS {
    enum {
        IsPointer = 0,
        IsPointerToDataMember = 0
    };
};

template <class T> struct PTS<T*> {
    enum {
        IsPointer = 1,
        IsPointerToDataMember = 0
    };
};

template <class T, class U> struct PTS<T U::*> {
    enum {
        IsPointer = 0,
        IsPointerToDataMember = 1
    };
};

struct S{};

extern "C" int printf_s(const char*,...);

int main() {
    printf_s("PTS<S>::IsPointer == %d PTS<S>::IsPointerToDataMember == %d\n",
        PTS<S>::IsPointer, PTS<S>::IsPointerToDataMember);
    printf_s("PTS<S*>::IsPointer == %d PTS<S*>::IsPointerToDataMember == %d\n",
        PTS<S*>::IsPointer, PTS<S*>::IsPointerToDataMember);
    printf_s("PTS<int S::*>::IsPointer == %d PTS"
        "<int S::*>::IsPointerToDataMember == %d\n",
        PTS<int S::*>::IsPointer, PTS<int S::*>::
        IsPointerToDataMember);
}
```

```
PTS<S>::IsPointer == 0 PTS<S>::IsPointerToDataMember == 0
PTS<S*>::IsPointer == 1 PTS<S*>::IsPointerToDataMember ==0
PTS<int S::*>::IsPointer == 0 PTS<int S::*>::IsPointerToDataMember == 1
```

## Example

If you have a template collection class that takes any type `T`, you can create a partial specialization that takes any pointer type `T*`. The following code demonstrates a collection class template `Bag` and a partial specialization for pointer types in which the collection dereferences the pointer types before copying them to the array. The collection then stores the values that are pointed to. With the original template, only the pointers themselves would have been stored in the collection, leaving the data vulnerable to deletion or modification. In this special pointer version of the collection, code to check for a null pointer in the `add` method is added.

```
// partial_specialization_of_class_templates2.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

// Original template collection class.
template <class T> class Bag {
    T* elem;
    int size;
    int max_size;

public:
    Bag() : elem(0), size(0), max_size(1) {}
    void add(T t) {
        T* tmp;
        if (size + 1 >= max_size) {
            max_size *= 2;
            tmp = new T [max_size];
            for (int i = 0; i < size; i++)
                tmp[i] = elem[i];
            tmp[size++] = t;
            delete[] elem;
            elem = tmp;
        }
        else
            elem[size++] = t;
    }

    void print() {
        for (int i = 0; i < size; i++)
            cout << elem[i] << " ";
        cout << endl;
    }
}
```

```

};

// Template partial specialization for pointer types.
// The collection has been modified to check for NULL
// and store types pointed to.
template <class T> class Bag<T*> {
    T* elem;
    int size;
    int max_size;

public:
    Bag() : elem(0), size(0), max_size(1) {}
    void add(T* t) {
        T* tmp;
        if (t == NULL) { // Check for NULL
            cout << "Null pointer!" << endl;
            return;
        }

        if (size + 1 >= max_size) {
            max_size *= 2;
            tmp = new T [max_size];
            for (int i = 0; i < size; i++)
                tmp[i] = elem[i];
            tmp[size++] = *t; // Dereference
            delete[] elem;
            elem = tmp;
        }
        else
            elem[size++] = *t; // Dereference
    }

    void print() {
        for (int i = 0; i < size; i++)
            cout << elem[i] << " ";
        cout << endl;
    }
};

int main() {
    Bag<int> xi;
    Bag<char> xc;
    Bag<int*> xp; // Uses partial specialization for pointer types.

    xi.add(10);
    xi.add(9);
    xi.add(8);
    xi.print();

    xc.add('a');
    xc.add('b');
    xc.add('c');
    xc.print();
}

```

```

int i = 3, j = 87, *p = new int[2];
*p = 8;
*(p + 1) = 100;
xp.add(&i);
xp.add(&j);
xp.add(p);
xp.add(p + 1);
p = NULL;
xp.add(p);
xp.print();
}

```

```

10 9 8
a b c
Null pointer!
3 87 8 100

```

## Example

The following example defines a template class that takes pairs of any two types and then defines a partial specialization of that template class specialized so that one of the types is **int**. The specialization defines an additional sort method that implements a simple bubble sort based on the integer.

```

// partial_specialization_of_class_templates3.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

template <class Key, class Value> class Dictionary {
    Key* keys;
    Value* values;
    int size;
    int max_size;
public:
    Dictionary(int initial_size) : size(0) {
        max_size = 1;
        while (initial_size >= max_size)
            max_size *= 2;
        keys = new Key[max_size];
        values = new Value[max_size];
    }
    void add(Key key, Value value) {
        Key* tmpKey;
        Value* tmpVal;
        if (size + 1 >= max_size) {
            max_size *= 2;
            tmpKey = new Key [max_size];
            tmpVal = new Value [max_size];
            for (int i = 0; i < size; i++) {

```

```

        tmpKey[i] = keys[i];
        tmpVal[i] = values[i];
    }
    tmpKey[size] = key;
    tmpVal[size] = value;
    delete[] keys;
    delete[] values;
    keys = tmpKey;
    values = tmpVal;
}
else {
    keys[size] = key;
    values[size] = value;
}
size++;
}

void print() {
    for (int i = 0; i < size; i++)
        cout << "{" << keys[i] << ", " << values[i] << "}" << endl;
}
};

```

// Template partial specialization: Key is specified to be int.

```

template <class Value> class Dictionary<int, Value> {
    int* keys;
    Value* values;
    int size;
    int max_size;
public:
    Dictionary(int initial_size) : size(0) {
        max_size = 1;
        while (initial_size >= max_size)
            max_size *= 2;
        keys = new int[max_size];
        values = new Value[max_size];
    }
    void add(int key, Value value) {
        int* tmpKey;
        Value* tmpVal;
        if (size + 1 >= max_size) {
            max_size *= 2;
            tmpKey = new int [max_size];
            tmpVal = new Value [max_size];
            for (int i = 0; i < size; i++) {
                tmpKey[i] = keys[i];
                tmpVal[i] = values[i];
            }
            tmpKey[size] = key;
            tmpVal[size] = value;
            delete[] keys;
            delete[] values;
            keys = tmpKey;
            values = tmpVal;
        }
    }
};

```

```

    }
    else {
        keys[size] = key;
        values[size] = value;
    }
    size++;
}

void sort() {
    // Sort method is defined.
    int smallest = 0;
    for (int i = 0; i < size - 1; i++) {
        for (int j = i; j < size; j++) {
            if (keys[j] < keys[smallest])
                smallest = j;
        }
        swap(keys[i], keys[smallest]);
        swap(values[i], values[smallest]);
    }
}

void print() {
    for (int i = 0; i < size; i++)
        cout << "{" << keys[i] << ", " << values[i] << "}" << endl;
}

};

int main() {
    Dictionary<char*, char*>* dict = new Dictionary<char*, char*>(10);
    dict->print();
    dict->add("apple", "fruit");
    dict->add("banana", "fruit");
    dict->add("dog", "animal");
    dict->print();

    Dictionary<int, char*>* dict_specialized = new Dictionary<int, char*>(10);
    dict_specialized->print();
    dict_specialized->add(100, "apple");
    dict_specialized->add(101, "banana");
    dict_specialized->add(103, "dog");
    dict_specialized->add(89, "cat");
    dict_specialized->print();
    dict_specialized->sort();
    cout << endl << "Sorted list:" << endl;
    dict_specialized->print();
}

```

```

{apple, fruit}
{banana, fruit}
{dog, animal}
{100, apple}

```

```
{101, banana}  
{103, dog}  
{89, cat}
```

Sorted list:

```
{89, cat}  
{100, apple}  
{101, banana}  
{103, dog}
```



# Templates and Name Resolution

---

In template definitions, there are three types of names.

- Locally declared names, including the name of the template itself and any names declared inside the template definition.
- Names from the enclosing scope outside the template definition.
- Names that depend in some way on the template arguments, referred to as dependent names.

While the first two names also pertain to class and function scopes, special rules for name resolution are required in template definitions to deal with the added complexity of dependent names. This is because the compiler knows little about these names until the template is instantiated, because they could be totally different types depending on which template arguments are used. Nondependent names are looked up according to the usual rules and at the point of definition of the template. These names, being independent of the template arguments, are looked up once for all template specializations. Dependent names are not looked up until the template is instantiated and are looked up separately for each specialization.

A type is dependent if it depends on the template arguments. Specifically, a type is dependent if it is:

- The template argument itself:

```
T
```

- A qualified name with a qualification including a dependent type:

```
T::myType
```

- A qualified name if the unqualified part identifies a dependent type:

```
N::T
```

- A const or volatile type for which the base type is a dependent type:

```
const T
```

- A pointer, reference, array, or function pointer type based on a dependent type:

```
T *, T &, T [10], T (*)()
```

- An array whose size is based on a template parameter:

```
template <int arg> class X {  
    int x[arg] ; // dependent type  
}
```

- a template type constructed from a template parameter:

```
T<int>, MyTemplate<T>
```

## Type Dependence and Value Dependence

Names and expressions dependent on a template parameter are categorized as type dependent or value dependent, depending on whether the template parameter is a type parameter or a value parameter. Also, any identifiers declared in a template with a type dependent on the template argument are considered value dependent, as is a integral or enumeration type initialized with a value-dependent expression.

Type-dependent and value-dependent expressions are expressions that involve variables that are type dependent or value dependent. These expressions can have semantics that differ, depending on the parameters used for the template.

## See also

[Templates](#)

# Name Resolution for Dependent Types

---

Use **typename** for qualified names in template definitions to tell the compiler that the given qualified name identifies a type. For more information, see [typename](#).

```
// template_name_resolution1.cpp
#include <stdio.h>
template <class T> class X
{
public:
    void f(typename T::myType* mt) {}
};

class Yarg
{
public:
    struct myType { };
};

int main()
{
    X<Yarg> x;
    x.f(new Yarg::myType());
    printf("Name resolved by using typename keyword.");
}
```

Name resolved by using typename keyword.

Name lookup for dependent names examines names from both the context of the template definition—in the following example, this context would find `myFunction(char)`—and the context of the template instantiation. In the following example, the template is instantiated in `main`; therefore, the `MyNamespace::myFunction` is visible from the point of instantiation and is picked as the better match. If `MyNamespace::myFunction` were renamed, `myFunction(char)` would be called instead.

All names are resolved as if they were dependent names. Nevertheless, we recommend that you use fully qualified names if there is any possible conflict.

```
// template_name_resolution2.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

void myFunction(char)
{
    cout << "Char myFunction" << endl;
}
```

```

}

template <class T> class Class1
{
public:
    Class1(T i)
    {
        // If replaced with myFunction(1), myFunction(char)
        // will be called
        myFunction(i);
    }
};

namespace MyNamespace
{
    void myFunction(int)
    {
        cout << "Int MyNamespace::myFunction" << endl;
    }
};

using namespace MyNamespace;

int main()
{
    Class1<int>* c1 = new Class1<int>(100);
}

```

## Output

```
Int MyNamespace::myFunction
```

## Template Disambiguation

Visual Studio 2012 enforces the C++98/03/11 standard rules for disambiguation with the "template" keyword. In the following example, Visual Studio 2010 would accept both the nonconforming lines and the conforming lines. Visual Studio 2012 accepts only the conforming lines.

```

#include <iostream>
#include <ostream>
#include <typeinfo>
using namespace std;

template <typename T> struct Allocator {
    template <typename U> struct Rebind {
        typedef Allocator<U> Other;
    };
};

```

```

template <typename X, typename AY> struct Container {
    #if defined(NONCONFORMANT)
        typedef typename AY::Rebind<X>::Other AX; // nonconformant
    #elif defined(CONFORMANT)
        typedef typename AY::template Rebind<X>::Other AX; // conformant
    #else
        #error Define NONCONFORMANT or CONFORMANT.
    #endif
};

int main() {
    cout << typeid(Container<int, Allocator<float>>::AX).name() << endl;
}

```

Conformance with the disambiguation rules is required because, by default, C++ assumes that `AY::Rebind` isn't a template, and so the compiler interprets the following "<" as a less-than. It has to know that `Rebind` is a template so that it can correctly parse "<" as an angle bracket.

## See also

[Name Resolution](#)

# Name Resolution for Locally Declared Names

---

The template's name itself can be referred to with or without the template arguments. In the scope of a class template, the name itself refers to the template. In the scope of a template specialization or partial specialization, the name alone refers to the specialization or partial specialization. Other specializations or partial specializations of the template can also be referenced, with the appropriate template arguments.

## Example

The following code shows that the class template's name A is interpreted differently in the scope of a specialization or partial specialization.

```
// template_name_resolution3.cpp
// compile with: /c
template <class T> class A {
    A* a1;    // A refers to A<T>
    A<int>* a2; // A<int> refers to a specialization of A.
    A<T*>* a3; // A<T*> refers to the partial specialization A<T*>.
};

template <class T> class A<T*> {
    A* a4; // A refers to A<T*>.
};

template<> class A<int> {
    A* a5; // A refers to A<int>.
};
```

## Example

In the case of a name conflict between a template parameter and another object, the template parameter can or cannot be hidden. The following rules will help determine precedence.

The template parameter is in scope from the point where it first appears until the end of the class or function template. If the name appears again in the template argument list or in the list of base classes, it refers to the same type. In standard C++, no other name that is identical to the template parameter can be declared in the same scope. A Microsoft extension allows the template parameter to be redefined in the scope of the template. The following example shows using the template parameter in the base specification of a class template.

```
// template_name_resolution4.cpp
// compile with: /EHsc
template <class T>
class Base1 {};

template <class T>
```

```
class Derived1 : Base1<T> {};
```

```
int main() {
    // Derived1<int> d;
}
```

## Example

When defining a template's member functions outside the class template, a different template parameter name can be used. If the template member function definition uses a different name for the template parameter than the declaration does, and the name used in the definition conflicts with another member of the declaration, the member in the template declaration takes precedence.

```
// template_name_resolution5.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

template <class T> class C {
public:
    struct Z {
        Z() { cout << "Z::Z()" << endl; }
    };
    void f();
};

template <class Z>
void C<Z>::f() {
    // Z refers to the struct Z, not to the template arg;
    // Therefore, the constructor for struct Z will be called.
    Z z;
}

int main() {
    C<int> c;
    c.f();
}
```

```
Z::Z()
```

## Example

When defining a template function or member function outside the namespace in which the template was declared, the template argument takes precedence over the names of other members of the namespace.

```

// template_name_resolution6.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

namespace NS {
    void g() { cout << "NS::g" << endl; }

    template <class T> struct C {
        void f();
        void g() { cout << "C<T>::g" << endl; }
    };
};

template <class T>
void NS::C<T>::f() {
    g(); // C<T>::g, not NS::g
};

int main() {
    NS::C<int> c;
    c.f();
}

```

C<T>::g

## Example

In definitions that are outside of the template class declaration, if a template class has a base class that does not depend on a template argument and if the base class or one of its members has the same name as a template argument, then the base class or member name hides the template argument.

```

// template_name_resolution7.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

struct B {
    int i;
    void print() { cout << "Base" << endl; }
};

template <class T, int i> struct C : public B {
    void f();
};

template <class B, int i>
void C<B, i>::f() {

```



```
    B b;    // Base class b, not template argument.
    b.print();
    i = 1; // Set base class's i to 1.
}

int main() {
    C<int, 1> c;
    c.f();
    cout << c.i << endl;
}
```

```
Base
1
```

## See also

[Name Resolution](#)

# Overload Resolution of Function Template Calls

---

A function template can overload nontemplate functions of the same name. In this scenario, function calls are resolved by first using template argument deduction to instantiate the function template with a unique specialization. If template argument deduction fails, the other function overloads are considered to resolve the call. These other overloads, also known as the candidate set, include nontemplate functions and other instantiated function templates. If template argument deduction succeeds, then the generated function is compared with the other functions to determine the best match, following the rules for overload resolution. For more information, see [Function Overloading](#).

## Example

If a nontemplate function is an equally good match to a template function, the nontemplate function is chosen (unless the template arguments were explicitly specified), as in the call `f(1, 1)` in the following example.

```
// template_name_resolution9.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

void f(int, int) { cout << "f(int, int)" << endl; }
void f(char, char) { cout << "f(char, char)" << endl; }

template <class T1, class T2>
void f(T1, T2)
{
    cout << "void f(T1, T2)" << endl;
};

int main()
{
    f(1, 1);    // Equally good match; choose the nontemplate function.
    f('a', 1); // Chooses the template function.
    f<int, int>(2, 2); // Template arguments explicitly specified.
}
```

```
f(int, int)
void f(T1, T2)
void f(T1, T2)
```

## Example

The next example illustrates that the exactly matching template function is preferred if the nontemplate function requires a conversion.

```

// template_name_resolution10.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

void f(int, int) { cout << "f(int, int)" << endl; }

template <class T1, class T2>
void f(T1, T2)
{
    cout << "void f(T1, T2)" << endl;
};

int main()
{
    long l = 0;
    int i = 0;
    // Call the template function f(long, int) because f(int, int)
    // would require a conversion from long to int.
    f(l, i);
}

```

```
void f(T1, T2)
```

## See also

[Name Resolution](#)

[typename](#)

# Source code organization (C++ Templates)

---

When defining a class template, you must organize the source code in such a way that the member definitions are visible to the compiler when it needs them. You have the choice of using the *inclusion model* or the *explicit instantiation* model. In the inclusion model, you include the member definitions in every file that uses a template. This approach is simplest and provides maximum flexibility in terms of what concrete types can be used with your template. Its disadvantage is that it can increase compilation times. The impact can be significant if a project and/or the included files themselves are large. With the explicit instantiation approach, the template itself instantiates concrete classes or class members for specific types. This approach can speed up compilation times, but it limits usage to only those classes that the template implementer has enabled ahead of time. In general, we recommend that you use the inclusion model unless the compilation times become a problem.

## Background

Templates are not like ordinary classes in the sense that the compiler does not generate object code for a template or any of its members. There is nothing to generate until the template is instantiated with concrete types. When the compiler encounters a template instantiation such as `MyClass<int> mc;` and no class with that signature exists yet, it generates a new class. It also attempts to generate code for any member functions that are used. If those definitions are in a file that is not `#included`, directly or indirectly, in the `.cpp` file that is being compiled, the compiler can't see them. From the compiler's point of view, this isn't necessarily an error because the functions may be defined in another translation unit, in which case the linker will find them. If the linker does not find that code, it raises an **unresolved external** error.

## The inclusion model

The simplest and most common way to make template definitions visible throughout a translation unit, is to put the definitions in the header file itself. Any `.cpp` file that uses the template simply has to `#include` the header. This is the approach used in the Standard Library.

```
#ifndef MYARRAY
#define MYARRAY
#include <iostream>

template<typename T, size_t N>
class MyArray
{
    T arr[N];
public:
    // Full definitions:
    MyArray(){}
    void Print()
    {
        for (const auto v : arr)
        {
            std::cout << v << " , ";
        }
    }
}
```

```

    }

    T& operator[](int i)
    {
        return arr[i];
    }
};
#endif

```

With this approach, the compiler has access to the complete template definition and can instantiate templates on-demand for any type. It is simple and relatively easy to maintain. However, the inclusion model does have a cost in terms of compilation times. This cost can be significant in large programs, especially if the template header itself `#includes` other headers. Every `.cpp` file that `#includes` the header will get its own copy of the function templates and all the definitions. The linker will generally be able to sort things out so that you do not end up with multiple definitions for a function, but it takes time to do this work. In smaller programs that extra compilation time is probably not significant.

## The explicit instantiation model

If the inclusion model is not viable for your project, and you know definitively the set of types that will be used to instantiate a template, then you can separate out the template code into an `.h` and `.cpp` file, and in the `.cpp` file explicitly instantiate the templates. This will cause object code to be generated that the compiler will see when it encounters user instantiations.

You create an explicit instantiation by using the keyword `template` followed by the signature of the entity you want to instantiate. This can be a type or a member. If you explicitly instantiate a type, all members are instantiated.

```
template MyArray<double, 5>;
```

```

//MyArray.h
#ifndef MYARRAY
#define MYARRAY

template<typename T, size_t N>
class MyArray
{
    T arr[N];
public:
    MyArray();
    void Print();
    T& operator[](int i);
};
#endif

```

```

//MyArray.cpp
#include <iostream>

```

```

#include "MyArray.h"

using namespace std;

template<typename T, size_t N>
MyArray<T,N>::MyArray(){}

template<typename T, size_t N>
void MyArray<T,N>::Print()
{
    for(const auto v : arr)
    {
        cout << v << " ";
    }
    cout << endl;
}

template MyArray<double, 5>;template MyArray<string, 5>;

```

In the previous example, the explicit instantiations are at the bottom of the .cpp file. A `MyArray` may be used only for **double** or **String** types.

[!NOTE] In C++11 the **export** keyword was deprecated in the context of template definitions. In practical terms this has little impact because most compilers never supported it.