告知我们有关下载 PDF 体验的信息。

C++ 文档

了解如何使用 C++ 和 C++ 标准库。

在 Visual Studio 中学习 C++

下载

下载 Visual Studio for Windows

在 Visual Studio 中安装 C/C++ 支持

仅下载命令行生成工具

开始使用

在 Visual Studio 中使用 C++ 生成 Hello World

用 C++ 创建控制台计算器

VIDEO

了解 C++ - 常规用途语言和库

培训

欢迎回到 C++ - 现代 C++

示例和示例存档

Visual Studio 中的 C++ 新变化

新变化

Visual Studio 中的 C++ 新变化

C++ 的符合性改进

概述

Visual Studio 中的 C++ 开发概述

支持的目标平台

帮助和社区资源

报告问题或提出建议

使用编译器和工具

参考

C/C++ 生成参考

项目和生成系统

编译器参考

链接器参考

更多生成工具

错误和警告

C++ 语言

参考

C++ 语言参考

C++ 关键字

C++ 运算符

C/C++ 预处理器参考

C++ 标准库 (STL)

参考

C++ 标准库概述

C++ 标准库参考（按标题）

C++ 标准库容器

迭代器

算法

分配器

函数对象

iostream 编程

正则表达式

文件系统导航

C++ 语言参考

本参考将介绍在 Microsoft C++ 编译器中实现的 C++ 编程语言。 本文的结构基于 Margaret Ellis 和 Bjarne Stroustrup 撰写的《C++ 参考手册批注》和 ANSI/ISO C++ 国际标准 (ISO/IEC FDIS 14882)。 本文涵盖了 C++ 语言功能的 Microsoft 专用实现。

有关新式 C++ 编程做法的概述，请参阅欢迎回到 C++。

请参阅下面的表以快速查找关键字或运算符：

- C++ 关键字
- C++ 运算符

本节内容

词法约定
C++ 程序的基本词法元素：标记、注释、运算符、关键字、标点符号、文本。 此外，还有文件转换、运算符优先级别/关联性。

基本概念
范围、链接、程序启动和终止、存储类以及类型。

内置类型
内置于C++编译器及其值范围的基本类型。

标准转换
内置类型之间的类型转换。 此外，算术转换和指针、引用与成员指针类型之间的转换。

声明和定义
声明和定义变量、类型和函数。

运算符、优先级别和关联性
C++ 中的运算符。

表达式
表达式的类型、表达式的语义、有关运算符的参考主题、强制转换和强制转换运算符、运行时类型信息。

Lambda 表达式
隐式定义函数对象类和构造该类类型的函数对象的编程技术。

语句
表达式、null、复合、选择、迭代、跳转和声明语句。

类和结构

介绍类、结构和联合。 此外，还介绍成员函数、特殊成员函数、数据成员、位域、`this` 指针和嵌套类。

联盟

用户定义的类型，其中所有成员都共享同一个内存位置。

派生类

单一继承和多重继承、`virtual` 函数、多个基类、抽象类、范围规则。 此外还有 `__super` 和 `__interface` 关键字。

成员访问控制

控制对类成员的访问：`public`、`private` 和 `protected` 关键字。 友元函数和友元类。

重载

重载运算符、运算符重载规则。

异常处理

C++ 异常处理、结构化异常处理 (SEH)、编写异常处理语句所使用的关键字。

断言和用户提供的消息

`#error` 指令、`static_assert` 关键字、`assert` 宏。

模板

模板规范、函数模板、类模板、`typename` 关键字、模板与宏、模板和智能指针。

事件处理

声明事件和事件处理程序。

Microsoft 专用的修饰符

Microsoft C++ 专用修饰符。 内存寻址、调用约定、`naked` 函数、扩展的存储类特性 (`__declspec`)、`__w64`。

内联汇编程序

在 `__asm` 块中使用汇编语言和 C++。

编译器 COM 支持

有关用于支持 COM 类型的 Microsoft 专用类和全局函数的参考。

Microsoft 扩展

Microsoft 的 C++ 扩展。

非标准行为

有关 Microsoft C++ 编译器的非标准行为的信息。

欢迎回到 C++
有关编写安全、正确且高效的程序的新式 C++ 编程做法的概述。

相关章节

适用于运行时平台的组件扩展
有关使用 Microsoft C++ 以 .NET 为目标的参考材料。

C/C++ 生成参考
编译器选项、链接器选项和其他生成工具。

C/C++ 预处理器参考
有关杂注、预处理器指令、预定义宏和预处理器的参考材料。

Visual C++ 库
指向各种 Microsoft C++ 库的参考起始页的链接的列表。

另请参阅

C 语言参考

Last updated on 2025/11/07

欢迎回到 C++ - 现代 C++

自创建以来，C++ 即已成为世界上最常用的编程语言之一。 正确编写的 C++ 程序快速、高效。 相对于其他语言，该语言更加灵活：它可以在最高的抽象级别上运行，还可以在硅级低级别上运行。 C++ 提供高度优化的标准库。 它支持访问低级别硬件功能，从而最大限度地提高速度并最大程度地降低内存需求。 C++ 几乎可以创建任何类型的程序：游戏、设备驱动程序、HPC、云、桌面、嵌入式和移动应用等。 甚至用于其他编程语言的库和编译器也使用 C++ 编写。

C++ 的原始要求之一是与 C 语言向后兼容。 因此，C++ 始终允许 C 样式编程，其中包含原始指针、数组、以 null 结尾的字符串和其他功能。 它们可以实现良好的性能，但也可能会引发 bug 并增加复杂性。 C++ 的演变注重可显著降低 C 样式惯例使用需求的功能。 如果需要，你仍可以使用旧的 C 编程设施。 但是，在新式 C++ 代码中，对上述设施的需求会越来越少。 现代 C++ 代码更加简单、安全、美观，而且速度仍像以往一样快速。

下面几个部分概述了现代 C++ 的主要功能。 此处列出的功能在 C++11 及更高版本中可用，除非另有说明。 在 Microsoft C++ 编译器中，可以设置 /std 编译器选项，指定要用于项目的标准版本。

资源和智能指针

C 样式编程的一个主要 bug 类型是内存泄漏。 泄漏通常是由未能为使用 `delete` 分配的内存调用 `new` 导致的。 现代 C++ 强调“资源获取即初始化”(RAII) 原则。 其理念很简单。 资源（堆内存、文件句柄、套接字等）应由对象“拥有”。 该对象在其构造函数中创建或接收新分配的资源，并在其析构函数中将此资源删除。 RAII 原则可确保当所属对象超出范围时，所有资源都能正确返回到操作系统。

为了支持对 RAII 原则的简单采用，C++ 标准库提供了三种智能指针类型：std::unique_ptr、std::shared_ptr 和 std::weak_ptr。 智能指针可处理对其拥有的内存的分配和删除。 下面的示例演示了一个类，其中包含一个数组成员，该成员是在调用 `make_unique()` 时在堆上分配的。 对 `new` 和 `delete` 的调用将由 `unique_ptr` 类封装。 当 `widget` 对象超出范围时，将调用 unique_ptr 析构函数，此函数将释放为数组分配的内存。

C++

```
#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
```

请尽可能地使用智能指针管理堆内存。 如果必须显式使用 `new` 和 `delete` 运算符，请遵循 RAII 原则。 有关详细信息，请参阅对象生存期和资源管理 (RAII)。

`std::string` 和 `std::string_view`

C 样式字符串是 bug 的另一个主要来源。 通过使用 std::string 和 std::wstring，几乎可以消除与 C 样式字符串关联的所有错误。 还可以利用成员函数的优势进行搜索、追加和在前面追加等操作。 两者都对速度进行了高度优化。 将字符串传递到仅需要只读访问权限的函数时，在 C++17 中，可以使用 std::string_view，以便提高性能。

`std::vector` 和其他标准库容器

标准库容器都遵循 RAII 原则。 它们为安全遍历元素提供迭代器。 此外，它们对性能进行了高度优化，并且已充分测试正确性。 通过使用这些容器，可以消除自定义数据结构中可能引入的 bug 或低效问题。 使用 vector 替代原始数组，来作为 C++ 中的序列容器。

C++

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

使用 map（而不是 `unordered_map`），作为默认关联容器。 对于退化和多案例，使用 set、multimap 和 multiset。

C++

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

需要进行性能优化时，请考虑以下用法：

- 使用无序的关联容器，例如 unordered_map。 它们的每个元素的开销较低，并且具有固定时间查找功能，但正确高效地使用它们的难度更高。
- 使用排序的 `vector`。 有关详细信息，请参阅算法。

请勿使用 C 样式数组。 对于需要直接访问数据的旧 API，请改用 `f(vec.data(), vec.size());` 等访问器方法。 有关容器的详细信息，请参阅 C++ 标准库容器。

标准库算法

在假设需要为程序编写自定义算法之前，请先查看 C++ 标准库算法。 标准库包含许多常见操作（如搜索、排序、筛选和随机化）的算法分类，这些分类在不断增长。 数学库的内容很广泛。 在 C++17 及更高版本中，提供了许多算法的并行版本。

以下是一些重要示例：

- `for_each`，默认遍历算法（以及基于范围的 `for` 循环）。
- `transform`，用于对容器元素进行非就地修改
- `find_if`，默认搜索算法。
- `sort`、`lower_bound` 和其他默认的排序和搜索算法。

若要编写比较运算符，请使用严格的 `<`，并尽可能使用命名 lambda。

C++

```
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(), widget{0}, comp );
```

用 `auto` 替代显式类型名称

C++11 引入了 auto 关键字，以便可将其用于变量、函数和模板声明中。 `auto` 会指示编译器推导对象的类型，这样你就无需显式键入类型。 当推导出的类型是嵌套模板时，`auto` 尤其有用：

C++

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

基于范围的 `for` 循环

对数组和容器的 C 样式迭代容易引发索引错误，而且键入过程单调乏味。 若要消除这些错误，并提高代码的可读性，可使用基于范围的 `for` 循环，此循环包含标准库容器和原始数组。 有关详细信息，请参阅基于范围的 for 语句。

C++

```
#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;
    }
}
```

用 `constexpr` 表达式替代宏

C 和 C++ 中的宏是指编译之前由预处理器处理的标记。 在编译文件之前，宏标记的每个实例都将替换为其定义的值或表达式。 C 样式编程通常使用宏来定义编译时常量值。 但宏容易出错且难以调试。 在现代 C++ 中，应优先使用 constexpr 变量定义编译时常量：

C++

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

统一初始化

在现代 C++ 中，可以使用任何类型的括号初始化。 在初始化数组、矢量或其他容器时，这种初始化形式会非常方便。 在下面的示例中，使用三个 `v2` 实例初始化 `S`。 `v3` 将使用三个 `S` 实例进行初始化，这些实例使用括号初始化自身。 编译器基于 `v3` 声明的类型推断每个元素的类型。

C++

```
#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} };

}
```

若要了解详细信息，请参阅括号初始化。

移动语义

现代 C++ 提供了移动语义，此功能可以避免进行不必要的内存复制。 在此语言的早期版本中，在某些情况下无法避免复制。 移动操作会将资源的所有权从一个对象转移到下一个对象，而不必再进行复制。 一些类拥有堆内存、文件句柄等资源。 实现资源所属的类时，可以定义此类的移动构造函数和移动赋值运算符。 在解析重载期间，如果不需要进行复制，编译器会选择这些特殊成员。 如果定义了移动构造函数，则标准库容器类型会在对象中调用此函数。 有关详细信息，请参阅移动构造函数和移动赋值运算符 (C++)。

Lambda 表达式

在 C 样式编程中，可以通过使用函数指针将函数传递到另一个函数。 函数指针不便于维护和理解。 它们引用的函数可能是在源代码的其他位置中定义的，而不是从调用它的位置定义的。 此外，它们不是类型安全的。 现代 C++ 提供了函数对象和重写 operator() 运算符的类，可以像调用函数一样调用它们。 创建函数对象的最简便方法是使用内联 lambda 表达式。 下面的示例演示如何使用 lambda 表达式传递函数对象，然后由 `find_if` 函数在矢量的每个元素中调用此函数对象：

```
C++
```

```
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; });
```

可以将 lambda 表达式 `[=](int i) { return i > x && i < y; }` 理解为“采用类型 `int` 的单个参数并返回一个布尔值来表示此参数是否大于 `x` 并且小于 `y` 的函数”。请注意，可在 lambda 中使用来自周围上下文的 `x` 和 `y` 变量。 `[=]` 会指定通过值捕获这些变量；换言之，对于这些值，lambda 表达式具有自己的值副本。

异常

与错误代码相比，新式 C++ 更注重异常，将其作为报告和处理错误条件的最佳方法。 有关详细信息，请参阅现代 C++ 处理异常和错误的最佳做法。

`std::atomic`

对线程间通信机制使用 C++ 标准库 std::atomic 结构和相关类型。

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

C 样式编程通常通过并集使不同类型的成员可以占用同一个内存位置，从而节省内存。 但是，并集不是类型安全的，并且容易导致编程错误。 C++17 引入了更加安全可靠的 std::variant 类，来作为并集的替代项。 可以使用 std::visit 函数以类型安全的方式访问 `variant` 类型的成员。

另请参阅

C++ 语言参考
Lambda 表达式
C++ 标准库
Microsoft C/C++ 语言一致性

Last updated on 2025/11/07

词法约定

本节介绍 C++ 程序的基本元素。 你将使用这些名为“词法元素”或“标记”的元素构造用于构造完整程序的语句、定义和声明等。 本节将讨论以下词法元素：

- 令牌和字符集
- 注释
- 标识符
- 关键字
- 标点符号
- 数值、布尔值和指针文本
- 字符串和字符文本
- 用户定义的文本

有关如何分析 C++ 源文件的详细信息，请参阅转换阶段。

另请参阅

C++ 语言参考
翻译单元和链接

Last updated on 2025/11/07

标记和字符集

C++ 程序的文本由标记和空格组成。 标记是对编译器有用的 C++ 程序的最小元素。 C++ 分析器识别这些类型的标记：

- 关键字
- 标识符
- 数值、布尔和指针文本
- 字符串和字符文本
- 用户定义的文本
- 运算符
- 标点符号

标记通常由一个或多个空格分隔：

- 空白
- 水平或垂直制表符
- 新行
- 表单源
- 注释

基本源字符集

C++ 标准指定可用于源文件的基本源字符集。 若要表示这组字符之外的字符，可以通过使用 *通用字符名称*指定其他字符。 MSVC 实现允许使用附加字符。 基本源字符集由可用于源文件的 96 个字符组成。 这组字符包括空白字符、水平选项卡、垂直选项卡、换页符和换行控制字符以及这一组图形字符：

`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 专用

MSVC 包括作为基本源字符集成员的 `$` 字符。 MSVC 还允许基于文件编码在源文件中使用额外一组字符。 默认情况下，Visual Studio 通过使用默认代码页存储源文件。 当通过使用特定于区域设置的代码页或 Unicode 代码页保存源文件时，MSVC 允许你在你的源代码中使用该代码页的任何字符，基本源字符集中未明确允许的控制代码除外。 例如，如果你使用日语代码页保存文件，则可以在注释、标识符或字符串中放置日语字符。 MSVC 不允许使用不能转换为有效多

字节字符或 Unicode 码位的字符序列。 并非所有允许的字符均可显示在标识符中，具体取决于编译器选项。 有关详细信息，请参阅标识符。

结束 Microsoft 专用

通用字符名称

由于 C++ 程序可使用的字符要比在基本源字符集中指定的字符要多得多，所以可以通过使用 *通用字符名称*以可移植的方式指定这些字符。 通用字符名称由表示 Unicode 码位的字符序列组成。 采用两种形式。 使用 `\UNNNNNNNN` 表示形式为 U+NNNNNNNN 的 Unicode 码位，其中 NNNNNNNN 是八位的十六进制码位数字。 使用四位的 `\uNNNN` 表示形式为 U+0000NNNN 的 Unicode 码位。

通用字符名称可用于标识符、字符串和字符文本中。 通用字符名称不能用于表示范围 0xD800-0xDFFF 之内的代理项码位。 而应使用所需的码位；编译器会自动生成任何必需的代理项。 其他限制适用于可在标识符中使用的通用字符名称。 有关详细信息，请参阅 Identifiers 和 String and Character Literals。

Microsoft 专用

Microsoft C++ 编译器将通用字符名称形式的字符和文本形式的字符视为可互换。 例如，你可以声明一个使用通用字符名称形式的标识符，并以文本形式使用它：

C++

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

Windows 剪贴板上的扩展字符的格式特定于应用程序区域设置。 从另一个应用程序剪切这些字符并将其粘贴到你的代码中可能会引入意外的字符编码。 这可能会导致在你的代码中出现原因不可见的分析错误。 我们建议你在粘贴扩展的字符之前将你的源文件编码设置为 Unicode 代码页。 我们还建议你使用 IME 或字符映射应用生成扩展的字符。

结束 Microsoft 专用

执行字符集

执行字符集表示编译程序中可显示的字符和字符串。 这些字符集包含源文件中允许的所有字符，以及表示警告、退格、回车和空字符的控制字符。 执行字符集具有特定于区域设置的表示形式。

Last updated on 2025/11/07

注释 (C++)

注释是编译器忽略的文本，但它对程序员很有用。 注释通常用于批注代码以供将来参考。 编译器将它们视为空白。 可以在测试中使用注释来使某些代码行处于非活动状态；但是，`#if`/`#endif` 预处理器指令在这方面表现更佳，因为你可以环绕包含注释的代码，但不能嵌套注释。

C++ 注释的编写方法如下：

- `/*`（斜线、星号）字符，后跟任意字符序列（包括新行），然后是 `*/` 字符。 此语法与 ANSI C 相同。
- `//`（两个斜杠）字符，后跟任意字符序列。 没有紧跟反斜杠的新行将终止这种形式的注释。 因此，它通常称为“单行注释”。

注释字符（`/*`、`*/` 和 `//`）在字符常量、字符串字面量或注释中没有特殊含义。 因此，不能嵌套使用第一种语法的注释。

另请参阅

词法约定

Last updated on 2025/11/07

标识符 (C++)

标识符是用于表示以下内容之一的字符序列：

- 对象或变量名称
- 类、结构或联合名称
- 枚举类型名称
- 类、结构、联合或枚举的成员
- 函数或类成员函数
- typedef 名称
- 标签名称
- 宏名称
- 宏参数

允许将以下字符用作标识符的任意字符：

```
_ 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
```

还允许在标识符中使用通用字符名称的某些范围。 标识符中的通用字符名称不能指定控制字符或基本源字符集中的字符。 有关详细信息，请参阅 Character Sets。 允许将以下 Unicode 码位数字范围用作标识符中任意字符的通用字符名称：

- 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-FFFD、10000-1FFFD、20000-2FFFD、30000-3FFFD、40000-4FFFD、50000-5FFFD、60000-6FFFD、70000-7FFFD、80000-8FFFD、90000-9FFFD、A0000-AFFFD、B0000-BFFFD、C0000-CFFFD、D0000-DFFFD、E0000-EFFFD

允许将以下字符用作标识符中除第一个字符以外的任意字符：

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

还允许将以下 Unicode 码位数字范围用作标识符中除第一个字符以外任意字符的通用字符名称：

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

Microsoft 专用

只有 Microsoft C++ 标识符的前 2048 个字符是有意义的。 用户定义类型的名称由编译器“修饰”以保留类型信息。 结果名称（包括类型信息）不能超过 2048 个字符。 （有关详细信息，请参阅修饰名。）可能影响修饰标识符长度的因素包括：

- 标识符是表示用户定义类型的对象还是表示派生自用户定义类型的类型。
- 标识符是否表示派生自函数的函数或类型。
- 函数的参数的数量。

美元符号 `$` 在 Microsoft C++ 编译器 (MSVC) 中是有效标识符。 MSVC 还允许在标识符中使用通用字符名称允许的范围所表示的实际字符。 若要使用这些字符，必须使用包含它们的文件编码代码页保存文件。 此示例演示如何在代码中互换使用扩展字符和通用字符名称。

C++

```cpp
// 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
}
```

编译 C++/CLI 代码时，标识符中允许的字符范围限制更少。 使用 /clr 编译的代码中的标识符应遵循标准 ECMA-335：公共语言基础结构 (CLI)。

结束 Microsoft 专用

标识符的第一个字符必须是字母字符（大写、小写或带下划线 (_) 的字母）。 由于 C++ 标识符区分大小写，因此 `fileName` 与 `FileName` 不同。

标识符不能与关键字有完全相同的拼写和大小写。 包含关键字的标识符是合法的。 例如，`Pint` 是一个合法标识符，即使它包含 `int` 关键字。

在标识符中使用两个顺序下划线字符（__）或在单个前导下划线后跟一个大写字母的用法是专为所有范围的 C++ 实现保留的。 由于当前或将来的保留标识符可能发生冲突，因此应避免对文件范围的名称使用一个前导下划线后跟小写字母。

另请参阅

词法约定

Last updated on 2025/11/07

关键字 (C++)

关键字是具有特殊意义的预定义保留标识符。 它们不能用作程序中的标识符。 Microsoft C++ 保留了下列关键字。 带有前导下划线的名称，以及为 C++/CX 和 C++/CLI 指定的名称都是 Microsoft 扩展。

标准 C++ 关键字

alignas
alignof
and [b]
and_eq [b]
asm [一个]
auto
bitand [b]
bitor [b]
bool
break
case
catch
char
char8_t [c]
char16_t
char32_t
class
compl [b]
`concept` [c]
const
const_cast
`consteval` [c]
constexpr
`constinit` [c]
continue
`co_await` [c]
`co_return` [c]
`co_yield` [c]
decltype
default
delete

do
double
dynamic_cast
else
enum
explicit
`export` [c]
extern
false
float
for
friend
goto
if
inline

int
long
mutable
namespace
new
noexcept
not [b]
not_eq [b]
nullptr
operator
or [b]
or_eq [b]
private
protected
public
register
reinterpret_cast
`requires` [c]
return
short
signed
sizeof
static
static_assert

static_cast
struct
switch
template
this
thread_local
throw
true
try
typedef
typeid
typename
union
unsigned
using 声明
using 指令
virtual
void
volatile
wchar_t
while
xor [b]
xor_eq [b]

[a] Microsoft 专用 `__asm` 关键字替换了 C++ `asm` 语法。 保留了 `asm` 以便与其他 C++ 实现兼容，但未成功。 将 `__asm` 用于 x86 目标上的内联程序集。 Microsoft C++ 不支持其他目标的内联程序集。

[b] 当指定 /permissive- 或 /Za（禁用语言扩展）时，扩展运算符同义词是关键字。 当启用 Microsoft 扩展时，它们不是关键字。

[c] 指定 /std:c++20 或更高版本（例如 `/std:c++latest`）时受支持。

Microsoft 专用 C++ 关键字

在 C++ 中，包含两个连续下划线的标识符会保留用于编译器实现。 Microsoft 约定位于带双下划线的 Microsoft 专用关键字前面。 这些单词不能用作标识符名称。

默认情况下将启用 Microsoft 扩展。 若要确保你的程序是完全可移植的，可通过在编译期间指定 /permissive- 或 /Za（禁用语言扩展）选项来禁用 Microsoft 扩展。 这些选项禁用某些 Microsoft 专用关键字。

启用 Microsoft 扩展后，你可以在程序中使用 Microsoft 特定关键字。 为了符合 ANSI，这些关键字的前面有一条双下划线。 出于后向兼容性考虑，支持许多双下划线关键字的单下划线版本。 提供的 `__cdecl` 关键字没有前导下划线。

`__asm` 关键字替代了 C++ `asm` 语法。 保留了 `asm` 以便与其他 C++ 实现兼容，但未成功。 请使用 `__asm`。

`__based` 关键字对 32 位和 64 位目标编译的用途有限。

__alignof [e]
__asm [e]
__assume [e]
__based [e]
__cdecl [e]
__declspec [e]
__event
__except [e]
__fastcall [e]
__finally [e]
__forceinline [e]

__hook [d]
__if_exists
__if_not_exists
__inline [e]
__int16 [e]
__int32 [e]
__int64 [e]
__int8 [e]
__interface
__leave [e]
__m128

__m128d
__m128i
__m64
__multiple_inheritance [e]
__ptr32 [e]
__ptr64 [e]
__raise
__restrict [e]
__single_inheritance [e]

__sptr [e]
__stdcall [e]

__super
__thiscall
__unaligned [e]
__unhook [d]
__uptr [e]
__uuidof [e]
__vectorcall [e]
__virtual_inheritance [e]
__w64 [e]
__wchar_t

[d] 事件处理中使用的内部函数。

[e] 为了与以前的版本向后兼容，当启用 Microsoft 扩展时（默认），这些关键字既可以使用两个前导下划线，也可以使用一个前导下划线。

__declspec 修饰符中的 Microsoft 关键字

这些标识符是 `__declspec` 修饰符的扩展属性。 它们被视为该上下文中的关键字。

align
allocate
allocator
appdomain
code_seg
deprecated

dllexport
dllimport
jitintrinsic
naked
noalias
noinline

noreturn
no_sanitize_address
nothrow
novtable

process
property

restrict
safebuffers
selectany
spectre
thread
uuid

C++/CLI 和 C++/CX 关键字

__abstract [f]
__box [f]
__delegate [f]
__gc [f]
__identifier
__nogc [f]
__noop
`__pin` [f]
`__property` [f]
`__sealed` [f]

__try_cast [f]
__value [f]
abstract [g]
array [g]
as_friend
delegate [g]
enum class
enum struct
event [g]

finally
for each in
gcnew [g]
generic [g]
initonly
interface class [g]
interface struct [g]

interior_ptr [g]
literal [g]

new [g]
property [g]
ref class
ref struct
safecast
sealed [g]
typeid
value class [g]
value struct [g]

[f] 仅适用于 C++ 托管扩展。 此语法现已弃用。 有关更多信息，请参见 Component Extensions for Runtime Platforms。

[g] 适用于 C++/CLI。

另请参阅

词法约定
C++ 内置运算符、优先级和关联性

Last updated on 2025/11/07

标点符号 (C++)

在 C++ 中，标点符号相对于编译器来说具有语法意义和语义含义，但是它们本身不会指定一个产生数值的操作。 某些标点符号（单独或组合）也可以是 C++ 运算符或对预处理器很重要。

以下任意字符都被视为标点符号：

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

标点符号 [ ]、( ) 和 { } 必须成对出现在转换阶段 4 后。

另请参阅

词法约定

Last updated on 2025/11/07

数值、布尔和指针文本

文本是一种直接表示值的程序元素。 本文介绍整数、浮点、布尔和指针类型的文本。 有关字符串文本和字符文本的信息，请参阅字符串文本和字符文本 (C++)。 你也可以基于任何这些类别定义自己的文本。 有关详细信息，请参阅用户定义的文本 (C++)。

你可以在许多上下文中使用文本，但文本的最常用法是初始化命名变量以及将自变量传递给函数：

C++

```
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
```

有时需要指示编译器如何解释某个文本或者为其赋予哪种特定类型。 可以通过为文本追加前缀或后缀来实现此目的。 例如，前缀 `0x` 指示编译器将其后面的数字解释为十六进制值，例如 `0x35`。 后缀 `ULL` 指示编译器将值视为 **`unsigned long long`** 类型，就像 `5894345ULL` 中那样。 有关每个文本类型的前缀和后缀的完整列表，请参阅以下各节。

整数文本

整数文本以数字开头，没有小数部分或指数。 你可以指定十进制、二进制、八进制或十六进制形式的整数文本。 可以选择使用后缀将整数文本指定为无符号类型以及 long 类型或 long long 类型。

如果没有前缀或后缀，编译器将为整型文本值赋予 **`int`** 类型（32 位），前提是该值符合该类型，否则将赋予 **`long long`** 类型（64 位）。

要指定十进制整型文本，请以非零数字作为规范的开头。 例如：

C++

```
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
```

要指定八进制整型文本，请以 0 作为规范的开头，后跟 0 到 7 之间的一系列数字。 在指定八进制文本时，使用数字 8 和 9 是错误做法。 例如：

C++

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

要指定十六进制整型文本，请以 `0x` 或 `0X` 作为规范的开头（“x”的大小写形式并不重要），后跟 `0` 到 `9` 以及 `a`（或 `A`）到 `f`（或 `F`）之间的一系列数字。 十六进制数字 `a`（或 `A`）到 `f`（或 `F`）表示介于 10 和 15 之间的值。 例如：

C++

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

若要指定无符号类型，请使用 `u` 或 `U` 后缀。 若要指定 long 类型，请使用 `l` 或 `L` 后缀。 要指定 64 位整型类型，请使用 LL 或 ll 后缀。 i64 后缀仍受支持，但不建议使用。 它是 Microsoft 专用的，不可移植。 例如：

C++

```
unsigned val_1 = 328u;                  // Unsigned value
long val_2 = 0x7FFFFFL;                 // 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
```

数字分隔符：可以使用单引号字符（撇号）分隔较大数字中的位值，使它们更易于人类阅读。 分隔符不会对编译产生任何影响。

C++

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

浮点文本

浮点文本指定必须具有小数部分的值。 这些值包含小数点 (`.`) 并可能包含指数。

浮点文本有一个有效数字（有时称为尾数），它指定数字的值。 它们有一个指数，用于指定数字的度量值。 而且，它们有一个可选的后缀，用于指定文本的类型。 指定的有效数字的格式是一系列位数后跟一个句点，再后跟表示数字的小数部分的可选的一系列位数。 例如：

C++

```
18.46
38.
```

指数（如果有）指定数字的量级为 10 次幂，如以下示例所示：

```
C++
18.46e0      // 18.46
18.46e1      // 184.6
```

指数可以使用 `e` 或 `E` （意义相同）后跟可选的符号（+ 或 -）和一系列数字来指定。 如果指数存在，则整数（如 `18E0`）中不需要尾随的小数点。

浮点文本默认为 **`double`** 类型。 通过使用后缀 `f` 或 `l` 或 `F` 或 `L` （后缀不区分大小写），可以将文本指定为 **`float`** 或 **`long double`**。

虽然 **`long double`** 和 **`double`** 具有相同的表示形式，但它们不属于同一类型。 例如，你可能有类似于下面的重载函数

```
C++
void func( double );
```

和

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

布尔值文字

布尔文本为 **`true`** 和 **`false`**。

指针文本 (C++11)

C++ 引入了 nullptr 文本来指定初始化为零的指针。 在可移植代码中，应使用 **`nullptr`**，而不是整型类型零或宏（如 `NULL`）。

二进制文本 (C++14)

可以通过使用 `0B` 或 `0b` 前缀，后跟一系列 1 和 0，来指定二进制文本：

```
C++
```

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

避免将文本用作“魔术常量”

你可以在表达式和语句中直接使用文本，虽然这种编程做法并不一定好用：

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

在上一个示例中，更好的做法是使用能够传达明确含义的命名常量，例如“MAXIMUM_ERROR_THRESHOLD”。 如果最终用户看到返回值“成功”，那么使用命名字符串常量可能会更好。 可以将字符串常量保存在可本地化为其他语言的文件中的单个位置。 使用命名常量可帮助你自己和其他人了解代码的含义。

另请参阅

词法约定
C++ 字符串文本
C++ 用户定义的文本

Last updated on 2025/11/07

字符串和字符文本 (C++)

C++ 支持各种字符串和字符类型，并提供表示每种类型的文本值的方法。 在源代码中，使用字符集表示字符和字符串文本的内容。 通用字符名称和转义字符允许你仅使用基本源字符集表示任何字符串。 原始字符串使你可以避免使用转义字符，可以用于表示所有类型的字符串。 还可以创建 `std::string` 文本，而无需执行额外的构造或转换步骤。

C++

```
#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* before C++20, encoded as UTF-8,
                         // const char8_t* in C++20
    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* before C++20, encoded as UTF-8,
                                      // const char8_t* in C++20
    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 before C++20, std::u8string in C++20
    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* before
C++20, encoded as UTF-8,
                                       // std::u8string in C++20
```

```
    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
}
```

字符串文本可以米有前缀，也可以具有 `u8`、`L`、 `u`和 `U` 前缀以分别指示窄字符（单字节或多字节）、UTF-8、宽字符（UCS-2 或 UTF-16）、UTF-16 和 UTF-32 编码。 原始字符串文本可以具有 `R`、`u8R`、`LR`、`uR` 和 `UR` 前缀来表示这些编码的原始版本等效项。 若要创建临时或静态 `std::string` 值，可以使用带 `s` 后缀的字符串文本或原始字符串文本。 有关详细信息，请参阅下面的字符串文本部分。 有关基本源字符集、通用字符名称以及在源代码中使用扩展代码页中的字符的详细信息，请参阅字符集。

字符文本

字符文本 由一个字符常量构成。 它是用单引号括起来的字符来表示的。 有五种类型的字符文本：

- `char` 类型的普通字符文本，例如 `'a'`
- `char` 类型的 UTF-8 字符文本（C++20 中的 `char8_t`），例如 `u8'a'`
- 类型 `wchar_t`的宽字符文本，例如 `L'a'`
- `char16_t` 类型的 UTF-16 字符文本，例如 `u'a'`
- `char32_t` 类型的 UTF-32 字符文本，例如 `U'a'`

用于字符文本的字符可以是除保留字符反斜杠（`\`）、单引号（`'`）和换行符以外的任何字符。 可以使用转义序列指定保留字符。 可以通过使用通用字符名称指定字符，只要类型的大小足以保留字符。

编码

字符文本根据其前缀以不同的方式进行编码。

- 没有前缀的字符文本是普通字符文本。 包含可在执行字符集中表示的单个字符、转义序列或通用字符名称的普通字符文本的值等于它在执行字符集中的编码数值。 包含多个字符、转义序列或通用字符名称的普通字符文本是多字符文本。 无法在执行字符集中表示的多字符文本或普通字符文本的类型为 `int`，其值由实现定义。 有关 MSVC，请参阅下面的 Microsoft 特定部分。

- 以 `L` 前缀开头的字符文本是宽字符文本。 包含单个字符、转义序列或通用字符名称的宽字符文本的值等于它在执行宽字符集中的编码数值，除非该字符文本在执行宽字符集中没有表示形式，在这种情况下，值由实现定义。 包含多个字符、转义序列或通用字符名称的宽字符文本的值由实现定义。 有关 MSVC，请参阅下面的 Microsoft 特定部分。

- 以 `u8` 前缀开头的字符文本是 UTF-8 字符文本。 如果包含单个字符、转义序列或通用字符名称的 UTF-8 字符文本的值可以由单个 UTF-8 代码单元（对应于 C0 控件和基本拉丁语 Unicode 块）表示，该值等于其 ISO 10646 码位值。 如果该值不能由单个 UTF-8 代码单元表示，则程序的格式不当。 包含多个字符、转义序列或通用字符名称的 UTF-8 字符文本是格式不当的。

- 以 `u` 前缀开头的字符文本是 UTF-16 字符文本。 如果包含单个字符、转义序列或通用字符名称的 UTF-16 字符文本的值可以由单个 UTF-16 代码单元（对应于基本多语言平面）表示，该值等于其 ISO 10646 码位值。 如果该值不能由单个 UTF-16 代码单元表示，则程序的格式不当。 包含多个字符、转义序列或通用字符名称的 UTF-16 字符文本是格式不当的。

- 以 `U` 前缀开头的字符文本是 UTF-32 字符文本。 包含单个字符、转义序列或通用字符名称的 UTF-32 字符文本的值等于其 ISO 10646 码位值。 包含多个字符、转义序列或通用字符名称的 UTF-32 字符文本是格式不当的。

转义序列

有三种类型的转义序列：简单、八进制和十六进制。 转义序列可为下列任一值：

展开表

价值	转义序列
换行符	\n
反斜杠	\\
水平制表符	\t
问号	? 或 \?
垂直制表符	\v
单引号	\'
退格符	\b
双引号	\"
回车符	\r

价值	转义序列
空字符	\0
换页符	\f
八进制	\ooo
警报（响铃）	\a
十六进制	\xhhh

八进制转义序列包含一个反斜杠，后跟 1 到 3 个八进制数字的序列。 如果在第三位数之前遇到八进制转义序列，该转义序列将在第一个不是八进制数字的字符处终止。 可能的最高八进制值为 `\377`。

十六进制转义序列包含一个反斜杠，后接 `x` 字符，再后接由一个或多个十六进制数字组成的序列。 将忽略前导零。 在普通或以 u8 为前缀的字符文本中，最高十六进制值为 0xFF。 在使用 L 或 u 前缀的宽字符文本中，最大的十六进制值为 0xFFFF。 在使用 U 前缀的宽字符文本中，最大的十六进制值为 0xFFFFFFFF。

此示例代码演示了一些使用普通字符文本的转义字符示例。 相同的转义序列语法对其他字符文本类型有效。

C++

```
#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
*/
```

反斜杠字符 (`\`) 在位于行末尾时将作为行继续符。 如果你希望反斜杠字符显示为字符文本，则必须在一行中键入两个反斜杠 (`\\`)。 有关行继续符的详细信息，请参阅 Phases of Translation。

Microsoft 专用

为了从窄多字符文本创建值，编译器将单引号之间的字符或字符序列转换为 32 位整数内的 8 位值。 文本中的多个字符根据需要从高序位到低序位填充相应字节。 然后编译器按照一般规则将整数转换为目标类型。 例如，为了创建 `char` 值，编译器采用低序位字节。 为了创建 `wchar_t` 或 `char16_t` 值，编译器采用低序位字。 如果在分配的字节或字上设置了任何位，则编译器会警告结果被截断。

C++

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

显示为包含超过三位数字的八进制转义序列被视为 3 位数八进制序列，之后的数字被视为多字符文本中的字符，这可能导致令人惊讶的结果。 例如：

C++

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

对于将包含非八进制字符的转义序列，其求值结果为全部由八进制字符组成的八进制序列，后跟剩余字符作为多字符文本中的后续字符。 如果第一个非八进制字符是十进制数字，则会生成警告 C4125。 例如：

C++

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

值高于 `\377` 的八进制转义序列会导致错误 C2022：“value-in-decimal”: 对于字符而言太大。

对于将具有十六进制和非十六进制字符的转义序列，其求值结果为多字符文本，其中包含全部由十六进制字符组成的十六进制转义序列，后跟非十六进制字符。 不包含十六进制数字的十六进制转义序列将导致编译器错误 C2153：“十六进制文本必须至少有一个十六进制数字”。

C++

```
char c6 = '\x0050'; // 'P'
```

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

如果以 `L` 为前缀的宽字符文本包含多字符序列，则值取自第一个字符，并且编译器会引发警告 C4066。 将忽略后续字符，这不同于等效普通多字符文本的行为。

C++

```
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
```

“特定于 Microsoft”部分到此结束。

通用字符名称

在字符字面量和原生（非原始）字符串字面量中，任何字符都可由通用字符名称表示。 通用字符名称由前缀 `\U` 后跟八位数 Unicode 码位组成，或者由前缀 `\u` 后跟四位数 Unicode 码位组成。 必须分别显示所有八个或四个数字，以组成一个格式正确的通用字符名称。

C++

```
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'
```

代理项对

通用字符名称不能对代理项码位范围 D800-DFFF 内的值进行编码。 对于 Unicode 代理项对，通过使用 `\UNNNNNNNN`（其中，NNNNNNNN 是字符的八位数码位）指定通用字符名称。 如果需要，编译器将生成一个代理项对。

在 C++03 中，语言只允许字符串子集通过其通用字符名称来表示，并且允许某些实际不表示任何有效 Unicode 字符的通用字符名称。 此错误已在 C++ 11 标准中解决。 在 C++ 11 中，字符以及字符串文本和标识符可以使用通用字符名称。 有关通用字符名称的详细信息，请参阅 Character Sets。 有关 Unicode 的详细信息，请参阅 Unicode。 有关代理对的详细信息，请参阅代理对与补充字符。

字符串文本

字符串文本表示字符序列，这些字符合起来可组成以 null 结尾的字符串。 字符必须放在双引号之间。 字符串文本有以下类型：

窄字符串文本

窄字符串文本是一个没有前缀且以双引号分隔、以 null 结尾的 `const char[n]` 类型的数组，其中 n 是数组的长度（以字节为单位）。 窄字符串文本可包含除双引号 (`"`)、反斜杠 (`\`) 或换行符以外的所有图形字符。 窄字符串文本还可包含上面列出的转义序列和装入一个字节中的通用字符名称。

C++

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

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

UTF-8 编码的字符串

UTF-8 编码的字符串是一个前缀为 u8 且以双引号分隔、以 null 结尾的 `const char[n]` 类型的数组，其中 n 是编码的数组的长度（以字节为单位）。 以 u8 为前缀的字符串文本可包含除双引号 (`"`)、反斜杠 (`\`) 或换行符以外的所有图形字符。 以 u8 为前缀的字符串文本还可包含上面列出的转义序列和任何通用字符名称。

C++20 引入了可移植的 **`char8_t`** （UTF-8 编码的 8 位 Unicode）字符类型。 在 C++20 中， `u8` 文本前缀指定 **`char8_t`** 而不是 **`char`** 的字符或字符串。

C++

```
// Before C++20
const char* str1 = u8"Hello World";
const char* str2 = u8"\U0001F607 is O:-)";
// C++20 and later
const char8_t* u8str1 = u8"Hello World";
const char8_t* u8str2 = u8"\U0001F607 is O:-)";
```

宽字符串文本

宽字符串是一个以 null 结尾且具有前缀“**`wchar_t`**”的常数 `L` 数组，其中包含除双引号 (`"`)、反斜杠 (`\`) 或换行符以外的所有图形字符。 宽字符串文本可包含上面列出的转义序列和任何通用字符名称。

C++

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

char16_t 和 char32_t (C++11)

C++11 引入了可移植的 `char16_t` （16 位 Unicode）和 `char32_t` （32 位 Unicode）字符类型：

C++

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

原始字符串文本 (C++11)

原始字符串是一个以 null 结尾的数组（属于任何字符类型），其中包括含双引号 (`"`)、反斜杠 (`\`) 或换行符在内的所有图形字符。 原始字符串通常用于使用字符类的正则表达式，还用于 HTML 字符串和 XML 字符串。 有关示例，请参阅以下文章： 关于 C++11 的 Bjarne Stroustrup 常见问题。

C++

```
// 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_utf8a = u8R"(An unescaped \ character)"; // Before C++20
const char8_t*  raw_utf8b = u8R"(An unescaped \ character)"; // C++20
const char16_t* raw_utf16 = uR"(An unescaped \ character)";
const char32_t* raw_utf32 = UR"(An unescaped \ character)";
```

分隔符是用户定义的最多包含 16 个字符的序列，它紧贴在原始字符串文本的左括号之前，紧跟在右括号之后。 例如，在 `R"abc(Hello"\()abc"` 中，分隔符序列为 `abc` ，字符串内容为 `Hello"\(` 。 你可使用分隔符来消除同时含有双引号和括号的原始字符串。 此字符串文本会导致编译器错误：

C++

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

但分隔符可以解决它。

C++

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

可以构造在源中包含换行的原始字符串文本（非转义字符）：

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

std::string 文本 (C++14)

`std::string` 文本是用户定义的文本（请参阅下文）的标准库实现，表示为 `"xyz"s`（具有 `s` 后缀）。 这种字符串文本根据指定的前缀生成 `std::string`、`std::wstring`、`std::u32string` 或 `std::u16string` 类型的临时对象。 如上所示不使用任何前缀时，会生成 `std::string`。 `L"xyz"s` 生成 `std::wstring`。 `u"xyz"s` 生成 std::u16string，`U"xyz"s` 生成 std::u32string。

```C++
//#include <string>
//using namespace std::string_literals;
string str{ "hello"s };
string str2{ u8"Hello World" };     // Before C++20
u8string u8str2{ u8"Hello World" }; // C++20
wstring str3{ L"hello"s };
u16string str4{ u"hello"s };
u32string str5{ U"hello"s };
```

`s` 后缀也可以用于原始字符串：

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

`std::string` 文本在 `std::literals::string_literals`string< 头文件的命名空间 > 中定义。 因为 `std::literals::string_literals`和 `std::literals` 都声明为 内联命名空间，所以会自动将 `std::literals::string_literals` 视为如同它直接属于命名空间 `std`。

字符串文本大小

对于 ANSI `char*` 字符串和其他单字节编码（而非 UTF-8），字符串文本的大小（以字节为单位）是字符的数量加上终止的空字符的 1。 对于所有其他字符串类型，大小不与字符数严格相关。 UTF-8 使用最多四个 `char` 元素对某些代码单位进行编码，编码为 UTF-16 的 或 `char16_t`

可以使用两个元素（针对总共四个字节）对单个 “代码单位”进行编码`wchar_t`。 本示例演示了宽字符串字面量的大小，单位是字节：

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

请注意，`strlen()` 和 `wcslen()` 不包括 null 终止字符的大小，该字符的大小等于字符串类型的元素大小：`char*` 或 `char8_t*` 字符串中是一个字节，`wchar_t*` 或 `char16_t*` 字符串中是两个字节，`char32_t*` 字符串中是四个字节。

串联后字符串文本的最大长度：

- 2022 年版本 17.0 之前的 Visual Studio：串联后字符串文本的最大长度为 65,535 字节。 这同时适用于窄字符串字面量和宽字符串字面量。
- 从 Visual Studio 2022 版本 17.0 开始：串联后字符串文本的最大长度仅受可用内存的限制。 但是，串联前的大小限制仍为 16,384 字节

修改字符串文本

因为字符串（不包括 `std::string` 文本）是常量，所以尝试修改它们（例如，`str[2] = 'A'`）会导致编译器错误。

Microsoft 专用

在 Microsoft C++ 中，可以使用字符串文本将指针初始化，使指针成为非常数 `char` 或 `wchar_t`。 此非常数初始化可以在 C99 代码中使用，但在 C++98 中已弃用，在 C++11 中已删除。 尝试修改该字符串将导致访问冲突，例如：

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

当你设置 /Zc:strictStrings（禁用字符串字面量类型转换）编译器选项且字符串文本转化为非常数字符指针时，可导致编译器发生错误。 我们建议将其用于符合标准的可移植代码。 使用 `auto` 关键字声明经过字符串文本初始化的指针也是一个很好的做法，因为它可以解析为正确（常数）的类型。 例如，此代码示例捕捉到一次在编译时写入字符串文本的尝试：

```
C++
```

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

在某些情况下，可以合并相同的字符串文本，节省可执行文件的空间。 字符串文本合并过程中，编译器将导致对特定字符串文本的所有引用都指向内存中的同一位置，而不是每次引用都指向一个单独的字符串文本实例。 若要启用字符串合并，请使用 /GF 编译器选项。

“特定于 Microsoft”部分到此结束。

串联相邻字符串文本

相邻宽或窄字符串文本是串联的。 声明如下：

C++

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

与此声明相同：

C++

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

也和此声明相同：

C++

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

使用嵌入式十六进制转义代码来指定字符串文本可能会导致意外的结果。 以下示例旨在创建包含 ASCII 5 字符、后跟字符 f、i、v 和 e 的字符串文本：

C++

```
"\x05five"
```

实际结果是十六进制 5F，它是一个下划线 ASCII 代码，后跟字符 i、v 和 e。 若要获得正确的结果，可以使用以下转义序列之一：

C++

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

`std::string` 文本（以及相关的 `std::u8string`、`std::u16string` 和 `std::u32string`）可与为 `+` 类型定义的 basic_string 运算符连接。 它们还可以通过与相邻字符串相同的方式进行串联。 在两种情况下，字符串编码和后缀都必须匹配：

C++

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

具有通用字符名称的字符串文本

本机（非原始）字符串文本可能使用通用字符名称来表示任何字符，只要通用字符名称可被编码为字符串类型中的一个或多个字符。 例如，表示扩展字符的通用字符名称不能以使用 ANSI 代码页的窄字符串进行编码，但可以使用一些多字节代码页中的窄字符串、UTF-8 字符串或宽字符串进行编码。 在 C++11 中，Unicode 支持由 `char16_t*` 和 `char32_t*` 字符串类型扩展，C++20 将其扩展为 `char8_t` 类型：

C++

```
// 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*     s3a = u8"😇 = \U0001F607 is O:-)"; // Before C++20
const char8_t*  s3b = u8"😇 = \U0001F607 is O:-)"; // C++20

// 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-)";
```

另请参阅

字符集
数值、布尔和指针文本
用户定义的文本

Last updated on 2025/11/07

用户定义的文本

在 C++ 中，文本有六个主要类别：整数、字符、浮点、字符串、布尔和指针。 从 C++11 开始，可以基于这些类别定义你自己的文本，以便为常见惯用语提供快捷语法，并提高类型安全性。 例如，假设有一个 `Distance` 类。 你可以将一个文本定义为表示公里，将另一个文本定义为表示英里，并通过编写以下内容帮助用户明确度量单位：`auto d = 42.0_km` 或 `auto d = 42.0_mi`。 用户定义的文本没有任何性能优势或劣势；它们的主要作用在于方便或实现编译时类型推断。 标准库具有 `std::string`、`std::complex` 以及 <chrono> 标头中的时间和持续时间操作单位的用户定义文本：

C++

```
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
```

用户定义的文本运算符签名

通过以下形式之一在命名空间范围定义 operator"" 来实现用户定义的文本：

C++

```
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
```

上例中的运算符名是你提供的任意占位符，但需要前导下划线。 （仅标准库才允许定义不带下划线的文本。）在返回类型中，你可以自定义文本执行的转换或其他操作。 此外，这些运算符中的任何一个都可定义为 `constexpr`。

加工的文本

在源代码中，任何文本（无论是否为用户定义的）实质上都是字母数字字符序列，例如 `101`、`54.7`、`"hello"` 或 `true`。 编译器将序列解释为整数、浮点、常量字符*字符串等。 接受编译器分配给文本值的任何类型作为输入的用户定义的文本非正式地称为“加工的文本”。 以上所有运算符（`_r` 和 `_t` 除外）均为加工的文本。 例如，文本 `42.0_km` 将绑定到签名与 _b 类似的运算符 _km；而文本 `42_km` 将绑定到签名与 _a 类似的运算符。

下面的示例演示用户定义的文本如何帮助用户明确其输入。 若要构造 `Distance`，用户必须通过使用相应的用户定义文本显式指定公里或英里。 也可以通过其他方式实现相同的结果，但用户定义的文本比其他方案简便。

C++

```
// 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;
}
```

文本数字必须使用十进制数。 否则数字将被解释为整数，而该类型与运算符不兼容。 对于浮点数输入，类型必须是 `long double`；而对于整数类型则必须是 `long long`。

原始文本

在原始的用户定义文本中，你定义的运算符将文本作为字符值的序列接受。 由你决定将该序列解释为数字、字符串还是其他类型。 在此页上方显示的运算符列表中，`_r` 和 `_t` 可用于定义原始文本：

C++

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

可使用原始文本来提供不同于编译器正常行为的输入序列的自定义解释。 例如，可以定义一段文本，用于将序列 `4.75987` 转换为自定义的十进制类型，而不是 IEEE 754 浮点类型。 原始文本（如加工的文本）还可用于输入序列的编译时验证。

示例：原始文本的限制

原始文本运算符和文本运算符模板仅适用于整型和浮点型用户定义文本，如下面的示例所示：

C++

```
#include <cstddef>
#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;
}
```

Output

```
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<===
```

Last updated on 2025/11/07

基本概念 (C++)

本部分介绍对于了解 C++ 而言至关重要的概念。 C 程序员会很熟悉其中许多概念，但其间存在一些细微差异，这些差异可能会导致意外的程序结果。 本文包含以下主题：

- C++ 类型系统
- 范围
- 翻译单元和链接
- main 函数和命令行参数
- 程序终止
- Lvalues 和 rvalues
- 临时对象
- 对齐
- Trivial、standard-layout 和 POD 类型

另请参阅

C++ 语言参考

Last updated on 2025/11/07

C++ 类型系统

在 C++ 中，*类型*的概念非常重要。 每个变量、函数自变量和函数返回值必须具有一个类型以进行编译。 此外，在计算表达式前，编译器会隐式给出每个表达式（包括文本值）的类型。 类型的一些示例包括内置类型，例如用于存储整数值的 `int`，用于存储浮点值的 `double`，或用于存储文本的标准库类型，如类 std::basic_string。 可以通过定义 `class` 或 `struct` 创建自己的类型。 该类型指定为变量（或表达式结果）分配的内存量。 该类型还指定可存储的值类型、编译器如何解释这些值中的位模式以及可以对它们执行的操作。 本文包含对 C++ 类型系统的主要功能的非正式概述。

术语

标量类型：包含定义范围的单个值的类型。 标量包括算术类型（整型或浮点值）、枚举类型成员、指针类型、指针到成员类型以及 `std::nullptr_t`。 基本类型通常是标量类型。

复合类型：不是标量类型的类型。 复合类型包括数组类型、函数类型、类（或结构）类型、联合类型、枚举、引用和指向非静态类成员的指针。

变量：数据量的符号名称。 该名称可用于访问它在整个定义代码范围内引用的数据。 在 C++ 中，*变量*通常用于指标量数据类型的实例，而其他类型的实例通常称为*对象*。

对象：为了简洁一致，本文使用术语“*对象*”指代类或结构的任何实例。 在一般意义上使用时，它包括所有类型，甚至标量变量。

POD 类型（纯旧数据）：C++ 中的此类非正式数据类型类别是指作为标量（参见基础类型部分）的类型或 POD 类。 POD 类没有不是 POD 的静态数据成员，没有用户定义的构造函数、用户定义的析构函数或用户定义的赋值运算符。 此外，POD 类无虚函数、基类、私有的或受保护的非静态数据成员。 POD 类型通常用于外部数据交换，例如与用 C 语言编写的模块（仅具有 POD 类型）进行的数据交换。

指定变量和函数类型

C++ 既是*强类型*语言，也是*静态类型*语言；每个对象都有一个类型，并且该类型永远不会更改。 在代码中声明变量时，你必须显式指定其类型或使用 `auto` 关键字指示编译器通过初始值设定项推断类型。 在代码中声明函数时，必须指定其返回值的类型以及每个自变量的类型。 如果函数未返回任何值，则使用返回值类型 `void`。 例外情况是，当使用允许任意类型自变量的函数模板时。

在你首次声明变量后，稍后无法更改其类型。 但是，你可以将变量的值或函数的返回值复制到其他类型的另一个变量中。 此类操作称作“类型转换”，这些操作有时很必要，但也是造成数据丢失或不正确的潜在原因。

在声明 POD 类型的变量时，强烈建议你将其*初始化*，也就是为其指定初始值。 在初始化某个变量之前，该变量会有一个“垃圾”值，该值包含之前正好位于该内存位置的位数。 这是需要注意的 C++ 的一个重要方面，尤其是当你使用另一种语言来处理初始化时。 如果声明非 POD 类类型的变量，则构造函数会处理初始化。

下面的示例演示了一些简单变量声明，并分别对它们进行了说明。 该示例还演示了编译器如何使用类型信息允许或禁止对变量进行某些后续操作。

C++

```
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 intializing 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.
```

基本（内置）类型

不同于某些语言，C++ 中不存在派生所有其他类型的通用基类型。 该语言包括许多基本类型（也称为“内置类型”）。 这些类型包括数值类型，例如 `int`、`double`、`long`、`bool`，以及分别针对 ASCII 和 UNICODE 字符的 `char` 和 `wchar_t` 类型。 大多数基础类型（`bool`、`double`、`wchar_t` 和相关类型除外）都具有 `unsigned` 版本，这些版本修改了变量可存储的值的范围。 例如，`int` 存储 32 位已签名整数，可表示介于 -2,147,483,648 和 2,147,483,647 之间的值。 `unsigned int` 也存储为 32 位，它可存储介于 0 和 4,294,967,295 之间的值。 可能的值的总数在每种情况下都相同；仅范围不同。

编译器可识别这些内置类型，并具有内置规则，用于控制可对其执行的操作，以及如何将其转换为其他基本类型。 有关内置类型及其大小和数值限制的完整列表，请参阅内置类型。

下图显示了 Microsoft C++ 实现中内置类型的相对大小：

下表列出了 Microsoft C++ 实现中最常使用的基本类型及其大小：

展开表

类型	大小	注释
`int`	4 个字节	整数值的默认选择。
`double`	8 字节	浮点值的默认选择。
`bool`	1 个字节	表示可为 true 或 false 的值。
`char`	1 个字节	用于早期 C 样式字符串或 std:: 字符串对象中无需转换为 UNICODE 的 ASCII 字符。
`wchar_t`	2 个字节	表示可能以 UNICODE 格式进行编码的“宽”字符值（Windows 上为 UTF-16，其他操作系统上可能不同）。 `wchar_t` 是 `std::wstring` 类型字符串中使用的字符类型。
`unsigned char`	1 个字节	C++ 没有内置字节类型。 使用 `unsigned char` 来表示字节值。
`unsigned int`	4 个字节	位标志的默认选项。
`long long`	8 字节	表示更大的整数值范围。

其他 C++ 实现可能对某些数值类型使用不同的大小。 若要详细了解 C++ 标准所需的大小和大小关系，请参阅内置类型。

`void` 类型

`void` 类型是一种特殊类型；不能声明 `void` 类型的变量，但可声明 `void *` 类型的变量（指向 `void` 的指针），有时当分配原始（非类型化）内存时这样做很有必要。 但是，指向 `void` 的指针不是类型安全的指针，建议不要在新式 C++ 中使用它们。 在函数声明中，`void` 返回值意味着该函数不返回值；将其用作返回类型是 `void` 的常见且可接受的用法。 尽管 C 语言需要含零个参数的函数以在参数列表中声明 `void`（例如 `fn(void)`），但新式 C++ 中建议不要这样做，而是应声明无参数函数 `fn()`。 有关详细信息，请参阅类型转换和类型安全性。

`const` 类型限定符

任何内置或用户定义的类型都可由 `const` 关键字限定。 此外，成员函数可受到 `const` 限定，甚至可重载 `const`。 `const` 类型的值在初始化后将无法修改。

C++

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

`const` 限定符在函数和变量声明中使用广泛，而“常量正确性”是 C++ 中的一个重要概念；它实质上表示使用 `const` 来确保在编译时不会无意中修改值。 有关详细信息，请参阅 const。

`const` 类型与其非 `const` 版本截然不同；例如，`const int` 是与 `int` 截然不同的类型。 如果发生必须从变量中移除常量性的这类少数情况，可使用 C++ `const_cast` 运算符。 有关详细信息，请参阅类型转换和类型安全性。

字符串类型

严格来说，C++ 语言没有内置的字符串类型；`char` 和 `wchar_t` 存储单个字符 - 必须声明这些类型的数组来估计字符串，从而将一个终止 null 值（例如，ASCII `'\0'`）添加到最后一个有效字符数后的数组元素（也称为“C 样式字符串”）。 C 样式字符串需要编写更多的代码或者需要使用外部字符串实用工具库函数。 但是，在新式 C++ 中，我们具有标准库类型 `std::string`（用于 8 位 `char` 型字符串）或 `std::wstring`（用于 16 位 `wchar_t` 型字符串）。 这些 C++ 标准库容器可被视为本机字符串类型，因为它们是所有兼容 C++ 生成环境包含的标准库的一部分。 使用 `#include <string>` 指令使这些类型在你的程序中可用。 （如果使用的是 MFC 或 ATL，还可使

用 `CString` 类，但其不符合 C++ 标准。）建议不要在新式 C++ 中使用 null 终止字符数组（前面提到的 C 样式字符串）。

用户定义类型

在定义 `class`、`struct`、`union` 或 `enum` 时，该构造会在代码的其余部分使用，如同它是一个基础类型一样。 它具有内存的已知大小以及一些有关可以如何在程序生命期内将其用于编译时检查和运行时的规则。 基本内置类型和用户定义的类型之间的主要区别如下：

- 编译器没有用户定义的类型的内置知识。 它在编译过程中首次遇到此定义时就学习了此类型。
- 通过定义（通过重载）适当的运算符作为类成员或非成员函数，可以指定可对你的类型执行的操作以及你的类型转换为其他类型的方式。 有关详细信息，请参阅函数重载

指针类型

与早期版本的 C 语言一样，C++ 继续通过使用特殊声明符 `*`（星号）声明指针类型的变量。 指针类型在存储实际数据值的内存中存储位置地址。 在新式 C++ 中，这些指针类型称为*原始指针*，它们通过特殊运算符在代码中访问：`*`（星号）或 `->`（带大于号的短划线，通常称为*箭头*）。 此内存访问操作称为*取消引用*。 所使用的运算符取决于是取消引用指向标量指针的指针，还是取消引用指向对象中成员的指针。

使用指针类型很长时间以来都是 C 和 C++ 程序开发的最具挑战性和最难以理解的方面之一。 本节概述了一些事实和实践，以帮助你在需要时使用原始指针。 但是，在新式 C++ 中，由于智能指针的演变（在本部分末尾进行了更多讨论），因此不再需要（或建议）将原始指针用于对象所有权。 使用原始指针来观察对象仍然是有用和安全的。 但是，如果必须将其用于对象所有权，则需要谨慎操作，并仔细考虑如何创建和销毁其拥有的对象。

首先应知道的是，原始指针变量声明只分配足够的内存来存储地址：指针在取消引用时引用的内存位置。 指针声明不会分配存储数据值所需的内存。 （该内存也称为*后备存储*。）换言之，通过声明原始指针变量，将创建内存地址变量而非实际数据变量。 如果在确保指针变量包含后备存储的有效地址前取消对其的引用，则将导致程序发生未定义行为（通常为严重错误）。 下面的示例演示了此种错误：

C++

```
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.
```

该示例取消引用指针类型，未分配用于存储实际整数数据的任何内存或向其分配有效内存地址。 下面的代码更正这些错误：

```
C++
    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".
```

已纠正的代码示例使用本地堆栈内存创建 `pNumber` 指向的后备存储。 我们使用基本类型，以求简单。 实际上，指针的后备存储通常是用户定义类型，这些类型通过使用 关键字表达式（在 C 样式编程中，使用旧的 C 运行时库函数）动态分配到称为“**`new`**”（或“`malloc()`”）的内存区域中。 分配后，这些变量通常指“*对象*”，尤其是基于类定义这些变量的情况下。 使用 **`new`** 分配的内存必须由相应的 **`delete`** 语句删除（如果使用 `malloc()` 函数进行关联，则使用 C 运行时函数 `free()` 执行删除操作）。

但是，很容易忘记删除动态分配的对象（特别是在复杂代码中），这会导致产生名为“*内存泄漏*”的资源 Bug。 为此，建议你不要在新式 C++ 中使用原始指针。 在智能指针中包装原始指针几乎总是更好的，该指针在调用其析构函数时自动释放内存。 （也就是说，当代码超出智能指针的范围时。）通过使用智能指针，几乎消除了 C++ 程序中的一系列 bug。 在下面的示例中，假定 `MyClass` 是具有公共方法 `DoSomeWork();` 的用户定义的类型

```
C++
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.
```

有关智能指针的详细信息，请参阅智能指针。

有关指针转换的详细信息，请参阅类型转换和类型安全性。

有关指针的一般性详细信息，请参阅指针。

Windows 数据类型

在 C 和 C++ 的经典 Win32 编程中，大多数函数使用 Windows 特定的 Typedef 和 `#define` 宏（在 `windef.h` 中定义）来指定参数类型和返回值。 这些 Windows 数据类型通常是为 C/C++ 内置类型提供的特殊名称（别名）。 有关这些 typedef 和预处理器定义的完整列表，请参阅 Windows 数据类型。 其中一些 typedef（例如 `HRESULT` 和 `LCID`）很有用且具有描述性。 `INT` 等其他类型没有特殊含义，只是基础 C++ 类型的别名。 其他 Windows 数据类型的名称自 C 编程和 16 位处理器得到保留，并且在现代硬件或操作系统中不具有目的和意义。 还有与 Windows 运行时库相关的特定数据类型，它们在列表中显示为 Windows 运行时基础数据类型。 在新式 C++ 中，一般准则是首选 C++ 基本类型，除非 Windows 类型传达一些有关如何解释值的额外意义。

更多信息

有关 C++ 类型系统的详细信息，请参阅以下文章。

值类型
描述值类型以及与其使用相关的问题。

类型转换和类型安全
描述常见类型转换问题并说明如何避免这些问题出现。

另请参阅

欢迎回到 C++
C++ 语言参考
C++ 标准库

Last updated on 2025/11/07

范围 (C++)

声明类、函数或变量等程序元素时，其名称只能在程序的某些部分“查看”和使用。 名称可见的上下文称为其*范围*。 例如，如果在函数中声明变量 `x`，则 `x` 仅在该函数正文中可见。 它具有*局部范围*。 程序中可能存在具有相同名称的其他变量；只要它们位于不同的范围，就不会违反“一个定义规则”，也不会引发任何错误。

对于自动非静态变量，范围还确定它们在程序内存中创建和销毁的时间。

有六种范围：

- **全局范围**，全局名称是在任何类、函数或命名空间之外声明的名称。 但是，在 C++ 中，即使是这些名称也具有隐式全局命名空间。 全局名称的范围从声明点扩展到声明文件末尾。 对于全局名称，可见性也受链接规则的约束，这些规则确定名称是否在程序中的其他文件中可见。

- **命名空间范围**，在命名空间中声明的名称（在任何类或枚举定义或函数块之外）从其声明点到命名空间末尾可见。 命名空间可以在跨不同文件的多个块中定义。

- **局部范围**，在函数或 lambda 中声明的名称（包括参数名称）具有局部范围。 它们通常被称为“局部变量”。 它们仅从声明点到函数或 lambda 正文的末尾可见。 局部范围是一种块范围，本文稍后将对此进行讨论。

- **类范围**，类成员的名称具有类范围，该范围在整个类定义中扩展，与声明点无关。 类成员可访问性由 `public`、`private` 和 `protected` 关键字进一步控制。 只能使用成员选择运算符（`.` 或 `->`）或指向成员的指针运算符（`.*` or `->*`）访问公共或受保护成员。

- **语句范围**，在 `for`、`if`、`while` 或 `switch` 语句中声明的名称在语句块结束之前可见。

- **函数范围**，标签具有函数范围，这意味着它在整个函数正文中甚至在声明点之前都是可见的。 函数范围允许在声明 `goto cleanup` 标签之前编写 `cleanup` 等语句。

隐藏名称

可通过在封闭块中声明名称来隐藏该名称。 在下图中，在内部块中重新声明 `i`，从而隐藏与外部块范围中的 `i` 关联的变量。

块范围和名称隐藏

来自图中显示的程序的输出为：

C++

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

> ⓘ **注意**
>
> 自变量 `szWhat` 被视为处于函数的范围内。 因此，它被当做就像已在函数的最外层块中声明一样。

隐藏类名

通过声明同一范围内的函数、对象或变量或枚举器，可以隐藏类名称。 但是，在关键字 `class` 作为前缀时仍可以访问类名称。

C++

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

> ⓘ **注意**
>
> 在任何位置调用类名称（`Account`），关键字类都必须用于将其与全局范围内的变量 Account 区分开来。 当类名出现在范围解析运算符 (::) 的左侧时，此规则不适用。 在范围解析运算符的左侧的名称始终被视为类名称。

下面的示例演示如何使用 `Account` 关键字声明指向类型 `class` 的对象的指针：

C++

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

前面的语句中的初始值设定项（括号内）的 `Account` 具有全局范围；它的类型为 `double`。

> ⓘ **注意**
>
> 此示例中所示的标识符名称的重用被视为较差的编程样式。

有关类对象的声明和初始化的信息，请参阅类、结构和联合。 有关使用 `new` 和 `delete` 自由存储运算符的信息，请参阅new 和 delete 运算符。

隐藏具有全局范围的名称

通过在块范围中显式声明相同的名称，可以隐藏带全局范围的名称。 但是，可以使用范围解析运算符（`::`）访问全局范围名称。

C++

```
#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";
}
```

Output

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

另请参阅

基本概念

Last updated on 2025/11/07

头文件 (C++)

必须在使用变量、函数、类等程序元素的名称之前对其进行声明。 例如，不能在没有声明“x”之前编写 `x = 42`。

C++

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

声明告知编译器，元素是 `int`、`double`、函数、 还是其他内容。 `class` 此外，必须在使用每个名称时所在的每个 .cpp 文件中（直接或间接）声明每个名称。 编译程序时，每个 .cpp 文件都会独立编译为一个编译单元。 编译器不知道在其他编译单元中声明了哪些名称。 这意味着，如果你定义类、函数或全局变量，则必须在使用它的每个附加 .cpp 文件中提供对它的声明。 在所有文件中，对它的每个声明必须完全相同。 当链接器尝试将所有编译单元合并成单个程序时，出现轻微的不一致会导致错误或意外行为。

为了最大程度地减少出错的可能性，C++ 采用了使用头文件来包含声明的约定。 在一个头文件中进行声明，然后在每个 .cpp 文件或其他需要该声明的头文件中使用 #include 指令。 #include 指令在编译之前将头文件的副本直接插入 .cpp 文件中。

> ⓘ **注意**
>
> 在 Visual Studio 2019 中，C++20 模块功能作为头文件的改进和最终替代引入。 有关详细信息，请参阅 **C++ 中的模块概述**。

示例

以下示例演示了一种声明类的常见方法，然后在另一源文件中使用它。 我们将从头文件 `my_class.h` 开始。 它包含类定义，但请注意，定义不完整；未定义成员函数 `do_something`：

C++

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

}
```

接下来，创建一个实现文件（通常使用 .cpp 或类似的扩展名）。 我们将调用文件 my_class.cpp，并为成员声明提供定义。 我们为“my_class.h”文件添加一个 `#include` 指令，以便立刻将 my_class 声明插入到 .cpp 文件中。我们包括 `<iostream>`，用于拉入 `std::cout` 的声明。 请注意，引号用于源文件所在目录中的头文件，尖括号用于标准库标头。 此外，许多标准库标头没有 .h 或任何其他文件扩展名。

在实现文件中，可以选择使用 **`using`** 语句来避免使用“N::”或“std::”限定每个提及的“my_class”或“cout”。 不要在头文件中放置 **`using`** 语句！

C++

```
// 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;
}
```

现在，我们可以在另一个 .cpp 文件中使用 `my_class`。 我们 #include 头文件，以便编译器拉入声明。 所有编译器都需要知道的是，my_class 是一个类，它有一个名为 `do_something()` 的公共成员函数。

C++

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

using namespace N;

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

编译器完成将每个 .cpp 文件编译为 .obj 文件的操作后，会将 .obj 文件传递给链接器。 链接器合并对象文件时，会发现 my_class 的一个定义；它位于为 my_class.cpp 生成的 .obj 文件中，生成成功。

Include 防范

通常，头文件有一个 include 防范或 `#pragma once` 指令，用于确保它们不会多次插入到单个 .cpp 文件中。

C++

```
// 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 */
```

要放入头文件的内容

由于一个头文件可能会被多个文件执行 include 操作，因此它不能包含可能生成多个同名定义的定义。 不允许以下操作，否则会被视为非常糟糕的做法：

- 命名空间或全局范围内的内置类型定义
- 非内联函数定义
- 非常量变量定义
- 聚合定义
- 未命名的命名空间
- using 指令

使用 `using` 指令不一定会导致错误，但可能会导致问题，因为它将命名空间引入每个直接或间接包含该标头的 .cpp 文件中的范围。

示例头文件

以下示例显示了头文件中允许的各种声明和定义：

C++

```
// sample.h
#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;
}
```

Last updated on 2025/11/07

翻译单元和链接

在 C++ 程序中，符号（例如变量或函数名称）可以在其范围内进行任意次数的声明。 但是，一个符号只能被定义一次。 这就是“单一定义规则”(ODR)。 声明在程序中引入（或重新引入）一个名称，以及足够的信息，以便以后将该名称与定义联系起来。 定义引入一个名称，并提供创建它所需的全部信息。 如果名称表示变量，则定义会显式创建存储并进行初始化。 函数定义由签名和函数体组成。 类定义由类名和一个列出所有类成员的块组成。 （成员函数体可以选择在另一个文件中单独定义。）

下面的示例演示了一些声明：

C++

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

下面的示例演示了一些定义：

C++

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

一个程序包括一个或多个翻译单元。 一个翻译单元由一个实现文件及其直接或间接包含的所有标头组成。 实现文件通常具有文件扩展名 `.cpp` 或 `.cxx`。 头文件通常具有扩展名 `.h` 或 `.hpp`。 每个翻译单元由编译器独立编译。 编译完成后，链接器会将编译后的翻译单元合并到单个程序中。 ODR 规则的冲突通常显示为链接器错误。 在多个翻译单元中定义同一名称时，将发生链接器错误。

通常，使变量在多个文件中可见的最佳方式是在头文件中声明它。 然后，在需要声明的每个 `#include` 文件中添加一个 `.cpp` 指令。 通过在标头内容周围添加 include 防范，可以确保标头声明的名称对每个翻译单元只声明一次。 仅在一个实现文件中定义名称。

在 C++20 中，模块作为头文件的改进替代方法引入。

在某些情况下，可能需要在 `.cpp` 文件中声明全局变量或类。 在这些情况下，你需要一种方法来告知编译器和链接器名称所具有的链接类型。 链接的类型指定对象的名称是仅在一个文件中可见，还是在所有文件中可见。 链接的概念仅适用于全局名称。 链接的概念不适用于在一定范围内声明的名称。 范围是由一组封闭的大括号指定的，例如在函数或类的定义中。

外部链接与内部链接

Free 函数是在全局范围或命名空间范围内定义的函数。 默认情况下，非常量全局变量和 Free 函数具有外部链接；它们在程序中的任何翻译单元内可见。 其他任何全局对象都不能具有该名称。 具有内部链接或无链接的符号仅在声明它的翻译单元内可见。 当一个名称具有内部链接时，同一名称可能存在于另一个翻译单元中。 类定义或函数体中声明的变量没有链接。

如果要强制一个全局名称具有内部链接，可以将它显式声明为 `static`。 此关键字将它的可见性限制在声明它的同一翻译单元内。 在此上下文中， `static` 表示与应用于局部变量时不同的内容。

默认情况下，以下对象具有内部链接：

- `const` 对象
- `constexpr` 对象
- `typedef` 对象
- 命名空间范围中的 `static` 对象

若要为 `const` 对象提供外部链接，请将其声明为 `extern` 并为其赋值：

C++

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

有关详细信息，请参阅 extern。

另请参阅

基本概念

Last updated on 2025/11/07

`main` 函数和命令行参数

所有 C++ 程序都必须具有 `main` 函数。 如果尝试在没有 `main` 函数的情况下编译 C++ 程序，编译器将引发错误。 （动态链接库和 static 库没有 `main` 函数。） `main` 函数是源代码开始执行的位置，但在程序进入 `main` 函数之前，没有显式初始值设定项的所有 static 类成员都设为零。 在 Microsoft C++ 中，全局 static 对象在进入 `main` 前也进行初始化。 一些限制适用于 `main` 函数，而不适用于任何其他 C++ 函数。 `main` 函数：

- 无法重载（请参阅函数重载）。
- 无法声明为 **`inline`**。
- 无法声明为 **`static`**。
- 无法提取其地址。
- 无法从程序调用。

`main` 函数签名

`main` 函数没有声明，因为它内置于语言中。 如果有，则 `main` 的声明语法如下所示：

C++

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

如果 `main` 中未指定返回值，编译器会提供零作为返回值。

标准命令行参数

`main` 的参数可进行方便的命令行分析。 `argc` 和 `argv` 的类型由语言定义。 名称 `argc` 和 `argv` 是传统名称，但你可以按自己的意愿命名。

自变量定义如下所示：

`argc`
包含 *argv* 后面的参数计数的整数。 *argc* 参数始终大于或等于 1。

`argv`
表示由杂注用户输入的命令行自变量的以 null 结尾的字符串的数组。 按照约定， `argv[0]` 是用于调用程序的命令。 `argv[1]` 是第一个命令行参数。 命令行的最后一个参数是 `argv[argc - 1]`，并且 `argv[argc]` 始终为 NULL。

有关如何禁用命令行处理的信息，请参阅自定义 C++ 命令行处理。

> ⓘ **注意**
>
> 按照约定，`argv[0]` 是程序的文件名。 但在 Windows 上，可以使用 CreateProcess 来生成进程。 如果同时使用了第一个和第二个参数（*`lpApplicationName`* 和 *`lpCommandLine`*），则 `argv[0]` 可能不是可执行名称。 可使用 GetModuleFileName 来检索可执行名称及其完全限定的路径。

特定于 Microsoft 的扩展

以下部分介绍特定于 Microsoft 的行为。

`wmain` 函数和 `_tmain` 宏

如果将源代码设计为使用 Unicode 宽字符，则可以使用特定于 Microsoft 的 `wmain` 入口点，即宽字符版的 `main`。 下面是 `wmain` 的有效声明语法：

C++

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

还可以使用特定于 Microsoft 的 `_tmain`，它是 *`tchar.h`* 中定义的预处理器宏。 除非定义了 `_tmain`，否则 `main` 解析为 `_UNICODE`。 在该示例中，`_tmain` 将解析为 `wmain`。 对于需要分别生成窄字符集和宽字符集的代码来说，`_tmain` 宏和以 `_t` 开头的其他宏非常有用。 有关详细信息，请参阅使用一般文本映射。

从 `void` 返回 main

`main` 和 `wmain` 函数作为 Microsoft 扩展，可以声明为返回 **`void`**（没有返回值）。 此扩展在其他一些编译器中也可用，但不建议使用它。 当 `main` 不返回值时，它可用于保持对称。

如果将 `main` 或 `wmain` 声明为返回 **`void`**，则无法使用 exit 语句将 return 代码返回到父进程或操作系统中。 若要在将 exit 或 `main` 声明为 `wmain` 时返回 **`void`** 代码，则必须使用 exit 函数。

`envp` 命令行参数

`main` 或 `wmain` 签名允许可选的 Microsoft 特定扩展访问环境变量。 此扩展在 Windows 和 UNIX 系统的其他编译器中也很常见。 名称 *`envp`* 是传统名称，但你可以根据自己的意愿命名环境参

数。 下面是包含环境参数的参数列表的有效声明：

C++

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

envp

可选 *envp* 参数是表示用户环境中设置的变量的字符串数组。 该数组由 NULL 项终止。 它可以声明为指向 **char** (`char *envp[]`) 的指针数组，也可以声明为一个指针来指向多个指向 **char** (`char **envp`) 的指针。 如果程序使用 `wmain` 而不是 `main`，请使用 **wchar_t** 数据类型而不是 **char**。

传递给 `main` 和 `wmain` 的环境块是当前环境的"冻结"副本。 如果随后通过调用 `putenv` 或 `_wputenv` 来更改环境，则当前环境（由 `getenv` 或 `_wgetenv` 以及 `_environ` 或 `_wenviron` 变量返回）将发生更改，但 *envp* 指向的块不会更改。 有关如何禁用环境处理的更多信息，请参阅自定义 C++ 命令行处理。 *envp* 参数与 C89 标准兼容，但与 C++ 标准不兼容。

`main` 的示例参数

下面的示例演示如何使用 *argc* 的 *argv*、*envp* 和 `main` 变量：

C++

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

using namespace std;
int main( int argc, char *argv[], char *envp[] )
{
    bool numberLines = false;    // 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 )
         numberLines = true;

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

分析 C++ 命令行自变量

Microsoft C/C++ 代码使用的命令行分析规则特定于 Microsoft。 在解释操作系统命令行上给出的参数时，运行时启动代码使用这些规则：

- 参数用空白分隔，空白可以是一个空格或制表符。
- 第一个参数（`argv[0]`）是经过专门处理的。 它表示程序名称。 因为它必须是有效的路径名，因此允许用双引号（`"`）括起来一些部分。 双引号不包含在 `argv[0]` 输出中。 用双引号括起来的部分可以防止将空格或制表符解释为参数的末尾。 此列表中的后续规则不适用。
- 将双引号括起来的字符串解释为单个参数，哪怕其中可能包含空格字符。 带引号的字符串可以嵌入在自变量内。 插入点（`^`）未被识别为转义字符或者分隔符。 在带引号的字符串中，一对双引号被解释为单个转义的双引号。 如果在找到右双引号之前命令行结束，则到目前为止读取的所有字符都将输出为最后一个参数。
- 前面有反斜杠的双引号（`\"`）被解释为原义双引号（`"`）。
- 反斜杠按其原义解释，除非它们紧位于双引号之前。
- 如果偶数个反斜杠后跟双引号，则每对反斜杠（`\`）中有一个反斜杠（`argv`）被置于 `\\` 数组中，而双引号（`"`）被解释为字符串分隔符。
- 如果奇数个反斜杠后跟双引号，则每对反斜杠（`\`）中有一个反斜杠（`argv`）被置于 `\\` 数组中。 双引号由剩余反斜杠解释为转义序列，导致原义双引号（`"`）被置于 `argv` 中。

命令行参数分析示例

以下程序演示了如何传递命令行参数：

C++

```
// 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";
}
```

命令行分析结果

下表显示了示例输入和预期输出，演示了前面列表中的规则。

展开表

命令行输入	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`
`a"b"" c d`	`ab" c d`		

通配符扩展

Microsoft 编译器根据需要允许你使用*通配符*字符、问号 (`?`) 和星号 (`*`)，以在命令行上指定文件名和路径参数。

命令行参数由运行时启动代码中的内部历程处理，默认情况下，该例程不会将通配符扩展到 `argv` 字符串数组的单独字符串中。 通过将 *`setargv.obj`* 文件（对 *`wsetargv.obj`* 来说为 `wmain` 文件）包括在 **`/link`** 编译器选项或 **`LINK`** 命令行中，可以启用通配符扩展。

有关运行时启动链接器选项的详细信息，请参阅链接选项。

自定义 C++ 命令行处理

如果程序不采用命令行参数，则可以取消命令行处理例程来节省少量空间。 若要禁止使用该方法，请在 *`noarg.obj`* 编译器选项或 `main` 命令行中包含 `wmain` 文件（用于 **`/link`** 和 **`LINK`**）。

同样，如果从不通过 *`envp`* 参数访问环境表，则可以取消内部环境处理例程。 若要禁止使用该方法，请在 *`noenv.obj`* 编译器选项或 `main` 命令行中包含 `wmain` 文件（用于 **`/link`** 和 **`LINK`**）。

程序可以调用 C 运行时库中的 `spawn` 或 `exec` 系列例程。 如果是这样，则不应取消环境处理例程，因为可使用它将环境从父进程传递到子进程中。

另请参阅

基本概念

Last updated on 2025/11/07

C++ 程序终止

在 C++ 中，可以通过以下方式退出程序：

- 调用 exit 函数。
- 调用 abort 函数。
- 从 return 执行 `main` 语句。

`exit` 函数

exitstdlib.h< 中声明的 > 函数将终止 C++ 程序。 作为 `exit` 的自变量提供的值将作为程序的返回代码或退出代码返回到操作系统。 按照约定，返回代码为零表示该程序已成功完成。 可以使用同样在 `EXIT_FAILURE`stdlib.h`EXIT_SUCCESS` 中定义的常量 < 和 > 来指示程序是成功还是失败。

`abort` 函数

同样在标准包含文件 abortstdlib.h< 中声明的 > 函数用于终止 C++ 程序。 `exit` 与 `abort` 之间的差异在于， `exit` 允许执行 C++ 运行时终止处理（调用全局对象析构函数）。 `abort` 可立即终止程序。 `abort` 函数绕过初始化的全局静态对象的一般析构过程。 它还绕过使用 atexit 函数指定的任何特殊处理。

Microsoft 专用：出于 Windows 兼容性原因， `abort` 的 Microsoft 实现可能允许 DLL 终止代码在某些情况下运行。 有关详细信息，请参阅 abort。

`atexit` 函数

使用 atexit 函数指定在程序终止之前执行的操作。 在执行退出处理函数之前，不会销毁在调用 `atexit` 之前初始化的任何全局静态对象。

`return` 中的 `main` 语句

使用 `return` 语句可以从 `main` 指定一个返回值。 return 中 `main` 语句的行为首先类似于任何其他 `return` 语句。 任何自动变量都将被销毁。 然后， `main` 以返回值作为参数调用 `exit`。 请考虑以下示例：

C++

```
// return_statement.cpp
#include <stdlib.h>
```

```
struct S
{
    int value;
};
int main()
{
    S s{ 3 };

    exit( 3 );
    // or
    return 3;
}
```

前面示例中的 `exit` 和 `return` 语句具有类似的行为。 两者都会终止程序并向操作系统返回值 3。 不同之处在于，`exit` 不会销毁自动变量 `s`，而 `return` 语句会销毁。

通常，C++ 需要具有 `void` 之外的返回类型的函数返回一个值。 `main` 函数是一个异常；它可以在没有 `return` 语句的情况下结束。 在这种情况下，它会将特定于实现的值返回到调用过程。（默认情况下，MSVC 返回 0。）

线程和静态对象的销毁

直接调用 `exit` 时（或在 `return` 的 `main` 语句之后调用它时），将销毁与当前线程关联的线程对象。 然后，按与初始化相反的顺序销毁静态对象（在调用指定给 `atexit` 的函数（如果有）之后）。 以下示例演示如何进行此类初始化和清理工作。

示例

在下面的示例中，在进入 `sd1` 之前，将创建和初始化静态对象 `sd2` 和 `main`。 使用 `return` 语句终止此程序后，首先销毁 `sd2`，然后销毁 `sd1`。 `ShowData` 类的析构函数将关闭与这些静态对象关联的文件。

C++

```
// 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" );
}
```

另一种编写此代码的方式为，使用块范围声明 `ShowData` 对象，这将在它们超出范围时时将其隐式销毁：

C++

```
int main() {
   ShowData sd1( "CON" ), sd2( "hello.dat" );

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

另请参阅

main 函数和命令行参数

Last updated on 2025/11/07

Lvalues 和 Rvalues (C++)

每个 C++ 表达式都有一个类型，属于值类别。 值类别是编译器在表达式计算期间创建、复制和移动临时对象时必须遵循的规则的基础。

C++17 标准对表达式值类别的定义如下：

- glvalue 是一个表达式，它的计算可以确定对象、位域或函数的标识。
- prvalue 是一个表达式，它的计算可以初始化对象或位域，或计算运算符的操作数值，这是由它出现的上下文所指定的。
- xvalue 是一个 glvalue，表示一个对象或位域，该对象或位域的资源可重复使用（通常是因为它接近其生存期的末尾）。 示例：某些涉及 rvalue 引用 (8.3.2) 的类型的表达式会生成 xvalue，例如对返回类型为 rvalue 引用或强制转换为 rvalue 引用类型的函数的调用。
- *lvalue* 为非 xvalue 的 glvalue。
- rvalue 是一个 prvalue 或 xvalue。

下图阐释了各类别之间的关系：

lvalue 具有程序可访问的地址。 例如，lvalue 表达式包括变量名称，其中包括 `const` 变量、数组元素、返回 lvalue 引用的函数调用、位域、联合和类成员。

prvalue 表达式没有可供程序访问的地址。 例如，prvalue 表达式包括文本、可返回非引用类型的函数调用，以及在表达式计算期间创建的但只能由编译器访问的临时对象。

xvalue 表达式具有程序不再可访问的地址，但可用于初始化右值引用，该引用提供对表达式的访问权限。 例如，它包括可返回 rvalue 引用的函数调用，以及数组下标、成员和指向其中数组或对象是 rvalue 引用的成员表达式的指针。

示例

以下示例演示左值和右值的多种正确的和错误的用法：

C++

```
// 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
}
```

> **注意**
>
> 此主题中的示例阐释了未重载运算符时的正确和错误用法。 通过重载运算符，可以使表达式（如 `j * 4`）成为左值。

在提到对象引用时，通常会使用术语 lvalue 和 rvalue。 有关引用的详细信息，请参阅 lvalue 引用声明符：& 和 Rvalue 引用声明符：&&。

另请参阅

基本概念
Lvalue 引用声明符：&
引用声明符：&&

Last updated on 2025/11/07

临时对象

临时对象是由编译器创建的未命名对象，用于存储临时值。

备注

在某些情况下，编译器需要创建临时对象。 可能会出于下列原因创建这些临时对象：

- 使用一个类型不同于所初始化引用的基础类型的初始化表达式来初始化 `const` 引用。
- 存储返回用户定义类型 (UDT) 的函数的返回值。 仅当程序未将返回值复制到对象时，才会创建这些临时对象。 例如：

  ```
  C++
  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.
  ```

  由于未将返回值复制到另一个对象，因此创建了临时对象。 创建临时内存更常见的情况是在计算必须调用重载运算符函数的表达式时。 这些重载的运算符函数将返回用户定义类型，该类型通常不会复制到另一个对象。

  请考虑表达式 `ComplexResult = Complex1 + Complex2 + Complex3`。 将计算表达式 `Complex1 + Complex2`，并且结果将存储在临时对象中。 接下来，计算表达式 temporary ，并将结果复制到 `+ Complex3` （假设未重载赋值运算符） `ComplexResult`。

- 存储强制转换为用户定义的类型的结果。 在给定类型的对象显式转换为用户定义的类型时，将构造一个新对象作为临时对象。

临时对象具有根据其创建点和销毁点定义的生存期。 任何创建多个临时对象的表达式最终都会按与这些对象的创建顺序相反的顺序来销毁它们。

临时销毁发生的时间取决于它的使用方式：

- 用于初始化 `const` 引用的临时对象：
  如果初始化表达式不是与正在初始化的引用类型相同的左值，则会创建基础对象类型的临时值。 初始化表达式会对其进行初始化。 此临时对象将在其绑定到的引用对象销毁后立即销毁。 由于此销毁很可能在创建临时表达式之后发生，因此有时称为生存期扩展。

- 作为表达式计算结果创建的临时对象：
  所有不属于第一类的临时对象，以及作为表达式计算结果而创建的所有临时对象，都在表达式语句的末尾（即分号处）或用于 `for`、`if`、`while`、`do` 和 `switch` 语句的控制表达式的末尾销毁。

另请参阅

Herb Sutter 关于简单参考 的博客

Last updated on 2025/11/07

保持同步

C + + 的低级功能之一是能够指定内存中对象的精确对齐方式，以最大限度利用特定的硬件体系结构。 默认情况下，编译器会根据大小值对齐类和结构成员： `bool` 和 `char` 在 1 字节边界上对齐， `short` 在 2 字节边界上对齐， `int`、 `long` 和 `float` 在 4 字节边界上对齐， `long long`、 `double` 和 `long double` 在 8 字节边界上对齐。

在大多数情况下，你永远无需注意对齐方式，因为默认对齐方式已经是最佳的。 但是，在某些情况下，你可以通过指定数据结构的自定义对齐方式，获得显著的性能提升或节约内存。 在 Visual Studio 2015 之前，可以使用 Microsoft 专用关键字 `__alignof` 和 `__declspec(align)` 来指定大于默认值的对齐方式。 从 Visual Studio 2015 开始，应使用 C++11 标准关键字 `alignof` 和 `alignas` 以获得最强大的代码可移植性。 新关键字实质上以与 Microsoft 专用扩展相同的方式运行。 这些扩展的文档也适用于这些新关键字。 有关详细信息，请参阅 alignof 运算符、alignas 说明符和对齐。 C++ 标准不指定用于在小于目标平台编译器默认值的边界上对齐的装箱行为，因此在这种情况下，你仍需要使用 Microsoft #pragma pack。

使用 aligned_storage 类 为具有自定义对齐方式的数据结构分配内存。 aligned_union 类用于指定与非平凡构造函数或析构函数联合时的对齐方式。

对齐方式和内存地址

对齐方式是内存地址的一个属性，表示为数字地址对 2 的幂次方取模。 例如，地址 0x0001103F 对 4 取模为 3。 该地址对齐到 4n+3，其中 4 表示选择的 2 的幂次方。 地址的对齐方式取决于选择的 2 的幂次方。 相同的地址对 8 取模为 7。 如果一个地址的对齐方式是 Xn+0，它将对齐到 X。

CPU 执行作用于内存中所存储数据的指令。 数据在内存中用地址标识。 单个基准也具有大小。 如果一个基准的地址对齐到其大小，则称它为自然对齐。 否则，称为未对齐。 例如，如果地址用于标识其采用 8 字节对齐，则自然对齐 8 字节浮点基准。

数据对齐的编译器处理

编译器尝试以防止数据未对齐的方式分配数据。

对于简单的数据类型，编译器将分配是数据类型的大小（以字节为单位）的倍数的地址。 例如，编译器将地址分配给类型为 `long` 且是 4 的倍数的变量，并将地址的最底部 2 位设置为零。

编译器还以自然对齐结构的每一个元素的方式填充结构。 来看看下面代码示例中的结构 `struct x_`：

```
C++
```

```
struct x_
{
   char a;     // 1 byte
   int b;      // 4 bytes
   short c;    // 2 bytes
   char d;     // 1 byte
} bar[3];
```

编译器填充此结构以自然强制实施对齐方式。

下面的代码示例展示了编译器如何将填充的结构置于内存中：

C++

```
// 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];
```

两个声明都将 `sizeof(struct x_)` 作为 12 个字节返回。

第二个声明包括两个填充元素：

1. `char _pad0[3]`，对齐 4 字节边界上的 `int b` 成员。
2. `char _pad1[1]` 用于在 4 字节边界上对齐结构 `struct _x bar[3];` 的数组元素。

填充以允许自然访问的方式对齐 `bar[3]` 的元素。

下面的代码示例展示了 `bar[3]` 的数组布局：

Output

```
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` 和 `alignas`

`alignas` 说明符是一种可移植的 C++ 标准方法，用于指定变量和用户定义类型的自定义对齐方式。 `alignof` 运算符同样也是一种标准的、可移植的方法，可以获取指定类型或变量的对齐方式。

示例

你可以在类（结构或联合）或者单个成员上使用 `alignas`。 遇到多个 `alignas` 说明符时，编译器会选择其中值最大的一个。

C++

```cpp
// 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
}
```

另请参阅

数据结构对齐方式

Last updated on 2025/11/07

普通、标准布局、POD 和文本类型

术语“布局”是指类、结构或联合类型的对象的成员在内存中的排列方式。 在某些情况下，布局由语言规范明确定义。 但是，当类或结构包含某些 C++ 语言功能（如虚拟基类、虚拟函数、具有不同访问控制的成员）时，编译器可以自由选择布局。 该布局可能会基于正在执行的优化而有所不同，并且在许多情况下，该对象甚至可能不会占用连续内存区域。 例如，如果某个类具有虚拟函数，则该类的所有实例可能会共享单个虚拟函数表。 此类型非常有用，但它们也有限制。 由于布局未定义，因此无法将其传递到使用其他语言（例如 C）编写的程序，并且由于它们可能是非连续的，因此无法使用快速低级函数（例如 `memcopy`）对其进行可靠复制，或者通过网络对其进行序列化。

为使编译器以及 C++ 程序和元程序能够推断出任何给定类型对于依赖于特定内存布局的操作的适用性，C++14 引入了三种类别的简单类和结构：普通、标准布局和 POD（或简单旧数据）。 标准库具有函数模板 `is_trivial<T>`、`is_standard_layout<T>` 和 `is_pod<T>`，这些模板可以确定某一给定类型是否属于某一给定类别。

普通类型

当 C++ 中的类或结构具有编译器提供的或显式默认设置的特殊成员函数时，该类或结构为普通类型。 它占用连续内存区域。 它可以具有含不同访问说明符的成员。 在 C++ 中，编译器可以自由选择在此情况下对成员排序的方式。 因此，你可以在内存中复制此类对象，但不能从 C 程序中可靠地使用它们。 可以将普通类型 T 复制到 char 或无符号 char 数组，并安全地复制回 T 变量。 请注意，由于对齐要求，类型成员之间可能存在填充字节。

普通类型具有普通默认构造函数、普通复制构造函数、普通复制赋值运算符和普通析构函数。 在各种情况下，“普通”意味着构造函数/运算符/析构函数并非用户提供，并且属于存在以下情况的类

- 没有虚拟函数或虚拟基类，
- 没有具有相应非普通构造函数/运算符/析构函数的基类
- 没有具有相应非普通构造函数/运算符/析构函数的类类型的数据成员

以下示例演示普通类型。 在 Trivial2 中，`Trivial2(int a, int b)` 构造函数的存在要求提供默认构造函数。 对于符合普通资格的类型，必须显式默认设置该构造函数。

C++

```
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
};
```

标准布局类型

当类或结构不包含某些 C++ 语言功能（例如无法在 C 语言中找到的虚拟函数），并且所有成员都具有相同的访问控制时，该类或结构为标准布局类型。 可以在内存中对其进行复制，并且布局已经过充分定义，可以由 C 程序使用。 标准布局类型可以具有用户定义的特殊成员函数。 此外，标准布局类型还具有以下特征：

- 没有虚拟函数或虚拟基类
- 所有非静态数据成员都具有相同的访问控制
- 类类型的所有非静态成员均为标准布局
- 所有基类都为标准布局
- 没有与第一个非静态数据成员类型相同的基类。
- 满足以下条件之一：
  - 最底层派生类中没有非静态数据成员，并且具有非静态数据成员的基类不超过一个，或者
  - 没有含非静态数据成员的基类

以下代码演示标准布局类型的一个示例：

C++

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

可能使用代码能够更好地说明最后两个要求。 在下一个示例中，即使 Base 是标准布局，`Derived` 也不是标准布局，因为它（最底层派生类）和 `Base` 都具有非静态数据成员：

C++

```
struct Base
{
   int i;
   int j;
};

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

在此示例中，`Derived` 是标准布局，因为 `Base` 没有非静态数据成员：

C++

```
struct Base
{
   void Foo() {}
};

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

如果 `Base` 具有数据成员，并且 `Derived` 仅具有成员函数，则 Derived 也是标准布局。

POD 类型

当某一类或结构同时为普通和标准布局时，该类或结构为 POD（简单旧数据）类型。 因此，POD 类型的内存布局是连续的，并且每个成员的地址都比在其之前声明的成员要高，以便可以对这些类型执行逐字节复制和二进制 I/O。 标量类型（例如 int）也是 POD 类型。 作为类的 POD 类型只能具有作为非静态数据成员的 POD 类型。

示例

以下示例演示普通、标准布局和 POD 类型之间的区别：

C++

```cpp
#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;
}
```

文本类型

文本类型是可在编译时确定其布局的类型。 以下均为文本类型：

- void
- 标量类型
- 引用
- Void、标量类型或引用的数组
- 具有普通析构函数以及一个或多个 constexpr 构造函数且不移动或复制构造函数的类。 此外，其所有非静态数据成员和基类必须是文本类型且不可变。

另请参阅

基本概念

Last updated on 2025/11/07

作为值类型的 C++ 类

默认情况下，C++ 类是值类型。 可以将其指定为引用类型，使多态行为支持面向对象的编程。 值类型有时从内存和布局控件的角度进行查看，而引用类型与基类和虚拟函数有关，用于多态目的。 默认情况下，值类型可以复制，这意味着总是有一个复制构造函数和一个复制赋值运算符。 对于引用类型，将类设为不可复制（禁用复制构造函数和复制赋值运算符），并使用支持其预期多态性的虚拟析构函数。 值类型还与内容有关，复制时，总是提供可单独修改的两个独立值。 引用类型与标识有关 - 它是哪种类型的对象？ 因此，“引用类型”也称为“多态类型”。

如果确实需要类似引用的类型（基类、虚拟函数），则需要显式禁用复制，如以下代码中的 `MyRefType` 类所示。

C++

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

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

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

编译上述代码将导致以下错误：

Output

```
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'
```

值类型和移动效率

由于新的复制优化，避免了复制分配开销。 例如，在字符串向量中间插入字符串时，没有复制重新分配开销，只有一次移动，即使它会导致向量本身的增长。 这些优化也适用于其他操作：例如，对两个巨大的对象执行添加操作。 如何启用这些值操作优化？ 编译器可以隐式启用它们，就像编译器自动生成复制构造函数一样。 但是，类必须通过在类定义中声明移动赋值和移

动构造函数“选择加入”它们。 移动在适当的成员函数声明和定义移动构造函数和移动赋值方法中使用双与号 (&&) rvalue 引用。 还需要插入正确的代码，以从源对象中“窃取内容”。

如何确定是否需要启用移动操作？ 如果你已经知道需要启用复制构造，你可能也希望启用移动构造，特别是它比深层副本便宜的情况。 但是，如果知道需要移动支持，这不一定意味着要启用复制操作。 后一种情况称为“仅移动类型”。 标准库中已有的示例是 `unique_ptr`。 顺便说一下，旧的 `auto_ptr` 已被弃用，并被 `unique_ptr` 取代，这正是由于以前版本的 C++ 中缺乏移动语义支持。

通过使用移动语义，可以按值返回或在中间插入。 移动是复制的优化。 无需将堆分配作为解决方法。 请看下面的伪代码：

C++

```
#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
```

为适当的值类型启用移动

对于类似于值的类，移动比深度副本便宜，启用移动构造和移动赋值以提高效率。 请看下面的伪代码：

C++

```
#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!");
    }
};
```

如果启用复制构造/赋值，也会启用移动构造/赋值（如果它比深度副本便宜）。

某些非值类型是仅移动类型，例如，无法克隆资源时，仅传输所有权。 示例：`unique_ptr`。

另请参阅

C++ 类型系统
欢迎回到 C++
C++ 语言参考
C++ 标准库

Last updated on 2025/11/07

类型转换和类型安全

本文档确定了一些常见的类型转换问题，并介绍了如何在 C++ 代码中避免它们。

在编写 C++ 程序时，请务必确保它是类型安全的。 这意味着每个变量、函数自变量和函数返回值将存储一个可接受的类型的数据，并意味着涉及不同类型的值的操作“有意义”且不会导致数据丢失、位模式解释不正确或内存损坏。 从不显式或隐式地将值从一种类型转换为另一种类型的程序在定义上是类型安全的。 但是，有时需要进行类型转换，甚至需要进行不安全的类型转换。 例如，可能必须在 `int` 类型的变量中存储浮点运算的结果，或者可能必须将 `unsigned int` 中的值传递到采用 `signed int` 的函数。 这两个示例列举了不安全的转换，因为它们可能会导致数据丢失或值的重新解释。

当编译器检测到不安全的转换时，它会发出错误或警告。 如果发出错误，则编译会停止；如果发出警告，则编译可以继续，但会指示代码中可能存在错误。 但是，即使您编译程序后没有收到警告，它仍然可能包含导致生成错误结果的隐式类型转换。 类型错误也可能由代码中的显式转换或强制转换引入。

隐式类型转换

当表达式包含不同内置类型的操作数且不存在显式强制转换时，编译器将使用内置的“标准转换”来转换其中一个操作数，从而使类型相匹配。 编译器将尝试按一个明确定义的顺序进行转换，直到有一个转换成功。 如果所选转换是提升转换，则编译器不会发出警告。 如果转换是收缩转换，则编译器会发出有关数据可能丢失的警告。 尽管是否真的发生数据丢失取决于涉及的实际值，但我们建议您将此警告视为错误。 如果涉及到用户定义的类型，则编译器将尝试使用您在类定义中指定的转换。 如果编译器找不到可接受的转换，则会发出错误且不会编译程序。 有关治理标准转换的规则的详细信息，请参阅标准转换。 有关用户定义转换的详细信息，请参阅用户定义转换 (C++/CLI)。

扩大转换（提升）

在扩大转换中，较小的变量中的值将赋给较大的变量，同时不会丢失数据。 由于扩大转换始终是安全的，编译器将在不提示的情况下执行它们且不会发出警告。 以下转换是扩大转换。

展开表

来自	功能
除 `signed` 或 `unsigned` 以外的任何 `long long` 或 `__int64` 整型类型	`double`
`bool` 或 `char`	任何其他内置类型
`short` 或 `wchar_t`	.- .

来自	功能
%>	`long long`
`float`	`double`

收缩转换（强制）

编译器隐式执行收缩转换，但会发出有关数据丢失可能的警告。 请重视这些警告。 如果您确定数据丢失不会发生（因为较大的变量中的值始终适合较小的变量），则添加显式强制转换，使编译器不再发出警告。 如果不确定转换是否安全，请为代码添加某种运行时检查以处理可能出现的数据丢失，从而确保转换不会导致程序生成错误的结果。

任何从浮点类型到整型的转换都是收缩转换，因为浮点值的小数部分将会丢弃和丢失。

以下代码示例演示了一些隐式收缩转换以及编译器为其发出的警告。

C++

```
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
```

有符号到无符号的转换

有符号整型及其对应的无符号整型的大小总是相同，但它们的区别在于为值转换解释位模式的方式。 以下代码示例演示将相同的位模式解释为有符号值和无符号值时发生的情况。 存储在 `num` 和 `num2` 中的位模式将与前面的演示中所示的位模式保持相同，绝不会发生更改。

C++

```
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"
```

请注意，值将在两个方向重新解释。 如果程序生成了奇怪的结果，其中的值的符号似乎与您预期的相反，请查找有符号和无符号整型之间的隐式转换。 在以下示例中，当表达式的结果 (0 - 1) 存储在 `int` 中时，该结果将从 `unsigned int` 隐式转换为 `num`。 这将导致位模式被重新解释。

C++

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

编译器不会发出有关有符号和无符号整型之间的隐式转换的警告。 因此，建议完全避免有从符号到无符号的转换。 如果无法避免它们，则添加运行时检查，以检测所转换的值是大于或等于零还是小于或等于有符号类型的最大值。 此范围内的值将从有符号转换为无符号或从无符号转换为有符号而不用重新解释。

指针转换

在很多表达式中，C 样式数组将隐式转换为指向该数组中的第一个元素的指针，并且可能在没有提示的情况下进行常量转换。 尽管此方法很方便，但它也可能容易出错。 例如，以下设计得不合理的代码示例似乎没有意义，但它会编译并生成结果“p”。 首先，“Help”字符串常量将转换为指向数组的第一个元素的 `char*`；该指针随后递增 3 个元素，以便现在能够指向最后一个元素“p”。

C++

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

显式转换（强制转换）

利用强制转换运算，您可以指示编译器将一种类型的值转换为另一种类型。 在某些情况下，如果两个类型完全无关，编译器将引发错误，但在其他情况下，即使运算不是类型安全的，编译器也不会引发错误。 应谨慎使用强制转换，因为从一种类型到另一个类型的任何转换都可能导致程序错误。 但是，强制转换有时是必需的，并且不是所有强制转换都同样危险。 强制转换的一个有效的使用情况是：当你的代码执行收缩转换并且你知道该转换不会导致程序生成错误的结果时。 实际上，这是在告诉编译器您知道自己在做什么，不要发出相关警告来打扰您。 另一种使用情况是从指向派生类的指针到指向基类的指针的强制转换。 还有一种使用情况是强制转换掉变量的常量性以将其传递给需要非常量参数的函数。 大多数强制转换运算都存在一定的风险。

在 C 样式程序中，同一 C 样式强制转换运算符可用于所有类型的强制转换。

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

C 样式强制转换运算符与调用运算符 () 相同，因此在代码中不显眼，容易被忽略。 这两个运算符都不好，因为它们难以让人一眼认出，并且它们差异很大，无法调用 `static`、`const` 和 `reinterpret_cast` 的任意组合。 确定旧式强制转换的实际作用可能很困难并且容易出错。 基于所有这些原因，当需要进行强制转换时，我们建议使用以下 C++ 强制转换运算符之一。在某些情况下，它们更加类型安全，并且可以更加明确地表达编程目的：

- `static_cast`，用于仅在编译时检查的强制转换。 如果编译器检测到你尝试在完全不兼容的类型之间强制转换，`static_cast` 将返回错误。 您还可以使用它在指向基对象的指针和指向派生对象的指针之间强制转换，但编译器无法总是判断出此类转换在运行时是否安全。

  ```C++
  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);
  ```

  有关详细信息，请参阅 static_cast。

- `dynamic_cast`，用于从指向基对象的指针到指向派生对象的指针的、安全且经过运行时检查的强制转换。 `dynamic_cast` 在向下转换方面比 `static_cast` 更安全，但运行时检查会产生一些开销。

  ```C++
  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;
```

有关详细信息，请参阅 dynamic_cast。

- `const_cast`，用于转换掉变量的 `const` 性，或者将非 `const` 变量转换为 `const`。 使用此运算符转换掉 `const` 性与使用 C 样式强制转换一样容易出错，只不过使用 `const_cast` 时不太可能意外地执行强制转换。 有时候，必须转换掉变量的 `const` 性。例如，将 `const` 变量传递给采用非 `const` 参数的函数。 以下示例演示如何执行此操作。

C++

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

有关详细信息，请参阅 const_cast。

- `reinterpret_cast`，用于无关类型（如指针类型和 `int`）之间的强制转换。

> ⓘ **注意**
>
> 此强制转换运算符不像其他运算符一样常用，并且不能保证可将其移植到其它编译器。

以下示例演示 `reinterpret_cast` 与 `static_cast` 的差异。

C++

```
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.
```

有关详细信息，请参阅 reinterpret_cast 运算符。

另请参阅

C++ 类型系统
欢迎回到 C++
C++ 语言参考
C++ 标准库

Last updated on 2025/11/07

标准转换

C++ 语言定义其基础类型之间的转换。 它还定义指针、引用和指向成员的指针派生类型的转换。 这些转换称为“标准转换”。

本节讨论下列标准转换：

- 整型提升
- 整型转换
- 浮点转换
- 浮点转换和整型转换
- 算术转换
- 指针转换
- 引用转换
- 指向成员的指针转换

> **注意**
>
> 用户定义的类型可指定其自己的转换。 **构造函数**和**转换**中介绍了用户定义类型的转换。

以下代码将导致转换（本例中为整型提升）：

C++

```
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;
```

仅当转换生成引用类型时，其结果才为左值。 例如，声明为 `operator int&()` 的用户定义转换返回一个引用且为 l-value。 但是，声明为 `operator int()` 的转换将返回一个对象，而不是 l-value。

整型提升

整型类型的对象可以转换为另一个更宽的整型类型，即，可表示更大的一组值的类型。 这种扩展类型的转换称为“整型提升”。 利用整型提升，你可以在可使用其他整型类型的任何位置将以下类型用于表达式：

- `char` 和 `short int` 类型的对象、文本和常量
- 枚举类型
- `int` 位域
- 枚举器

C++ 提升是“值保留”，即提升后的值一定与提升前的值相同。 在值保留提升中，如果 `char` 可以表示原始类型的完整范围，较短的整型类型的对象（如 `int` 类型的位域或对象）将提升到 `int` 类型。 如果 `int` 无法表示完整范围的值，该对象将提升到 `unsigned int` 类型。 尽管此策略与标准 C 中使用的策略相同，但值保留转换不保留对象的“符号”。

值保留提升和保留符号的提升通常会生成相同的结果。 但是，如果提升的对象显示如下，它们可能生成不同的结果：

- `/`、`%`、`/=`、`%=`、`<`、`<=`、`>` 或 `>=` 的操作数

  这些运算符依赖于用于确定结果的符号。 当值保留和符号保留提升应用于这些操作数时，它们将生成不同的结果。

- `>>` 或 `>>=` 的左操作数

  这些运算符在移位运算中会区别对待有符号的数量和无符号的数量。 对于有符号的数量，右移位运算会将符号位传播到空位位置，而空位位置则以无符号数量填充零。

- 重载函数的参数，或重载运算符的操作数（取决于该操作数类型用于参数匹配的符号）。 有关定义重载运算符的详细信息，请参阅重载运算符。

整型转换

整型转换是整型类型之间的转换。 整型类型为 `char`、`short` （或 `short int`）、`int`、`long` 和 `long long`。 这些类型可使用 `signed` 或 `unsigned` 进行限定， `unsigned` 可以用作 `unsigned int` 的简写。

有符号转换为无符号

有符号整数类型的对象可以转换为对应的无符号类型。 当发生这些转换时，实际位模式不会改变。 但是，对数据的解释会更改。 考虑此代码：

C++

```
#include <iostream>

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

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

在前面的示例中，定义了 `signed short` 类型的 `i`，并将其初始化为负数。 表达式 `(u = i)` 导致 `i` 在为 `unsigned short` 赋值前转换为 `u`。

无符号转换为有符号

无符号整数类型的对象可以转换为对应的有符号类型。 但是，如果无符号值超出有符号类型的可表示范围，则结果将没有正确的值，如以下示例所示：

C++

```
#include <iostream>

using namespace std;
int main()
{
short  i;
unsigned short u = 65533;

cout << (i = u) << "\n";
}
//Output: -3
```

在前面的示例中， `u` 是一个 `unsigned short` 整型对象，必须将其转换为有符号的数量来计算表达式 `(i = u)`。 由于其值无法在 `signed short` 中正确表示，因此数据被错误解释。

浮点转换

浮动类型的对象可以安全地转换为更精确的浮点类型，也就是说，转换不会导致基数丢失。 例如，从 `float` 到 `double` 或从 `double` 到 `long double` 的转换是安全的，并且值保持不变。

如果浮点类型的对象位于低精度类型可表示的范围中，则还可转换为该类型。 （请参阅浮点限制，了解浮点类型的范围。）如果原始值无法精确表示，则可以将其转换为下一个更高或更低的可表示值。 如果此类值不存在，则结果不确定。 请考虑以下示例：

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

类型 `float` 可表示的最大值为 3.402823466E38，这比 1E300 小很多。 因此，该数字将转换为无穷大，结果为“inf”。

整型和浮点型之间的转换

某些表达式可能导致浮点型的对象转换为整型，反之亦然。 当整型类型的对象转换为浮点类型且无法精确表示原始值时，结果要么是下一个较大的可表示值，要么是下一个较小的可表示值。

当浮点类型的对象转换为整型类型时，小数部分将被截断，或四舍五入至零。 比如数字 1.3 将转换为 1，-1.3 将转换为 -1。 如果截断的值高于最高可表示值或低于最低可表示值，则结果不确定。

算术转换

很多二元运算符（在带二元运算符的表达式中有讨论）会导致操作数转换，并以相同的方式产生结果。 这些运算符导致的转换称为“常用算术转换”。 具有不同本机类型的操作数的算术转换按下表所示的方式完成。 Typedef 类型的行为方式基于其基础本机类型。

类型转换的条件

展开表

满足的条件	转换
其中一个操作数是 `long double` 类型。	另一个操作数将转换为 `long double` 类型。
未满足上述条件，并且其中一个操作数是 `double` 类型。	另一个操作数将转换为 `double` 类型。
未满足上述条件，并且其中一个操作数是 `float` 类型。	另一个操作数将转换为 `float` 类型。
未满足上述条件（没有任何一个操作数属于浮动类型）。	操作数获得整型提升，如下所示： - 如果其中一个操作数的类型为 `unsigned long`，则另一个操作数被转换

满足的条件	转换
	为 `unsigned long` 类型。 - 如果未满足上述条件，并且其中一个操作数是 `long` 类型，另一个操作数是 `unsigned int` 类型，则这两个操作数都将转换为 `unsigned long` 类型。 - 如果未满足上述两个条件，并且其中一个操作数是 `long` 类型，则另一个操作数将转换为 `long` 类型。 - 如果未满足上述三个条件，并且其中一个操作数是 `unsigned int` 类型，则另一个操作数将转换为 `unsigned int` 类型。 - 如果上述条件均未满足，则两个操作数都将转换为 `int` 类型。

以下代码演示了上表中所述的转换规则：

C++

```
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;
}
```

上面的示例中的第一个语句显示了两个整数类型 `iVal` 和 `ulVal` 的相乘。 满足的条件是两个操作数都不是浮点类型，并且一个操作数是 `unsigned int` 类型。 因此，另一个操作数 `iVal` 将转换为 `unsigned int` 类型。 然后，结果将分配给 `dVal`。 这里满足的条件是，一个操作数是 `double` 类型；因此，乘法的 `unsigned int` 结果将转换为 `double` 类型。

前面示例中的第二个语句显示了 `float` 类型和整型类型的加法： `fVal` 和 `ulVal`。 `ulVal` 变量将转换为 `float` 类型（表中的第三个条件）。 加法的结果将转换为 `double` 类型（表中的第二个条件）并分配给 `dVal`。

指针转换

在赋值、初始化、比较和其他表达式中，可以转换指针。

指向类的指针

在两种情况下，指向类的指针可转换为指向基类的指针。

第一种情况是指定的基类可访问且转换是明确的。 有关不明确的基类引用的详细信息，请参阅多个基类。

基类是否可访问取决于派生中使用的继承的类型。 考虑下图中阐释的继承：

说明基类辅助功能的继承图

下表显示针对该图阐释的情况的基类可访问性。

展开表

函数的类型	衍生	从 `B*` 至 `A*` 合法?
外部（非类范围）函数	专用	否
	受保护	否
	公用	是
B 成员函数（在 B 范围内）	专用	是
	受保护	是
	公用	是
C 成员函数（在 C 范围内）	专用	否
	受保护	是
	公用	是

第二种情况是，在您使用显式类型转换时，指向类的指针可转换为指向基类的指针。 有关显性类型转换的详细信息，请参阅显式类型转换运算符。

此类转换的结果是指向完全由基类描述的对象部分（即“子对象”）的指针。

以下代码定义了两个类（即 `A` 和 `B`），其中 `B` 派生自 `A`。 有关继承的详细信息，请参阅 派生类。 然后，它定义 `bObject` 一个类型 `B` 的对象，以及指向该对象的两个指针（`pA` 和 `pB`）。

C++

```
// 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
}
```

指针 `pA` 的类型为 `A *`，可解释为“指向 `A` 类型的对象的指针”。 `bObject` 的成员（如 `BComponent` 和 `BMemberFunc`）对于类型 `B` 而言是唯一的，因此无法通过 `pA` 访问。 `pA` 指针只允许访问类 `A` 中定义的对象的那些特性（成员函数和数据）。

指向函数的指针

如果类型 `void *` 足以保留指向函数的指针，则该指针可以转换为 `void *` 类型。

指向 void 的指针

指向 **`void`** 类型的指针可以转换为指向其他任何类型的指针，但仅适合于显式类型转换（与在 C 中的情况不同）。 指向任何类型的指针可以隐式转换为指向 **`void`** 类型的指针。 指向类型的不完整对象的指针可以转换为指向 **`void`**（隐式）和 back（显式）的指针。 此类转换的结果与原始指针的值相等。 对象被视为是不完整的（如果已声明对象），但未提供足够多的可用信息，无法确定其大小或基类。

指向不是 **`const`** 或 **`volatile`** 的任何对象的指针可以隐式转换为 `void *` 类型的指针。

固定和可变指针

C++ 不会应用从 `const` 或 `volatile` 类型到不是 `const` 或 `volatile` 类型的标准转换。 但是，任何类型的转换都可以用显式类型强制转换指定（包括不安全的转换）。

> ⓘ **注意**
>
> 指向成员的 C++ 指针（指向静态成员的指针除外）与常规指针不同，二者具有不同的标准转换。 指向静态成员的指针是普通指针，且与普通指针具有相同的转换。

null 指针转换

计算结果为零的整型常量表达式，或到某个指针类型的此类表达式强制转换，将转换为称为“空指针”的指针。 此指针与指向任何有效对象或函数的指针相比总是不相等的。 一个例外是指向基于对象的指针，它们可以具有相同的偏移量，但仍然指向不同的对象。

在 C++11 中，nullptr 类型应优先于 C 样式空指针。

指针表达式转换

带数组类型的所有表达式都可以转换为同一类型的指针。 转换的结果是指向第一个数组元素的指针。 下面的示例演示了这样的转换：

C++

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

生成返回特定类型的函数的表达式将转换为指向返回该类型的函数的指针，以下情况除外：

- 表达式用作 address-of 运算符 (&) 的操作数。
- 表达式用作到 function-call 运算符的操作数。

引用转换

对类的引用可在以下情况下转换为对基类的引用：

- 指定的基类是可访问的。
- 转换是明确的。 有关不明确的基类引用的详细信息，请参阅多个基类。

转换的结果为指向表示基类的子对象的指针。

指向成员的指针

指向可在赋值、初始化、比较和其他语句中转换的类成员的指针。 本节描述以下指向成员的指针转换：

指向基类成员的指针

当满足以下条件时，指向基类的成员的指针可以转换为指向派生自基类的类的成员的指针：

- 从指向派生类的指针到基类指针的反向转换可以访问。
- 派生类并非以虚拟方式从基类继承。

当左操作数是指向成员的指针时，右操作数必须是 pointer-to-member 类型或计算结果为 0 的常量表达式。 此赋值仅在以下情况下有效：

- 右操作数是指向与左操作数相同的类的成员的指针。
- 左操作数是指向以公共但不明确的方式派生自右操作数的类的成员的指针。

指向成员的空指针转换

计算结果为零的整型常量表达式将转换为空指针。 此指针与指向任何有效对象或函数的指针相比总是不相等的。 一个例外是指向基于对象的指针，它们可以具有相同的偏移量，但仍然指向不同的对象。

以下代码演示了指向类 `i` 中的成员 `A` 的指针的定义。 指针 `pai` 将初始化为 0，因此是 null 指针。

C++

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

int A::*pai = 0;

int main()
{
}
```

另请参阅

C++ 语言参考

Last updated on 2025/11/07

内置类型 (C++)

内置类型（也称基本类型）由 C++ 语言标准指定，内置于编译器中。 内置类型未在任何头文件中定义。 内置类型分为三个主要类别：整型、浮点和 void。 整型类型表示整数。 浮点类型可以指定可能有小数部分的值。 编译器将大多数内置类型视为非重复类型。 但是，某些类型是同义词，或被编译器视为等效类型。

Void 类型

void 类型描述值的空集。 无法指定类型为 `void` 的变量。 `void` 类型主要用于声明不返回值的函数，或用于声明指向非类型化或任意类型化数据的一般指针。 任何表达式都可以显示或强制转换为类型 `void`。 但是，此类表达式仅限于下列用途：

- 表达式语句。 有关详细信息，请参阅 表达式。
- 逗号运算符的左操作数。 有关详细信息，请参阅 逗号运算符。
- 条件运算符 (`? :`) 的第二个或第三个操作数。 有关详细信息，请参阅 条件运算符的表达式。

std::nullptr_t

关键字 `nullptr` 是类型为 `std::nullptr_t` 的 null 指针常量，该类型可转换为任何原始指针类型。 有关详细信息，请参阅 nullptr。

布尔类型

bool 类型的值可以是 true 和 false。 `bool` 类型的大小特定于实现。 有关特定于 Microsoft 的实现的详细信息，请参阅内置类型的大小。

字符类型

`char` 类型是一种字符表示类型，可有效地对基本执行字符集的成员进行编码。 C++ 编译器将 `char`，`signed char` 和 `unsigned char` 类型的变量视为不同类型。

特定于 Microsoft： `char` 类型的变量将提升到 `int`，就像在默认情况下从 `signed char` 类型提升一样，除非使用 /J 编译选项。 在这种情况下，它们被视为 `unsigned char` 类型并提升为 `int`（没有符号扩展）。

`wchar_t` 类型的变量是宽字符或多字节字符类型。 在字符或字符串文本前使用 `L` 前缀可指定宽字符类型。

特定于 Microsoft：默认情况下，`wchar_t` 是原生类型，但你可以使用 /Zc:wchar_t- 使 `wchar_t` 成为 `unsigned short` 的 typedef。 `__wchar_t` 类型是本机 `wchar_t` 类型的 Microsoft 专用同义词。

`char8_t` 类型用于 UTF-8 字符表示形式。 它具有与 `unsigned char` 相同的表示形式，但被编译器视为非重复类型。 `char8_t` 类型是 C++20 中的新增类型。 特定于 Microsoft：使用 `char8_t` 需要 /std:c++20 编译器选项或更高版本（例如 `/std:c++latest`）。

`char16_t` 类型用于 UTF-16 字符表示形式。 它必须足够大才能表示任何 UTF-16 代码单元。 它被编译器视为非重复类型。

`char32_t` 类型用于 UTF-32 字符表示形式。 它必须足够大才能表示任何 UTF-32 代码单元。 它被编译器视为非重复类型。

浮点类型

浮点类型使用 IEEE-754 表示形式在各种数量级上提供小数值的近似值。 下表列出了 C++ 中的浮点类型以及浮点类型大小的相对限制。 这些限制由 C++ Standard 强制施加，独立于 Microsoft 实现。 C++ Standard 中未指定内置浮点类型的绝对大小。

展开表

类型	目录
`float`	在 C++ 中，`float` 类型是最小的浮点类型。
`double`	`double` 类型是大于或等于 `float`类型的大小但小于或等于 `long double`类型的大小的浮点类型。
`long double`	`long double` 类型是大于或等于 `double`类型的浮点类型。

特定于 Microsoft： `long double` 和 `double` 的表示形式完全相同。 但是，编译器将 `long double` 和 `double` 视为非重复类型。 Microsoft C++ 编译器使用 4 字节和 8 字节 IEEE-754 浮点表示形式。 有关详细信息，请参阅 IEEE 浮点表示形式。

整数类型

`int` 类型是默认的基本整数类型。 它可以表示某个特定于实现的范围的所有整数。

带符号整数表示形式可以同时保存正值和负值。 它默认使用，或者在存在 `signed` 修饰符关键字时使用。 `unsigned` 修饰符关键字指定一个只能保存非负值的无符号表示形式。

大小修饰符指定使用的整数表示形式的宽度（以位为单位）。 语言支持 `short`、`long` 和 `long long` 修饰符。 `short` 类型必须至少为 16 位宽。 `long` 类型必须至少为 32 位宽。 `long long` 类型必须至少为 64 位宽。 C++ Standard 指定整型类型之间的大小关系：

```
1 == sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long) <= sizeof(long long)
```

一个实现必须同时维护每种类型的最小大小要求和大小关系。 但是，实际大小在不同实现之间可能且确实有所不同。 有关特定于 Microsoft 的实现的详细信息，请参阅内置类型的大小。

指定 `int`、`signed` 或大小修饰符时，可以省略 `unsigned` 关键字。 修饰符和 `int` 类型（如果存在）可能按任何顺序显示。 例如，`short unsigned` 和 `unsigned int short` 指同一类型。

整数类型同义词

编译器将以下类型组视为同义词：

- `short`、`short int`、`signed short`、`signed short int`
- %>
- .- .
- %>
- `long`、`long int`、`signed long`、`signed long int`
- %>
- `long long`、`long long int`、`signed long long`、`signed long long int`
- %>

特定于 Microsoft 的整数类型包括特定宽度的 `__int8`、`__int16`、`__int32` 和 `__int64` 类型。 这些类型可以使用 `signed` 和 `unsigned` 修饰符。 `__int8` 数据类型与 `char` 类型同义，`__int16` 与 `short` 类型同义，`__int32` 与 `int` 类型同义，`__int64` 与 `long long` 类型同义。

内置类型的大小

大多数内置类型都有由实现定义的大小。 下表列出了 Microsoft C++ 中的内置类型所需的存储量。 具体而言，即使在 64 位操作系统上，`long` 也是 4 个字节。

展开表

类型	大小
`bool`、`char`、`char8_t`、`unsigned char`、`signed char`、`__int8`	1 个字节
`char16_t`、`__int16`、`short`、`unsigned short`、`wchar_t`、`__wchar_t`	2 个字节
`char32_t`、`float`、`__int32`、`int`、`unsigned int`、`long`、`unsigned long`	4 个字节
`double`、`__int64`、`long double`、`long long`、`unsigned long long`	8 字节

有关每个类型的值的范围的摘要，请参阅数据类型范围。

有关类型转换的详细信息，请参阅标准转换。

另请参阅

数据类型范围

Last updated on 2025/11/07

数据类型范围

Microsoft C++ 32 位和 64 位编译器可识别本文后面的表中的类型。

```
C++
- 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)
```

如果其名称以两条下划线 (`__`) 开始，则数据类型是非标准的。

下表中指定的范围均包含起始值和结束值。

展开表

类型名称	字节	其他名称	值的范围
`int`	4	`signed`	-2,147,483,648 到 2,147,483,647
`unsigned int`	4	`unsigned`	0 到 4,294,967,295
`__int8`	1	`char`	-128 到 127
`unsigned __int8`	1	`unsigned char`	0 到 255
`__int16`	2	.- .	-32,768 到 32,767
`unsigned __int16`	2	%>	0 到 65,535
`__int32`	4	.- .	-2,147,483,648 到 2,147,483,647
`unsigned __int32`	4	%>	0 到 4,294,967,295
`__int64`	8	%>	-9,223,372,036,854,775,808 到 9,223,372,036,854,775,807
`unsigned __int64`	8	`unsigned long long`	0 到 18,446,744,073,709,551,615
`bool`	1	无	`false` 或 `true`

类型名称	字节	其他名称	值的范围
`char`	1	无	-128 到 127（默认） 0 到 255（在通过使用 /J 进行编译时）
`signed char`	1	无	-128 到 127
`unsigned char`	1	无	0 到 255
`short`	2	%>	-32,768 到 32,767
`unsigned short`	2	`unsigned short int`	0 到 65,535
`long`	4	%>	-2,147,483,648 到 2,147,483,647
`unsigned long`	4	`unsigned long int`	0 到 4,294,967,295
`long long`	8	无（但与 `__int64` 等效）	-9,223,372,036,854,775,808 到 9,223,372,036,854,775,807
`unsigned long long`	8	无（但与 `unsigned __int64` 等效）	0 到 18,446,744,073,709,551,615
`enum`	多种多样	无	
`float`	4	无	3.4E +/- 38（七位数字）
`double`	8	无	1.7E +/- 308（15 位数字）
`long double`	与 `double` 相同	无	与 `double` 相同
`wchar_t`	2	`__wchar_t`	0 到 65,535

`__wchar_t` 的变量指定宽字符类型或多字节字符类型。 在字符或字符串常量前使用 `L` 前缀以指定宽字符类型常量。

`signed` 和 `unsigned` 是可用于任何整型（ `bool` 除外）的修饰符。 请注意，对于重载和模板等机制而言， `char`、 `signed char` 和 `unsigned char` 是三种不同的类型。

`int` 和 `unsigned int` 类型的大小为 4 个字节。 但是，由于语言标准允许可移植代码特定于实现，因此该代码不应依赖于 `int` 的大小。

Visual Studio 中的 C/C++ 还支持按大小分类的整型。 有关详细信息，请参阅 __int8, __int16, __int32, __int64 和整数限制。

有关每个类型的大小限制的详细信息，请参阅内置类型。

枚举类型的范围因语言上下文和指定的编译器标志而异。 有关详细信息，请参阅 C 枚举声明 和 枚举。

另请参阅

关键字
内置类型

Last updated on 2025/11/07

nullptr

关键字 **`nullptr`** 指定类型 `std::nullptr_t` 的 null 指针常量，该类型可转换为任何原始指针类型。 尽管可以使用关键字 **`nullptr`** 而不包含任何标头，但如果代码使用类型 `std::nullptr_t`，则必须通过包含标头 `<cstddef>` 来定义该类型。

> **注意**
>
> 用于托管代码应用程序的 C++/CLI 中也定义了 **`nullptr`** 关键字，并且它与 ISO 标准 C++ 关键字不可互换。 如果可以使用 /clr 编译器选项（以托管代码为目标）编译代码，则在必须保证编译器使用本机 C++ 解释的任何代码行中使用 `__nullptr`。 有关详细信息，请参阅 nullptr（C++/CLI 和 C++/CX）。

备注

请避免将 `NULL` 或零 (`0`) 用作 null 指针常量； **`nullptr`** 不仅不易被误用，并且在大多数情况下效果更好。 例如，给定 `func(std::pair<const char *, double>)`，那么调用 `func(std::make_pair(NULL, 3.14))` 会导致编译器错误。 宏 `NULL` 将扩展到 `0`，以便调用 `std::make_pair(0, 3.14)` 将返回 `std::pair<int, double>`，此结果不可转换为 `std::pair<const char *, double>` 的 `func` 参数类型。 调用 `func(std::make_pair(nullptr, 3.14))` 将会成功编译，因为 `std::make_pair(nullptr, 3.14)` 返回 `std::pair<std::nullptr_t, double>`，此结果可转换为 `std::pair<const char *, double>`。

另请参阅

关键字
nullptr（C++/CLI 和 C++/CX）

Last updated on 2025/11/07

void （C++）

用作函数返回类型时， `void` 关键字指定函数不返回值。 当用于函数的参数列表时， `void` 将指定函数不采用任何参数。 用于指针声明时， `void` 指定该指针为“通用”。

如果指针类型为 `void*`，则该指针可以指向任何未使用 `const` 或 `volatile` 关键字声明的变量。 `void*` 指针不能取消引用，除非它被强制转换为另一种类型。 `void*` 指针可以转换为任何其他类型的数据指针。

在 C++ 中， `void` 指针可以指向 free 函数（不是类成员的函数）或静态成员函数，但不能指向非静态成员函数。

无法声明 `void` 类型变量。

就样式而言，C++ Core Guidelines 建议不要将 `void` 用于指定空的形参列表。 有关详细信息，请参阅 C++ Core Guidelines NL.25：不要将 void 用作参数类型 。

示例

C++

```
// void.cpp

void return_nothing()
{
   // A void function can have a return with no argument,
   // or no return statement.
}

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

另请参阅

关键字
内置类型

Last updated on 2025/11/07

bool （C++）

此关键字是内置类型。 此类型的变量可以具有值 true 和 false。 条件表达式不仅具有类型 `bool`，还具有类型 `bool` 的值。 例如，`i != 0` 现在具有 `true` 或 `false`，具体取决于 `i` 的值。

Visual Studio 2017 版本 15.3 及更高版本（在 **/std:c++17** 和更高版本中可用）： 后缀或前缀递增或递减运算符的操作数可能不是类型 `bool`。 换句话说，如果是 `b` 类型的变量 `bool`，则不允许使用这些表达式：

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

值 `true`和 `false` 具有以下关系：

```
C++
!false == true
!true == false
```

在下面的语句中：

```
C++
if (condexpr1) statement1;
```

如果 `condexpr1` 为 `true`，则始终执行 `statement1`；如果 `condexpr1` 为 `false`，则从不执行 `statement1`。

当后缀或前缀 `++` 运算符应用于类型 `bool` 的变量时，该变量将设置为 `true`。

Visual Studio 2017 版本 15.3 及更高版本：已从语言中删除 `operator++` 的 `bool` 且不再提供支持。

后缀或前缀 `--` 运算符不能应用于此类型的变量。

`bool` 类型参与了默认整型提升。 类型 `bool` 的右值可以转换为类型 `int` 的右值，同时 `false` 会变为 0，且 `true` 会变为 1。 作为截然不同的类型，`bool` 参与重载决策。

另请参阅

关键字
内置类型

Last updated on 2025/11/07

false （C++）

关键字是类型为 bool 的变量或条件表达式（条件表达式现为 `true` 布尔表达式）的两个值之一。 例如，如果 `i` 是 `bool` 类型的变量，则 `i = false;` 语句会将 `false` 分配给 `i`。

示例

C++

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

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

Output

```
1
0
```

另请参阅

关键字

Last updated on 2025/11/07

true （C++）

语法

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

备注

此关键字是类型 bool 的变量或条件表达式（条件表达式现在是值为 true 的 Boolean 表达式）的两个值之一。 如果 `i` 的类型为 **`bool`**，则 `i = true;` 语句会将 **`true`** 赋给 `i`。

示例

C++

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

Output

```
1
0
```

另请参阅

关键字

Last updated on 2025/11/07

char、wchar_t、char8_t、char16_t、char32_t

类型 `char`、`wchar_t`、`char8_t`、`char16_t` 和 `char32_t` 是内置类型，可表示字母数字字符，非字母数字字形和非打印字符。

语法

C++

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

备注

`char` 类型是 C 和 C++ 中的原始字符类型。 `char` 类型可存储 ASCII 字符集或任何 ISO-8859 字符集中的字符，以及多字节字符的单个字节，例如 Shift-JIS 或 Unicode 字符集的 UTF-8 编码。 在 Microsoft 编译器中，`char` 是 8 位类型。 它是与 `signed char` 和 `unsigned char` 都不同的类型。 默认情况下，`char` 类型的变量将提升到 `int`，就像是从 `signed char` 类型一样，除非使用 /J 编译器选项。 在 `/J` 的情况下，它们被视为 `unsigned char` 类型并提升为 `int` （没有符号扩展）。

类型 `unsigned char` 通常用于表示 byte，它不是 C++ 中的内置类型。

`wchar_t` 类型是实现定义的宽字符类型。 在 Microsoft 编译器中，它表示一个 16 位宽字符，用于存储编码为 UTF-16LE 的 Unicode（Windows 操作系统上的本机字符类型）。 通用 C 运行时 (UCRT) 库函数的宽字符版本使用 `wchar_t` 及其指针和数组类型作为参数和返回值，本机 Windows API 的宽字符版本也是如此。

`char8_t`、`char16_t` 和 `char32_t` 类型分别表示 8 位、16 位和 32 位宽字符。 （`char8_t` 是 C++20 中的新增功能，需要 /std:c++20 或 `/std:c++latest` 编译器选项。）编码为 UTF-8 的 Unicode 可以存储在 `char8_t` 类型中。 `char8_t` 和 `char` 类型的字符串称为“窄”字符串，即使用于编码 Unicode 或多字节字符。 编码为 UTF-16 的 Unicode 可以存储在 `char16_t` 类型中，而编码为 UTF-32 的 Unicode 可以存储在 `char32_t` 类型中。 这些类型和 `wchar_t` 类型的字符串都称为“宽”字符串，但该术语通常特指 `wchar_t` 类型的字符串。

在 C++ 标准库中，basic_string 类型专用于窄字符串和宽字符串。 字符的类型为 `std::string` 时，使用 `char`；字符的类型为 `std::u8string` 时，使用 `char8_t`；字符的类型为

`std::u16string` 时，使用 **`char16_t`**；字符的类型为 `std::u32string` 时，使用 **`char32_t`**；而字符的类型为 `std::wstring` 时，使用 **`wchar_t`**。

其他表示文本的类型（包括 std::stringstream 和 std::cout）均可专用于窄字符串和宽字符串。

Last updated on 2025/11/07

__int8、__int16、__int32、__int64

Microsoft 专用

Microsoft C/C++ 功能支持固定大小整数类型。 可使用 `__intN` 类型说明符声明 8 位、16 位、32 位或 64 位整数变量，其中 `N` 为 8、16、32 或 64。

以下示例为这些类型的固定大小整数声明了一个变量：

C++

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

`__int8`、`__int16` 和 `__int32` 类型是大小相同的 ANSI 类型的同义词，用于编写在多个平台中具有相同行为的可移植代码。 `__int8` 数据类型是 `char` 类型的同义词，`__int16` 是 `short` 类型的同义词，而 `__int32` 是 `int` 类型的同义词。 `__int64` 类型是 `long long` 类型的同义词。

为了与以前的版本兼容，除非指定了编译器选项 `_int16`，否则 `_int32`、`_int64`、`__int8` 和 `__int16` 分别是 `__int32`、`__int64`、`/Za` 和 的同义词。

示例

以下示例显示 `__intN` 参数将提升为 `int`：

C++

```
// 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
}
```

Output

```
func
```

另请参阅

关键字
内置类型
数据类型范围

Last updated on 2025/11/07

`__m64`

Microsoft 专用

`__m64` 数据类型用于 MMX 和 3DNow! 内部函数，并在 <xmmintrin.h> 中定义。

C++

```
// data_types__m64.cpp
#include <xmmintrin.h>

int main()
{
    __m64 x;
}
```

备注

不应直接访问 `__m64` 字段。 但是，可在调试器中查看这些类型。 `__m64` 类型的变量将映射到 MM[0-7] 寄存器。

`__m64` 类型的变量在 8 字节边界上自动对齐。

x64 处理器不支持 `__m64` 数据类型。 必须将使用 `__m64` 作为 MMX 内部函数一部分的应用程序重写，以使用等效的 SSE 和 SSE2 内部函数。

结束 Microsoft 专用

另请参阅

关键字
内置类型
数据类型范围

Last updated on 2025/11/07

__m128

Microsoft 专用

可与流式处理 SIMD 扩展和流式处理 SIMD 扩展 2 内部指令一起使用的 __m128 数据类型在 xmmintrin.h< 中定义。 >

C++

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

注解

不应直接访问 `__m128` 字段。 但是，可在调试器中查看这些类型。 类型为 `__m128` 的变量映射到 XMM[0-7] 寄存器。

`__m128` 类型的变量将在 16 字节边界上自动对齐。

ARM 处理器不支持 `__m128` 数据类型。

结束 Microsoft 专用

另请参阅

关键字
内置类型
数据类型范围

Last updated on 2025/11/07

__m128d

Microsoft 专用

`__m128d` 数据类型，可与流式处理 SIMD 扩展 2 内部指令一起使用，在<emmintrin.h>中定义。

C++

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

备注

不应直接访问 `__m128d` 字段。 但是，可在调试器中查看这些类型。 类型为 `__m128` 的变量映射到 XMM[0-7] 寄存器。

类型为 _m128 的变量自动按 16 字节边界对齐。

ARM 处理器不支持 `__m128d` 数据类型。

结束 Microsoft 专用

另请参阅

关键字
内置类型
数据类型范围

Last updated on 2025/11/07

__m128i

Microsoft 专用

`__m128i` 数据类型，可与流式处理 SIMD 扩展 2 (SSE2) 内部指令一起使用，在 <emmintrin.h> 中定义。

C++

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

注解

不应直接访问 `__m128i` 字段。 但是，可在调试器中查看这些类型。 类型为 `__m128i` 的变量映射到 XMM[0-7] 寄存器。

`__m128i` 类型的变量将在 16 字节边界上自动对齐。

> **注意**
>
> 使用 `__m128i` 类型的变量将导致编译器生成 SSE2 `movdqa` 指令。 此指令不会导致 Pentium III 处理器出现故障，但会导致静默故障，并且可能会因 `movdqa` 在 Pentium III 处理器上转换为任意指令而产生副作用。

ARM 处理器不支持 `__m128i` 数据类型。

结束 Microsoft 专用

另请参阅

关键字
内置类型
数据类型范围

Last updated on 2025/11/07

__ptr32、__ptr64

Microsoft特定

用于 `__ptr32` 表示 32 位系统上的本机指针。 用于 `__ptr64` 表示 64 位系统上的本机指针。

和`__ptr32` `__ptr64`修饰符是特定于互作方案的扩展Microsoft。 对于标准 32 位（x86）或标准 x64 代码，请改用本机指针。

以下示例演示如何声明所有这些指针类型：

C++

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

在 32 位系统上，声明的 `__ptr64` 指针被视为 32 位指针，并截断分配给它的任意 64 位地址的上 32 位。 在 64 位系统上，声明的 `__ptr32` 指针被视为 32 位指针，并截断分配给它的任意 64 位地址的上 32 位。 如果 64 位地址高于 4GB，则此截断可能会导致指针无效。

> **注意**
>
> 不能使用 `__ptr32` 或 `__ptr64` 编译时使用 `/clr:pure`。 否则，编译器将生成错误 C2472。 Microsoft Visual Studio 2015 中弃用 **/clr：pure** 和 **/clr：safe** 编译器选项，Microsoft Visual Studio 2017 中不受支持。

为了与以前的版本兼容，并且`_ptr64`是以前版本的同义词`__ptr64` `__ptr32`， `_ptr32`除非你指定编译器选项 /Za （禁用语言扩展）。

示例

以下示例演示如何使用 `__ptr32` 和 `__ptr64` 关键字声明和分配指针。

此代码适用于 x86，但在 x64 上可能会崩溃。

- 它适用于 32 位编译时，因为 `__ptr64` 指针被视为 x86 上的 32 位指针。 在 x86（32 位）上， `malloc` 返回适合 `p64`的 32 位地址。
- 编译为 64 位时，它可能会崩溃，因为在 x64 上，返回一个 64 位指针， `malloc`此行截断为 32 位： `p32 = (int* __ptr32)malloc(4);` 如果分配超过 4GB，则截断 64 位地址到 32 位地址可能会导致指针无效。 在这种情况下， `*p32 = 32` 可能会尝试访问不属于进程的地址空间的截断地址，从而导致访问冲突。 即使它工作一次，如果内存分配器返回更高的地址，它以后也会失败。

C++

```cpp
#include <cstdlib>
#include <iostream>

int main()
{
    using namespace std;

    int* __ptr32 p32;
    int* __ptr64 p64;

    p64 = (int* __ptr64)malloc(4);
    *p64 = 64; // Works on x86 and x64
    cout << *p64 << endl;

    p32 = (int* __ptr32)malloc(4);
    *p32 = 32; // Works on x86. Possible exception on x64
    cout << *p32 << endl;
}
```

Output

```
64
32
```

结束 Microsoft 专用

另请参阅

内置类型

Last updated on 2025/12/18

数字限制 (C++)

这两个标准包含文件（limits.h 和 float.h）定义了数字限制或给定类型的变量可包含的最小值和最大值<><>。 这些最小值和最大值保证可移植到使用与 ANSI C 相同的数据表示法的任何 C++ 编译器。 <limits.h> 包含文件定义了整型类型的数字限制， <float.h> 定义了浮点类型的数字限制。

另请参阅

基本概念

Last updated on 2025/11/07

整数限制

Microsoft 专用

下表列出了整数类型的限制。 在包含标准头文件 <climit> 时，还会为这些限制定义预处理器宏。

对整数常量的限制

展开表

常量	含义	值
`CHAR_BIT`	不是位域的最小变量中的位数。	8
`SCHAR_MIN`	`signed char` 类型的变量的最小值。	-128
`SCHAR_MAX`	`signed char` 类型的变量的最大值。	127
`UCHAR_MAX`	`unsigned char` 类型的变量的最大值。	255 （0xff）
`CHAR_MIN`	`char` 类型的变量的最小值。	-128；如果使用了 `/J` 选项，则为 0
`CHAR_MAX`	`char` 类型的变量的最大值。	127；如果使用了 `/J` 选项，则为 255
`MB_LEN_MAX`	多字符常量中的最大字节数。	5
`SHRT_MIN`	`short` 类型的变量的最小值。	-32768
`SHRT_MAX`	`short` 类型的变量的最大值。	32767
`USHRT_MAX`	`unsigned short` 类型的变量的最大值。	65535 （0xffff）
`INT_MIN`	`int` 类型的变量的最小值。	-2147483648
`INT_MAX`	`int` 类型的变量的最大值。	2147483647
`UINT_MAX`	`unsigned int` 类型的变量的最大值。	4294967295（0xffffffff）
`LONG_MIN`	`long` 类型的变量的最小值。	-2147483648
`LONG_MAX`	`long` 类型的变量的最大值。	2147483647
`ULONG_MAX`	`unsigned long` 类型的变量的最大值。	4294967295（0xffffffff）
`LLONG_MIN`	类型为 `long long` 的变量的最小值	-9223372036854775808
`LLONG_MAX`	类型为 `long long` 的变量的最大值	9223372036854775807

常量	含义	值
`ULLONG_MAX`	类型为 `unsigned long long` 的变量的最大值	18446744073709551615（0xffffffffffffffff）

如果值超出了最大整数表示形式，则 Microsoft 编译器会产生错误。

另请参阅

浮点限制

Last updated on 2025/11/07

浮点限制

Microsoft 专用

下表列出了浮点常数的值的限制。 标准头文件 <float.h> 中也定义了这些限制。

对浮点常量的限制

展开表

返回的常量	含义	值

`FLT_DIG`
`DBL_DIG`
`LDBL_DIG`
	位数 q ，以便 q 十进制数的浮点数可以被舍入到浮点表示形式并返回，而不会丢失精度。	6
		15
		15

`FLT_EPSILON`
`DBL_EPSILON`
`LDBL_EPSILON`
	最小正数 x，以便 x + 1.0 不等于 1.0。	1.192092896e-07F
		2.2204460492503131e-016
		2.2204460492503131e-016

`FLT_GUARD`		0

`FLT_MANT_DIG`
`DBL_MANT_DIG`
`LDBL_MANT_DIG`
	由浮点有效位数中的 `FLT_RADIX` 指定的基数中的位数。基数为 2；因此这些值指定位。	24
		53
		53

`FLT_MAX`
`DBL_MAX`
`LDBL_MAX`
	可表示的最大浮点数。	3.402823466e+38F
		1.7976931348623158e+308
		1.7976931348623158e+308

`FLT_MAX_10_EXP`
`DBL_MAX_10_EXP`
`LDBL_MAX_10_EXP`
	最大整数，以便 10 的该数字的幂是一个可表示的浮点数。	38
		308
		308

`FLT_MAX_EXP`
`DBL_MAX_EXP`
`LDBL_MAX_EXP`
	最大整数，以便 `FLT_RADIX` 的该数字的幂是一个可表示的浮点数。	128
		1024
		1024

`FLT_MIN`
`DBL_MIN`
`LDBL_MIN`
	最小正值。	1.175494351e-38F
		2.2250738585072014e-308
		2.2250738585072014e-308

`FLT_MIN_10_EXP`
`DBL_MIN_10_EXP`
`LDBL_MIN_10_EXP`
	最小负整数，以便 10 的该数字的幂是一个可表示的浮点数。	-37
		-307
		-307

`FLT_MIN_EXP` `DBL_MIN_EXP`	最小负整数，以便 `FLT_RADIX` 的该数字的幂是一个可表示的浮点数。	-125 -1021

返回的常量	含义	值
`LDBL_MIN_EXP`		-1021
`FLT_NORMALIZE`		0

`FLT_RADIX`
`_DBL_RADIX`
`_LDBL_RADIX`
	基数的指数表示形式。	2
		2
		2

`FLT_ROUNDS`
`_DBL_ROUNDS`
`_LDBL_ROUNDS`
	浮点加法的舍入模式。	1（相邻）
		1（相邻）
		1（相邻）

> **注意**
>
> 表中信息可能与该产品的将来版本不同。

结束 Microsoft 专用

另请参阅

整数限制

Last updated on 2025/11/07

声明和定义 (C++)

C++ 程序由各种实体组成，例如变量、函数、类型和命名空间。 必须先声明其中的每个实体才能使用它们。 声明指定实体的唯一名称，以及有关其类型和其他特征的信息。 在 C++ 中，声明名称的位置就是它对编译器可见的位置。 无法引用稍后在编译单元中某个位置声明的函数或类。 变量应尽可能在靠近其使用位置之前的位置声明。

下面的示例演示了一些声明：

C++

```
#include <string>

int f(int i); // forward declaration

int main()
{
    const double pi = 3.14; //OK
    int i = f(2); //OK. f is forward-declared
    C obj; // error! C not yet declared.
    std::string str; // OK std::string is declared in <string> header
    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{/*...*/};
}
```

在第 5 行，声明了 `main` 函数。 在第 7 行，声明并初始化了名为 `const` 的 `pi` 变量。 在第 8 行，使用函数 `i` 生成的值声明并初始化了整数 `f`。 由于第 3 行中的前向声明，名称 `f` 将对编译器可见。

在第 9 行，声明了类型为 `obj` 的名为 `C` 的变量。 但是，此声明会引发错误，因为 `C` 只会在程序中的后期阶段声明，而不会前向声明。 若要修复该错误，可将 的整个定义移到 `C` 之前，或者为其添加前向声明`main`。 此行为不同于 C# 等其他语言。 在这些语言中，函数和类可以在源文件中的声明位置之前使用。

在第 10 行，声明了类型为 `str` 的名为 `std::string` 的变量。 名称 `std::string` 可见，因为它是在 `string`头文件中引入的，该文件已合并到第 1 行的源文件中。 `std` 是声明 `string` 类的命名空间。

在第 11 行，由于名称 `j` 尚未声明，因此会引发错误。 与 JavaScript 等其他语言不同，声明必须提供类型。 在第 12 行，使用了 `auto` 关键字，告知编译器根据用于初始化 `k` 的值来推理其类型。 在本例中，编译器选择 `int` 作为类型。

声明范围

声明引入的名称在出现声明的作用域内有效。 在以上示例中，在 `main` 函数内部声明的变量是局部变量。 可以在 main 外部的全局作用域内声明名为 `i` 的另一个变量，它将是一个单独的实体。 但是，这种名称重复可能导致编程器混淆和错误，应该避免。 在第 21 行，在命名空间 `C` 的作用域内声明了类 `N`。 使用命名空间有助于避免名称冲突。 大多数 C++ 标准库名称都在 `std` 命名空间中声明。 有关作用域规则如何与声明交互的详细信息，请参阅作用域。

定义

必须定义并声明某些实体，包括函数、类、枚举和常量变量。 稍后在程序中使用实体时，定义将为编译器提供生成计算机代码所需的所有信息。 在以上示例中，第 3 行包含函数 `f` 的声明，但该函数的定义在第 15 到 18 行中提供。 在第 21 行，声明和定义了类 `C`（不过，定义该类没有任何意义）。 必须在声明常量变量的同一语句中定义该变量，换言之，为其赋值。 内置类型（例如 `int`）的声明将自动成为定义，因为编译器知道要为其分配多少空间。

以下示例演示的声明也是定义：

C++

```
// 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;
};
```

下面这些声明不是定义：

C++

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

Typedef 和 using 语句

在早期版本的 C++ 中，typedef 关键字用于声明一个新名称，该名称是另一个名称的别名。 例如，类型 `std::string` 是 `std::basic_string<char>` 的另一个名称。 编程器使用 typedef 名称而不是实际名称的原因应该很明显。 新式 C++ 优先使用 using 关键字而不是 **`typedef`**，但思路是相同的：为已声明并定义的实体声明一个新名称。

静态类成员

静态类数据成员是类的所有对象共享的离散变量。 由于它们是共享的，因此必须在类定义的外部定义和初始化。 有关详细信息，请参阅类。

外部声明

C++ 程序可能包含多个编译单元。 若要声明在单独的编译单元中定义的实体，请使用 extern 关键字。 声明中的信息对于编译器而言已足够。 但是，如果在链接步骤中找不到实体的定义，则链接器将引发错误。

本部分内容

存储类
const
constexpr
extern
初始值设定项
别名和 typedef
using 声明
volatile
decltype
C++ 中的特性

另请参阅

基本概念

Last updated on 2025/11/07

存储类

C++ 变量声明上下文中的存储类是管理对象的生存期、链接和内存位置的类型说明符。 给定对象只能有一个存储类。 除非使用 `extern`、`static` 或 `thread_local` 说明符另行指定，否则在块中定义的变量具有自动存储。 自动对象和变量不具有链接；它们对于块外部的代码是不可见的。 在执行进入块时，会自动为其分配内存，并在退出块时取消分配内存。

说明

- 可将 mutable 关键字视为存储类说明符。 但是，它只存在于类定义的成员列表中。
- Visual Studio 2010 及更高版本： `auto` 关键字不再是 C++ 存储类说明符，且 `register` 关键字已弃用。 **Visual Studio 2017 版本 15.7 及更高版本：** （在 /std:c++17 模式及更高版本中可用）： `register` 关键字已从 C++ 语言中删除。 使用它会导致以下诊断消息：

  C++
  ```
  // c5033.cpp
  // compile by using: cl /c /std:c++17 c5033.cpp
  register int value; // warning C5033: 'register' is no longer a supported
  storage class
  ```

`static`

`static` 关键字可用于在全局范围、命名空间范围和类范围声明变量和函数。 静态变量还可在本地范围声明。

静态持续时间意味着，在程序启动时分配对象或变量，并在程序结束时释放对象或变量。 外部链接意味着，变量的名称在用于声明变量的文件的外部是可见的。 相反，内部链接意味着，名称在用于声明变量的文件的外部是不可见的。 默认情况下，在全局命名空间中定义的对象或变量具有静态持续时间和外部链接。 在以下情况下，可使用 `static` 关键字。

1. 在文件范围（全局和/或命名空间范围）内声明变量或函数时， `static` 关键字将指定变量或函数具有内部链接。 在声明变量时，变量具有静态持续时间，并且除非您指定另一个值，否则编译器会将变量初始化为 0。
2. 在函数中声明变量时， `static` 关键字将指定变量将在对该函数的调用中保持其状态。
3. 在类声明中声明数据成员时， `static` 关键字将指定类的所有实例共享该成员的一个副本。 必须在文件范围内定义 `static` 数据成员。 声明为 `const static` 的整型数据成员可以有初始化表达式。

4. 在类声明中声明成员函数时，`static` 关键字将指定类的所有实例共享该函数。 由于函数没有隐式 `static` 指针，因此 `this` 成员函数不能访问实例成员。 若要访问实例成员，请使用作为实例指针或引用的参数来声明函数。

5. 不能将 `union` 的成员声明为 `static`。 但是，全局声明的匿名 `union` 必须是显式声明的 `static`。

此示例说明了函数中声明的变量 `static` 如何在对该函数的调用间保持其状态。

C++

```
// 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 );
}
```

Output

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

此示例说明了 `static` 在类中的用法。

C++

```
// 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;
}
```

Output

```
0
0
1
1
2
2
3
3
```

以下示例显示了成员函数中声明的局部变量 `static`。 `static` 变量对整个程序可用；该类型的所有实例共享 `static` 变量的同一副本。

C++

```
// 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);
}
```

Output

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

从 C++11 开始，可以保证 `static` 局部变量初始化是线程安全的。 此功能有时称为神奇的静态对象。 但是，在多线程应用程序中，必须同步所有后续分配。 可以通过使用 /Zc:threadSafeInit- 标志避免对 CRT 形成依赖，来禁用线程安全的静态初始化功能。

`extern`

声明为 `extern` 的对象和变量将在另一个翻译单元或在一个封闭范围中定义的对象声明为具有外部链接。 有关详细信息，请参阅 extern 和翻译单元和链接。

`thread_local` (C++11)

使用 `thread_local` 说明符声明的变量仅可在它在其上创建的线程上访问。 变量在创建线程时创建，并在销毁线程时销毁。 每个线程都有其自己的变量副本。 在 Windows 上，`thread_local` 在功能上等效于特定于 Microsoft 的 __declspec(thread) 属性。

C++

```
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;
}
```

有关 `thread_local` 说明符的注意事项：

- DLL 中动态初始化的线程局部变量可能无法在所有调用线程上正确初始化。 有关详细信息，请参阅 thread。
- `thread_local` 说明符可以与 `static` 或 `extern` 合并。
- 可以将 `thread_local` 仅应用于数据声明和定义； `thread_local` 不能用于函数声明或定义。
- 只能在具有静态存储持续时间的数据项上指定 `thread_local`，其中包括全局数据对象（`static` 和 `extern`）、局部静态对象和类的静态数据成员。 如果未提供其他存储类，则声明 `thread_local` 的任何局部变量都为隐式静态；换句话说，在块范围内， `thread_local` 等效于 `thread_local static`。
- 必须为线程本地对象的声明和定义指定 `thread_local`，无论声明和定义是在同一文件中发生还是在单独的文件中发生。
- 不建议将 `thread_local` 变量与 `std::launch::async` 一起使用。 有关详细信息，请参阅 <future> 函数。

在 Windows 上， `thread_local` 在功能上等效于 __declspec(thread)，但 _`*__declspec(thread)`* 可以应用于类型定义并且在 C 代码中有效。 请尽可能使用 `thread_local`，因为它是 C++ 标准的一部分，因此更易于移植。

注册

Visual Studio 2017 版本 15.3 及更高版本（在 /std:c++17 模式及更高版本中可用）： `register` 关键字不再是受支持的存储类。 使用它会导致以下诊断： 关键字仍会保留在标准中以供将来使用。

C++

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

示例： 自动初始化与静态初始化

当控制流到达其定义时，就会初始化本地自动对象或变量。 当控制流首次到达其定义时，将初始化本地静态对象或变量。

考虑以下示例，该示例定义一个记录对象的初始化和析构的类，然后定义三个对象（即 `I1`、`I2` 和 `I3`）：

C++

```cpp
// 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";
}
```

Output

```
Initializing: Auto I1
In block.
Initializing: Auto I2
Initializing: Static I3
Destroying: Auto I2
Exited block.
Destroying: Auto I1
Destroying: Static I3
```

此示例演示如何以及何时初始化对象 `I1`、`I2` 和 `I3` 以及何时销毁它们。

有关程序，需要注意几点：

- 首先，当控制流退出在其中定义 `I1` 和 `I2` 的块时，二者将自动被销毁。
- 其次，在 C++ 中，没有必要在块的开始处声明对象或变量。 此外，只有当控制流到达其定义时，才会初始化这些对象。 （`I2` 和 `I3` 就是此类定义的示例。）输出准确地显示了它们初始化的时间。
- 最后，静态局部变量（如 `I3`）在程序运行期间保留其值，但在程序终止时将被销毁。

另请参阅

声明和定义

Last updated on 2025/11/07

`alignas` (C++)

`alignas` 说明符更改内存中类型或对象的对齐方式。

语法

C++

```
alignas(expression)
alignas(type-id)
alignas(pack...)
```

备注

可以对 `alignas`、`struct`、`class` 或变量声明使用 `union` 说明符。

对于 `alignas(expression)`，表达式必须是 0 或幂为 2（1、2、4、8、16、...）的整型常量表达式。所有其他表达式的格式不正确。

使用 `alignas`，而不是 __declspec(align(#)) 实现代码可移植性。

`alignas` 的常见用途是控制用户定义类型的对齐方式，如以下示例所示：

C++

```
struct alignas(8) S1
{
    int x;
};

static_assert(alignof(S1) == 8, "alignof(S1) should be 8");
```

当多个 `alignas` 应用于同一声明时，将使用具有最大值的声明。 忽略 `alignas` 的 `0` 值。

以下示例演示如何将 `alignas` 与用户定义的类型结合使用：

C++

```
class alignas(4) alignas(16) C1 {};

// `alignas(0)` ignored
union alignas(0) U1
{
    int i;
    float f;
};
```

```
union U2
{
    int i;
    float f;
};

static_assert(alignof(C1) == 16, "alignof(C1) should be 16");
static_assert(alignof(U1) == alignof(U2), "alignof(U1) should be equivalent to
alignof(U2)");
```

可以将类型作为对齐值提供。 该类型的默认对齐方式用作对齐值，如以下示例所示：

C++

```
struct alignas(double) S2
{
    int x;
};

static_assert(alignof(S2) == alignof(double), "alignof(S2) should be equivalent to
alignof(double)");
```

模板参数包（`alignas (pack...)`）可用于对齐值。 使用包中所有元素的最大对齐值。

C++

```
template <typename... Ts>
class alignas(Ts...) C2
{
    char c;
};

static_assert(alignof(C2<>) == 1, "alignof(C2<>) should be 1");
static_assert(alignof(C2<short, int>) == 4, "alignof(C2<short, int>) should be 4");
static_assert(alignof(C2<int, float, double>) == 8, "alignof(C2<int, float, double>)
should be 8");
```

如果应用了多个 **`alignas`**，则生成的对齐是所有 **`alignas`** 值中最大的，并且不能小于应用于它的类型的自然对齐方式。

用户定义的类型的声明和定义必须具有相同的对齐值。

C++

```
// Declaration of `C3`
class alignas(16) C3;

// Definition of `C3` with differing alignment value
class alignas(32) C3 {}; // Error: C2023 'C3': Alignment (32) different from prior
```

```
declaration (16)

int main()
{
    alignas(2) int x; // ill-formed because the natural alignment of int is 4
}
```

另请参阅

#pragma pack
对齐
alignof
编译器错误 C2023
编译器警告 C4359

Last updated on 2025/11/07

`auto` （C++）

从其初始化表达式中推导声明的变量的类型。

> ⓘ **注意**
>
> C++ 标准为此关键字定义了初始和修订的含义。 在 Visual Studio 2010 之前，`auto` 关键字在*自动存*储类中声明变量；即，具有局部生存期的变量。 从 Visual Studio 2010 开始，关键字 `auto` 声明其类型从其声明中的初始化表达式推导出的变量。 /Zc:auto[-] 编译器选项控制 `auto` 关键字的意义。

语法

> `auto` *声明符初始值设定项*`;`

> `[](auto` *param1*`, auto` *param2*`) {};`

注解

`auto` 关键字指示编译器使用已声明变量的初始化表达式或 lambda 表达式参数来推导其类型。

在大多情况下，建议使用 `auto` 关键字（除非确实需要转换），因为此关键字具有以下好处：

- **可靠性**：如果表达式的类型发生更改（包括函数返回类型发生更改的情况），它也能工作。
- **性能**：确保不会进行转换。
- **可用性**：不必担心类型名称拼写困难和拼写有误。
- **效率**：代码会变得更高效。

可能不需要使用 `auto` 的转换情况：

- 你需要一个特定类型，任何其他类型都不行。
- 例如，在表达式模板帮助程序类型 `(valarray+valarray)` 中。

若要使用 `auto` 关键字，请使用它而不是类型来声明变量，并指定初始化表达式。 此外，还可通过使用说明符和声明符（如 `auto`、`const`）、指针（`volatile`）、引用（`*`）以及右值引用（`&`）来修改 `&&` 关键字。 编译器计算初始化表达式，然后使用该信息来推断变量类型。

`auto` 初始化表达式可以采用多种形式：

- 通用初始化语法，例如 `auto a { 42 };`。
- 赋值语法，例如 `auto b = 0;`。
- 通用赋值语法，它结合了上述两种形式，例如 `auto c = { 3.14159 };`。
- 直接初始化或构造函数样式的语法，例如 `auto d( 1.41421f );`。

有关详细信息，请参阅初始值设定项和本文档后面的代码示例。

当 `auto` 用于在基于范围的 `for` 语句中声明循环参数时，它使用不同的初始化语法，例如 `for (auto& i : iterable) do_action(i);`。 有关详细信息，请参阅基于范围的 for 语句 (C++)。

`auto` 关键字是类型的占位符，但它本身不是类型。 因此，`auto` 关键字不能用于强制转换或运算符，如 sizeof 和（用于 C++/CLI）typeid。

有用性

`auto` 关键字是声明复杂类型变量的简单方法。 例如，可使用 `auto` 声明一个变量，其中初始化表达式涉及模板、指向函数的指针或指向成员的指针。

也可使用 `auto` 声明变量并将其初始化为 lambda 表达式。 您不能自行声明变量的类型，因为仅编译器知道 lambda 表达式的类型。 有关详细信息，请参阅 Lambda 表达式示例。

尾部的返回类型

您可将 `auto` 与 `decltype` 类型说明符一起使用来帮助编写模板库。 使用 `auto` 和 `decltype` 声明其返回类型取决于其模板自变量类型的函数模板。 或者，使用 `auto` 和 `decltype` 声明函数模板，该模板包装对其他函数的调用，然后返回任何返回类型的其他函数。 有关详细信息，请参阅 decltype。

引用和 cv 限定符

使用 `auto` 会删除引用、`const` 限定符和 `volatile` 限定符。 请考虑以下示例：

C++

```
// 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;
}
```

在前面的示例中，myAuto 是 `int`，而不是引用 `int`，因此，如果引用限定符尚未被 `11 11` 删除，则输出为 `11 12` 而不是 `auto`。

使用括号初始值设定项 (C++14) 的类型推导

下面的代码示例演示如何使用大括号初始化 `auto` 变量。 请注意 B 和 C 与 A 与 E 之间的差异。

C++

```
#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;
}
```

限制和错误消息

下表列出了使用 `auto` 关键字的限制，及编译器发出的相应诊断错误消息。

展开表

错误号	说明
C3530	`auto` 关键字不能与任何其他类型说明符组合。
C3531	使用 `auto` 关键字声明的符号必须具有初始值设定项。
C3532	你错误地使用了 `auto` 关键字来声明类型。 例如，声明了方法返回类型或数组。
C3533、C3539	不能使用 `auto` 关键字声明参数或模板自变量。
C3535	不能使用 `auto` 关键字声明的方法或模板自变量。
C3536	符号初始化之前无法使用。 在实践中，这意味着无法使用变量来初始化自身。
C3537	无法强制转换为使用 `auto` 关键字声明的类型。
C3538	使用 `auto` 关键字声明的声明符列表中的所有符号必须解析为相同的类型。 有关详细信息，请参阅声明和定义。
C3540、C3541	sizeof 和 typeid 运算符不能应用于使用 `auto` 关键字声明的符号。

示例

这些代码片段阐释了可使用 `auto` 关键字的一些方法。

下面的声明等效。 在第一个语句中，将变量 `j` 声明为类型 `int`。 在第二个语句中，将变量 `k` 推导为类型 `int`，因为初始化表达式 (0) 是整数。

C++

```
int j = 0;  // Variable j is explicitly type int.
auto k = 0; // Variable k is implicitly type int because 0 is an integer.
```

以下声明等效，但第二个声明比第一个更简单。 使用 `auto` 关键字的最令人信服的一个原因是简单。

C++

```
map<int,list<string>>::iterator i = m.begin();
auto i = m.begin();
```

当 `iter` 和范围 `elem` 循环启动时，下列代码片段将声明变量 `for` 和 `for` 的类型。

C++

```
// 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
    { /* ... */ }
}
```

下面的代码片段使用 `new` 运算符和指针声明来声明指针。

C++

```
double x = 12.34;
auto *y = new auto(x), **z = new auto(&x);
```

下一个代码片段在每个声明语句中声明多个符号。 请注意，每个语句中的所有符号将解析为同一类型。

C++

```
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.
```

此代码片段使用条件运算符 (`?:`) 将变量 `x` 声明为值为 200 的整数：

C++

```
int v1 = 100, v2 = 200;
auto x = v1 > v2 ? v1 : v2;
```

下面的代码片段将变量 `x` 初始化为类型 `int`，将变量 `y` 初始化对类型 `const int` 的引用，将变量 `fp` 初始化为指向返回类型 `int` 的函数的指针。

C++

```
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;
    //...
}
```

另请参阅

关键字
/Zc:auto（推导变量类型）
sizeof 运算符
typeid
operator new
声明和定义
Lambda 表达式的示例
初始值设定项
decltype

Last updated on 2025/11/07

`const` (C++)

当它修改数据声明时，`const` 关键字指定对象或变量不可修改。

语法

> *declarator*:
>
> *ptr-declarator*
>
> *noptr-declarator* *parameters-and-qualifiers* *trailing-return-type*
>
> *ptr-declarator*:
>
> *noptr-declarator*
>
> *ptr-operator* *ptr-declarator*
>
> *noptr-declarator*:
>
> *declarator-id* *attribute-specifier-seq* 选择
>
> *noptr-declarator* *parameters-and-qualifiers*
>
> *noptr-declarator* [*constant-expression* 选择] *attribute-specifier-seq* 选择
>
> (*ptr-declarator*)
>
> *parameters-and-qualifiers*:
>
> (*parameter-declaration-clause*) *cv-qualifier-seq* 选择
>
> *ref-qualifier* 选择 *noexcept-specifier* 选择 *attribute-specifier-seq* 选择
>
> *trailing-return-type*:
>
> -> *type-id*
>
> *ptr-operator*:
>
> * *attribute-specifier-seq* 选择 *cv-qualifier-seq* 选择
>
> & *attribute-specifier-seq* 选择
>
> && *attribute-specifier-seq* 选择
>
> *nested-name-specifier* * *attribute-specifier-seq* 选择 *cv-qualifier-seq* 选择
>
> *cv-qualifier-seq*:
>
> *cv-qualifier* *cv-qualifier-seq* 选择
>
> *cv-qualifier*:
>
> const
>
> volatile
>
> *ref-qualifier*:
>
> &
>
> &&
>
> *declarator-id*:
>
> ... 选择 *id-expression*

`const` 值

关键字 `const` 指定变量的值是常量，并告知编译器防止程序员修改它。

C++

```
// constant_values1.cpp
int main() {
   const int i = 5;
   i = 10;   // C3892
   i++;   // C2105
}
```

在C++中，可以使用 `const` 关键字而不是 #define 预处理器指令来定义常量值。 定义的 `const` 值受类型检查的约束，并可用于代替常量表达式。 在C++中，可以使用变量指定数组 `const` 的大小，如下所示：

C++

```
// constant_values2.cpp
// compile with: /c
const int maxarray = 255;
char store_char[maxarray];  // allowed in C++; not allowed in C
```

在 C 中，常量值默认为外部链接，因此它们只能在源文件中显示。 在C++中，常量值默认为内部链接，从而允许它们在头文件中显示。

关键字 `const` 也可以在指针声明中使用。

C++

```
// constant_values3.cpp
int main() {
   char this_char{'a'}, that_char{'b'};
   char *mybuf = &this_char, *yourbuf = &that_char;
   char *const aptr = mybuf;
   *aptr = 'c';   // OK
   aptr = yourbuf;   // C3892
}
```

指向声明为变量 `const` 的指针只能分配给也声明为的 `const`指针。

C++

```
// 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
}
```

可以使用指向常量数据的指针作为函数参数来防止函数修改通过指针传递的参数。

对于声明为 `const` 的对象，只能调用常量成员函数。 编译器确保永远不会修改常量对象。

C++

```
birthday.getMonth();    // Okay
birthday.setMonth( 4 ); // Error
```

可以为非常量对象调用常量或非常量成员函数。 还可以使用 `const` 关键字重载成员函数;此功能允许为常量和非常量对象调用不同版本的函数。

不能使用 `const` 关键字声明构造函数或析构函数。

`const` 成员函数

声明具有 `const` 关键字的成员函数指定该函数是一个“只读”函数，该函数不会修改其调用的对象。 常量成员函数不能修改任何非静态数据成员，也不能调用不常量的任何成员函数。 若要声明常量成员函数，请将 `const` 关键字置于参数列表的右括号后面。 声明 `const` 和定义都需要关键字。

C++

```
// 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 和C++ `const` 差异

在 C 源代码文件中定义 `const` 变量时，可以执行以下作：

```
C
const int i = 2;
```

然后，可以在另一个模块中使用此变量，如下所示：

```
C
extern const int i;
```

但是，若要在C++中获取相同的行为，必须将变量定义为 `const` ：

```
C++
extern const int i = 2;
```

与 C 类似，可以在另一个模块中使用此变量，如下所示：

```
C++
extern const int i;
```

如果要在C++源代码文件中定义变量 `extern` ，以便在 C 源代码文件中使用，请使用：

```
C++
extern "C" const int x=10;
```

以防止C++编译器进行名称混乱。

注解

在关注成员函数的参数列表后， `const` 关键字指定该函数不会修改为其调用的对象。

有关详细信息 `const`，请参阅以下文章：

- const 和 volatile 指针
- 类型限定符 （C 语言参考）
- volatile
- #define

另请参阅

关键字

Last updated on 2025/11/07

constexpr （C++）

关键字 `constexpr` 是在 C++11 中引入的，并在 C++14 中进行了改进。 它表示*常量表达式*。 与 `const` 一样，它可以应用于变量：当任何代码尝试修改值时，都会引发编译器错误。 与 `const` 不同， `constexpr` 还可以应用于函数和类构造函数。 `constexpr` 指示该值或返回值是常量，并且尽可能在编译时计算。

每当需要 `constexpr` 整数时（如在模板自变量和数组声明中），都可以使用 const 整数值。 如果在编译时（而非运行时）计算某个值，它可以使程序运行速度更快、占用内存更少。

为了限制编译时常量计算的复杂性及其对编译时间的潜在影响，C++14 标准要求常量表达式中的类型为文本类型。

语法

> `constexpr` *literal-typeidentifier=constant-expression*;
> `constexpr` *literal-typeidentifier*{*constant-expression*};
> `constexpr` *literal-typeidentifier* （*params*）；
> `constexpr` *ctor* （*params*）；

参数

params
一个或多个参数，每个参数都必须是文本类型，并且其本身必须是常量表达式。

返回值

`constexpr` 变量或函数必须返回文本类型。

constexpr 变量

`const` 和 `constexpr` 变量之间的主要区别在于， `const` 变量的初始化可以推迟到运行时。 `constexpr` 变量必须在编译时进行初始化。 所有的 `constexpr` 变量都是 `const`。

- 如果一个变量具有文本类型并且已初始化，则可以使用 `constexpr` 声明该变量。 如果初始化是由构造函数执行的，则必须将构造函数声明为 `constexpr`。
- 当满足以下两个条件时，可以将引用声明为 `constexpr`：引用的对象由常量表达式初始化，并且在初始化期间所调用的任何隐式转换也均是常数表达式。

- `constexpr` 变量或函数的所有声明都必须具有 `constexpr` 说明符。

```C++
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 函数

`constexpr` 函数是在使用需要它的代码时，可在编译时计算其返回值的函数。 使用代码需要编译时的返回值来初始化 `constexpr` 变量，或者用于提供非类型模板自变量。 当其参数是 `constexpr` 值时， `constexpr` 函数会生成编译时常量。 使用非 `constexpr` 自变量调用时，或者编译时不需要其值时，它将与正则函数一样，在运行时生成一个值。 （此双重行为使你无需编写同一函数的 `constexpr` 和非 `constexpr` 版本。）

`constexpr` 函数或构造函数是隐式 `inline`。

以下规则适用于 constexpr 函数：

- `constexpr` 函数必须只接受并返回文本类型。
- `constexpr` 函数可以是递归的。
- 在 C++20 之前， `constexpr` 函数不能是虚拟函数，并且当封闭类具有任何虚拟基类时，不能将构造函数定义为 `constexpr`。 在 C++20 及更高版本中， `constexpr` 函数可以是虚函数。 当您指定 `constexpr` 或更高版本编译器选项时，Visual Studio 2019 版本 16.10 及更高版本支持 /std:c++20 虚拟函数。
- 主体可以定义为 `= default` 或 `= delete`。
- 正文不能包含如何 `goto` 语句或 `try` 块。
- 可以将非 `constexpr` 模板的显式专用化声明为 `constexpr`：
- `constexpr` 模板的显式专用化不需要同时也是 `constexpr`：

以下规则适用于 Visual Studio 2017 及更高版本中的 `constexpr` 函数：

- 它可以包含 `if` 和 `switch` 语句，以及所有循环语句，包括 `for`、基于范围的 `for`、`while`、和 do-while。

- 它可能包含局部变量声明，但必须初始化该变量。 它必须是文本类型，不能是 `static` 或线程本地的。 本地声明的变量不需要是 `const`，并且可以变化。
- `constexpr` 非 `static` 成员函数不需要通过隐式方式 `const`。

C++

```
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;
}
```

提示

在 Visual Studio 调试器中，在调试非优化调试版本时，可以看出 `constexpr` 函数是否是通过在其内部放置一个断点在编译时计算的。 如果命中该断点，则在运行时调用该函数。 如果没有，则在编译时调用该函数。

extern constexpr

/Zc:externConstexpr 编译器选项使编译器将外部链接应用于使用 extern constexpr声明的变量。 在早期版本的 Visual Studio 中，无论是默认情况下还是在指定 **/Zc:externConstexpr-** 时，即使使用了 `constexpr` 关键字，Visual Studio 也会将内部链接应用于 `extern` 变量。 从 Visual Studio 2017 Update 15.6 开始，可以使用 /Zc:externConstexpr 选项，默认情况下它处于关闭状态。 /permissive- 选项不会启用 /Zc:externConstexpr。

示例

下面的示例展示了 `constexpr` 变量、函数和用户定义类型。 在 `main()` 的最后一个语句中，`constexpr` 成员函数 `GetValue()` 是一个运行时调用，因为不需要在编译时知道该值。

C++

```
// 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)
{
    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;
}
```

要求

Visual Studio 2015 或更高版本。

另请参阅

声明和定义
const

Last updated on 2025/11/07

`extern` （C++）

关键字 `extern` 可以应用于全局变量、函数或模板声明。 它指定符号具有*外部链接*。 有关链接的背景信息以及为何不鼓励使用全局变量，请参阅翻译单元和链接。

关键字 `extern` 具有四种含义，具体取决于上下文：

- 在非 `const` 全局变量声明中， `extern` 指定变量或函数在另一个转换单元中定义。 必须在除定义变量的文件之外的所有文件中应用 `extern`。
- 在 `const` 在变量声明中，它指定变量具有外部链接。 `extern` 必须应用于所有文件中的所有声明。 （默认情况下，全局 `const` 变量具有内部链接。）
- `extern "C"` 指定函数在别处定义并使用 C 语言调用约定。 `extern "C"` 修饰符也可以应用于块中的多个函数声明。
- 在模板声明中， `extern` 指定模板已在其他位置实例化。 `extern` 告知编译器它可以重复使用另一个实例化，而不是在当前位置创建新实例。 有关使用 `extern` 的详细信息，请参阅显式实例化。

`extern` 非 `const` 全局链接

链接器在全局变量声明之前看到 `extern` 时，它会在另一个转换单元中查找定义。 默认情况下，全局范围内的非 `const` 变量的声明是外部声明。 仅将 `extern` 用于未提供定义的声明。

C++

```
//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` 全局 `const` 链接

默认情况下， `const` 全局变量具有内部链接。 如果希望变量具有外部链接，请将 `extern` 关键字应用于定义和其他文件中的所有其他声明：

```C++
//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` 链接

在 Visual Studio 2017 版本 15.3 或更早版本中，编译器总是提供 `constexpr` 变量内部链接，即使在变量标记为 `extern` 时也是如此。 在 Visual Studio 2017 版本 15.5 或更高版本中，/Zc:externConstexpr 编译器开关启用正确且符合标准的行为。 选项最终将成为默认设置。 /permissive- 选项不启用 `/Zc:externConstexpr`。

```C++
extern constexpr int x = 10; //error LNK2005: "int const x" already defined
```

如果头文件包含声明 `extern` `constexpr` 的变量，必须将它标记为 `__declspec(selectany)`，以便正确组合其重复声明：

```C++
extern constexpr __declspec(selectany) int x = 10;
```

`extern "C"` 和 `extern "C++"` 函数声明

在 C++ 中，与字符串一起使用时， `extern` 指定其他语言的链接约定将用于声明符。 仅在之前被声明为具有 C 链接的情况下，才能访问 C 函数和数据。 但是，必须在单独编译的翻译单元中定义它们。

Microsoft C++ 支持 `"C"` 字段中的字符串 `"C++"` 和 。 所有标准包含文件都使用 `extern "C"` 语法以允许运行时库函数用于 C++ 程序。

示例

以下示例演示如何声明具有 C 链接的名称：

```C++
// 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;
```

如果一个函数具有多个链接规范，这些规范必须统一。 声明函数同时具有 C 和 C++ 链接是错误的。 此外，如果一个函数的两个声明出现在一个程序中，并且它们一个有链接规范，另一个没有，则有链接规范的声明必须是第一个。 将为已具有链接规范的函数的所有冗余声明提供第一个声明中指定的链接。 例如：

C++

```
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.
```

从 Visual Studio 2019 开始，当指定 `/permissive-` 时，编译器会检查 `extern "C"` 函数参数的声明是否也匹配。 不能重载声明为 `extern "C"` 的函数。 从 Visual Studio 2019 版本 16.3 开始，

可以使用 /Zc:externC- 选项后面的 `/permissive-` 编译器选项替代此检查。

另请参阅

关键字
翻译单元和链接
externC 中的存储类说明符
C 中的标识符行为
C 中的链接

Last updated on 2025/11/07

初始值设定项

初始值设定项可指定变量的初始值。 你可以在以下上下文中初始化变量：

- 在变量的定义中：

  C++
  ```
  int i = 3;
  Point p1{ 1, 2 };
  ```

- 作为函数的一个参数：

  C++
  ```
  set_point(Point{ 5, 6 });
  ```

- 作为函数的返回值：

  C++
  ```
  Point get_new_point(int x, int y) { return { x, y }; }
  Point get_new_point(int x, int y) { return Point{ x, y }; }
  ```

初始值设定项可以采用以下形式：

- 括号中的表达式（表达式的逗号分隔列表）：

  C++
  ```
  Point p1(1, 2);
  ```

- 等号后跟表达式：

  C++
  ```
  string s = "hello";
  ```

- 括号内的初始值设定项列表。 该列表可能为空，或可能包含一组列表，如下面的示例所示：

  C++
  ```
  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 } });
}
```

初始化类型

初始化有若干类型，可能出现在程序执行的不同点。 不同的初始化类型并不是相互排斥的。例如，列表初始化可触发值初始化，在其他情况中，它可以触发聚合初始化。

零初始化

零初始化是指将变量设置为隐式转换为该类型的零值：

- 数值变量初始化为 0（或 0.0、0.0000000000 等）。
- Char 变量初始化为 `'\0'`。
- 指针初始化为 `nullptr`。
- 数组、POD 类、结构和并的成员已初始化为零值。

零初始化在不同的时间执行：

- 在程序启动时，对具有静态持续时间的所有已命名变量进行初始化。 这些变量可以稍后再次初始化。
- 值初始化期间，对使用空大括号初始化的标量类型和 POD 类类型进行初始化。
- 对只有部分成员初始化的数组进行初始化。

以下是零初始化的一些示例：

C++

```
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 0.00000000000000000
    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'
}
```

默认初始化

类、结构和联合的默认初始化是具有默认构造函数的初始化。 可以不使用初始化表达式或使用 `new` 关键字调用默认构造函数：

C++

```
MyClass mc1;
MyClass* mc3 = new MyClass;
```

如果类、结构或联合没有默认构造函数，则编译器将发出错误。

如果定义标量变量时不使用初始化表达式，则进行默认初始化。 它们的值是不确定的。

C++

```
int i1;
float f;
char c;
```

如果定义数组时不使用初始化表达式，则进行默认初始化。 数组进行默认初始化时，其成员将进行默认初始化并具有不确定的值，如以下示例所示：

C++

```
int int_arr[3];
```

如果数组成员没有默认构造函数，则编译器将发出错误。

常量变量的默认初始化

常量变量必须与初始值设定项一起声明。 如果它们是标量类型，则会导致编译器错误，而如果是具有默认构造函数的类类型，则将导致警告：

C++

```
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
}
```

静态变量的默认初始化

如果静态变量的声明中没有初始值设定项，则初始化为 0（隐式转换为该类型）。

C++

```
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'}
}
```

有关全局静态对象初始化的详细信息，请参阅 main 函数和命令行参数。

值初始化

值初始化发生在以下情况下：

- 使用空大括号初始化来初始化已命名值
- 使用空圆括号或大括号初始化匿名临时对象
- 对象是使用 `new` 关键字和空圆括号或大括号来初始化的

值初始化执行以下操作：

- 对于至少有一个公共构造函数的类，将调用默认构造函数
- 对于没有声明构造函数的非联合类，该对象进行零初始化，并调用默认构造函数
- 对于数组，每个元素都进行值初始化

- 在其他所有情况下，变量进行零初始化

C++

```
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 0.00000000000000000
}
```

复制初始化

复制初始化是指使用一个不同的对象来初始化另一个对象。 在以下情况下，它会发生：

- 使用等号初始化变量
- 参数被传递给函数
- 从函数返回对象
- 引发或捕获异常
- 使用等号初始化非静态数据成员
- 在聚合初始化期间通过复制初始化来初始化类、结构和联合成员。 相关示例，请参阅聚合初始化。

下面的代码显示复制初始化的几个示例：

C++

```
#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();
    }
}
```

复制初始化不能调用显式构造函数。

C++

```
vector<int> v = 10; // the constructor is explicit; compiler error C2440: can't
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
```

有时，如果类的复制构造函数被删除或不可访问，复制初始化将导致编译器错误。

直接初始化

直接初始化是使用（非空）大括号或圆括号的初始化。 不同于复制初始化，它可以调用显式构造函数。 在以下情况下，它会发生：

- 使用非空大括号或圆括号初始化变量
- 变量是使用 `new` 关键字和非空大括号或圆括号来初始化的
- 使用`static_cast`初始化变量
- 在构造函数中，使用初始值设定项列表初始化基类和非静态成员
- lambda 表达式中捕获的变量的副本中

以下代码显示直接初始化的一些示例：

C++

```
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();
}
```

列表初始化

使用大括号内的初始值设定项列表初始化变量时，将发生列表初始化。 大括号内的初始值设定项列表可在以下情况中使用：

- 初始化变量
- 类是使用 `new` 关键字来初始化的
- 从函数返回对象
- 自变量传递给函数
- 直接初始化中的自变量之一
- 在非静态数据成员的初始值设定项中
- 在构造函数初始值设定项列表中

以下代码显示了列表初始化的一些示例：

C++

```
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' } };
}
```

聚合初始化

聚合初始化是针对数组或类类型（通常为结构或联合）的一种列表初始化形式：

- 没有私有或受保护成员
- 没有用户提供的构造函数，显式默认或删除的构造函数除外
- 没有基类
- 没有虚拟成员函数

> ⓘ **注意**
> 在 Visual Studio 2015 及更早版本中，不允许在聚合中对非静态成员使用大括号或等于初始值设定项。 此限制已在 C++14 标准中删除，并在 Visual Studio 2017 中实现。

聚合初始值设定项包括含等号或不含等号的括号内的初始化列表，如以下示例所示：

C++

```
#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;
}
```

应会看到以下输出：

Output

```
agg1: c: 1
agg2: Z: 2
myArr1: 1 2 3 4
myArr3: 8 9 10 0 0
```

> ⓘ **重要**
>
> 已声明但未在聚合初始化期间显式初始化的数组成员将进行零初始化，如上面的 `myArr3` 中所示。

初始化联合和结构

如果联合没有构造函数，你可以使用单个值（或使用联合的另一个实例）对其初始化。 该值用于初始化第一个非静态字段。 结构初始化与其不同，其初始值设定项中的第一个值用于初始化第一个字段，第二个值用于初始化第二个字段，依此类推。 比较以下示例中联合和结构的初始化：

C++

```
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'
}
```

初始化包含聚合的聚合

聚合类型可包含其他聚合类型，例如数组的数组、结构的数组等。 这些类型使用嵌套的大括号组进行初始化，例如：

C++

```
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'} };
}
```

引用初始化

引用类型的变量必须使用引用类型派生自的类型的对象进行初始化，或使用可转换为引用类型派生自的类型的类型的对象进行初始化。 例如：

C++

```
// 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
}
```

使用临时对象初始化引用的唯一方式是初始化常量临时对象。 初始化后，引用类型变量始终指向同一对象；不能将它修改为指向另一个对象。

尽管语法可以相同，但引用类型变量的引用和引用类型变量的赋值在语义上不同。 在前面的示例中，更改 `iVar` 和 `lVar` 的赋值看起来像初始化，但它们有不同的效果。 初始化指定引用类型变量指向的对象；赋值通过引用向引用目标对象赋值。

由于将引用类型的自变量传递给函数或从函数返回引用类型的值都是初始化，因此会正确初始化函数的形式自变量，就像它们是返回的引用一样。

只有在下列声明中才能在没有初始值设定项的情况下声明引用类型变量：

- 函数声明（原型）。 例如：

  C++

  ```
  int func( int& );
  ```

- 函数返回类型声明。 例如：

```
C++
int& func( int& );
```

- 引用类型类成员的声明。 例如：

```
C++
class c {public:   int& i;};
```

- 显式指定为 `extern` 的变量的声明。 例如：

```
C++
extern int& iVal;
```

初始化引用类型变量时，编译器将使用下图中所示的决策关系图，在创建对对象的引用或创建引用所指向的临时对象之间做出选择：

引用类型初始化的决策图

可使用同一类型的 `volatile` 对象或使用尚未声明为 `volatile` 的对象初始化对 类型（已声明为 typename&identifier）的引用。 `volatile` `volatile` 但是，不能使用该类型的`const`对象初始化它们。 同样，可使用同一类型的 `const` 对象（或具有到该类型的转换的任何内容或尚未声明为 `const` 的对象）初始化对 类型（已声明为 typename&identifier）的引用。 `const` `const` 但是，不能使用该类型的`volatile`对象初始化它们。

未使用`const`或`volatile`关键字限定的引用只能使用既未声明为`const`也未声明为`volatile`的对象进行初始化。

外部变量的初始化

自动变量、静态变量和外部变量的声明可包含初始值设定项。 但是，仅当外部变量没有声明为`extern`时，其声明中才可以包含初始值设定项。

Last updated on 2025/11/07

别名和 typedef (C++)

可以使用“别名声明”来声明一个要用作上一个声明类型的同义词的名称。 （此机制也非正式地称为“类型别名”）。 还可以使用此机制创建“别名模板”，这对自定义分配器特别有用。

语法

C++

```
using identifier = type;
```

备注

identifier
别名的名称。

type
您为其创建别名的类型标识符。

别名未引入新类型，且无法更改现有类型名称的含义。

别名的最简单形式等效于 C++03 中的 `typedef` 机制：

C++

```
// C++11
using counter = long;

// C++03 equivalent:
// typedef long counter;
```

两种形式都支持创建 `counter` 类型的变量。 对于 `std::ios_base::fmtflags` 而言，像这样的类型别名会更有用：

C++

```
// 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);
```

别名也使用函数指针，但比等效项 typedef 更易读：

C++

```
// 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;
```

`typedef` 机制的限制在于它无法使用模板。 但是，C++11 中的类型别名语法支持创建别名模板：

C++

```
template<typename T> using ptr = T*;

// the name 'ptr<T>' is now an alias for pointer to T
ptr<int> ptr_int;
```

示例

以下示例说明如何将别名模板与自定义分配器一起使用 - 在此示例中，它是一个整数矢量类型。 可以替换 `int` 的任何类型来创建一个方便别名，以便在主功能代码中隐藏复杂的参数列表。 通过在代码中使用自定义分配器，您可以提高可读性并降低引入由拼写错误导致的 Bug 的风险。

C++

```
#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;
}
```

Output

```
1701 1764 1664
```

Typedef

`typedef` 声明在其范围内引入一个名称，该名称成为声明的“类型声明”部分给定的类型的同义词。

使用 typedef 声明，可以为已由语言定义的类型和对你已声明的类型构造更短或更有意义的名称。 利用 Typedef 名称，您可以封装可能会发生更改的实现详细信息。

与 `class`、`struct`、`union` 和 `enum` 声明不同，`typedef` 声明不引入新类型；它们引入现有类型的新名称。

使用 `typedef` 声明的名称将占用与其他标识符相同的命名空间（不包括语句标签）。 因此，它们不能使用与前一个声明的名称相同的标识符（除了在类类型声明中）。 请考虑以下示例：

C++

```
// typedef_names1.cpp
// C2377 expected
typedef unsigned long UL;   // Declare a typedef name, UL.
int UL;                     // C2377: redefined.
```

适用于其他标识符的隐藏名称规则也控制使用 `typedef` 声明的名称的可见性。 因此，以下示例在 C++ 中是合法的：

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
```

另一个隐藏名称实例：

C++

```
// typedef_specifier1.cpp
typedef char FlagType;

int main()
{
}

void myproc( int )
{
    int FlagType;
}
```

当通过与 `typedef` 相同的名称声明本地范围标识符时，或者在同一范围内或内部范围内声明结构或联合的成员时，必须指定类型说明符。 例如：

C++

```
typedef char FlagType;
const FlagType x;
```

若要对标识符、结构成员或联合成员重用 `FlagType` 名称，则必须提供类型：

```C++
const int FlagType;  // Type specifier required
```

仅仅编写以下语句是不够的

```C++
const FlagType;      // Incomplete specification
```

由于 `FlagType` 被当做该类型的一部分，因此没有要重新声明的标识符。 此声明被视为非法声明，类似于：

```C++
int;  // Illegal declaration
```

可以使用 **`typedef`** 声明任何类型，包括指针、函数和数组类型。 只要定义具有与声明相同的可见性，那么在定义结构或联合类型之前，您就可以为指向结构或联合类型的指针声明 typedef 名称。

示例

`typedef` 声明的一个用途是使声明更加统一和精简。 例如：

```C++
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;"
```

若要使用 **`typedef`** 在同一声明中指定基础类型和派生类型，可以使用逗号分隔声明符。 例如：

```C++
typedef char CHAR, *PSTR;
```

下面的示例为不返回值并采用两个 int 自变量的函数提供了类型 `DRAWF`

C++

```
typedef void DRAWF( int, int );
```

上述 **`typedef`** 语句之后，声明

C++

```
DRAWF box;
```

将等效于声明

C++

```
void box( int, int );
```

`typedef` 通常与 **`struct`** 组合在一起来声明和命名用户定义的类型：

C++

```
// 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);
}
```

Output

```
10   0.990000
```

typedef 的重新声明

`typedef` 声明可用于将同一个名称重新声明为引用同一个类型。 例如：

源文件 `file1.h`：

```C++
// file1.h
typedef char CHAR;
```

源文件 `file2.h`：

```C++
// file2.h
typedef char CHAR;
```

源文件 `prog.cpp`：

```C++
// prog.cpp
#include "file1.h"
#include "file2.h"   // OK
```

文件 *`prog.cpp`* 包括两个头文件，它们都包含名称 **`typedef`** 的 `CHAR` 声明。 只要两个声明都引用同一个类型，则此类重新声明是可以接受的。

`typedef` 不能重新定义之前声明为不同类型的名称。 请考虑以下替代项 `file2.h`：

```C++
// file2.h
typedef int CHAR;     // Error
```

由于尝试了将名称 *`prog.cpp`* 重新声明为引用不同类型，编译器在 `CHAR` 中引发了一个错误。 此策略的影响范围包含了构造，例如：

```C++
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;
```

C++ 与 C 中的 Typedef

将 `typedef` 说明符与类类型一起使用受到广泛支持，这是因为在 `typedef` 声明中声明未命名的结构的 ANSI C 操作。 例如，许多 C 程序员都使用以下习语：

C++

```
// 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;
```

此类声明的优点是它允许如下声明：

C++

```
POINT ptOrigin;
```

而不是：

C++

```
struct point_t ptOrigin;
```

在 C++ 中， `typedef` 名称和实际类型（使用 `class`、 `struct`、 `union` 和 `enum` 关键字声明）之间的差异更为明显。 尽管在 `typedef` 语句中声明无名称的结构的 C 做法仍有效，但它不会提供像在 C 中一样的记数性的好处。

C++

```
// typedef_with_class_types2.cpp
// compile with: /c /W1
typedef struct {
   int POINT();
   unsigned x;
   unsigned y;
} POINT;
```

前面的示例声明使用未命名的类 `POINT` 语法声明一个名为 `typedef` 的类。 `POINT` 被视为类名称；但是，以下限制适用于通过这种方式引入的名称：

- 名称（同义词）不能出现在 `class`、 `struct` 或 `union` 前缀的后面。
- 名称不能用作类声明中的构造函数名称或析构函数名称。

总之，该语法不提供针对继承、构造或析构的任何机制。

Last updated on 2025/11/07

using 声明

`using` 声明将名称引入声明性区域，在该区域中显示 using 声明。

语法

```
using [typename] nested-name-specifier unqualified-id ;
using declarator-list ;
```

参数

nested-name-specifier 命名空间、类或枚举名称和范围解析运算符 (::) 的序列，由范围解析运算符终止。 单个范围解析运算符可用于从全局命名空间引入名称。 关键字 `typename` 是可选的，在从基类引入到类模板时，可用于解析依赖名称。

unqualified-id 非限定 ID 表达式，可以是标识符、重载运算符名称、用户定义的文本运算符或转换函数名称、类析构函数名称或模板名称和参数列表。

declarator-list [`typename`] *nested-name-specifierunqualified-id* 声明符的逗号分隔列表，后跟省略号（可选）。

注解

using 声明引入非限定名称作为在其他地方声明的实体的同义词。 它允许使用特定命名空间中的单个名称，而无需在其显示的声明区域中显式限定。 这与 using 指令相反，该指令允许命名空间中的*所有*名称在没有限定的情况下使用。 `using` 关键字还用于类型别名。

示例：类字段中的 `using` 声明

可以在类定义中使用 using 声明。

C++

```
// 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');
}
```

Output

```
In D::f()
In B::f()
In B::g()
```

示例：用于声明成员的 `using` 声明

用于声明成员时，using 声明必须引用基类的成员。

C++

```
// 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');
}
```

Output

```
In B::f()
```

示例：具有显式限定的 `using` 声明

通过使用 using 声明所声明的成员可以通过使用显式限定来引用。 `::` 前缀引用全局命名空间。

C++

```
// 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();
}
```

Output

```
In h
In f
In A::g
```

示例： `using` 声明同义词和别名

进行 using 声明时，由该声明创建的同义词仅引用在进行 using 声明时有效的定义。 using 声明后添加到命名空间的定义不是有效同义词。

`using` 声明定义的名称是其原始名称的别名。 它不会影响原始声明的类型、链接或其他属性。

C++

```
// 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);
}
```

示例：局部声明和 `using` 声明

对于命名空间中的函数，如果在声明性区域中给出了单个名称的一组局部声明和 using 声明，则它们必须全部引用同一实体，或者必须全部引用函数。

C++

```
// 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
}
```

在上面的示例中，`using B::i` 语句会导致在 `int i` 函数中声明第二个 `g()`。 `using B::f` 语句与 `f(char)` 函数不冲突，因为 `B::f` 引入的函数名称具有不同的参数类型。

示例：局部函数声明和 `using` 声明

局部函数声明不能与使用声明引入的函数具有相同的名称和类型。 例如：

C++

```
// 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)
}
```

示例： `using` 声明和继承

对于继承，当 using 声明将基类中的名称引入到派生类范围时，派生类中的成员函数将重写基类中具有相同名称和参数类型的虚拟成员函数。

C++

```cpp
// 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);
}
```

Output

```
In D::f(int)
In B::f(char)
In B::g
In D::g(char)
```

示例： `using` 声明可访问性

using 声明中提及的名称的所有实例都必须可访问。 具体而言，如果派生类使用 using 声明访问基类的成员，则成员名称必须可访问。 如果名称是重载成员函数的名称，则命名的所有函数都必须可访问。

有关成员可访问性的详细信息，请参阅成员访问控制。

C++

```cpp
// 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
};
```

另请参阅

命名空间
关键字

Last updated on 2025/11/07

可变（C++）

可用于声明可在程序中由硬件修改的对象的类型限定符。

语法

```
volatile declarator ;
```

备注

可以使用 /volatile 编译器开关来修改编译器解释此关键字的方式。

Visual Studio 根据目标体系结构以不同的方式解释 `volatile` 关键字。 对于 ARM，如果未指定 /volatile 编译器选项，则编译器的执行方式就如同指定了 /volatile:iso 一样。 对于 ARM 之外的体系结构，如果未指定 /volatile 编译器选项，则编译器的执行方式就如同指定了 /volatile:ms 一样；因此，对于 ARM 之外的体系结构，我们强烈建议在处理跨线程共享的内存时，指定 /volatile:iso，并使用显式同步基元和编译器内部函数。

可以使用 `volatile` 限定符提供对异步过程（如中断处理程序）使用的内存位置的访问权。

在对也具有 `volatile` 关键字的变量使用 时，优先考虑 `volatile`。

如果将 `struct` 成员标记为 `volatile`，则 `volatile` 将传播到整个结构。 如果结构不具有可通过使用一个指令在当前体系结构上复制的长度，则此结构上可能完全丢失 `volatile`。

如果满足下列条件之一，则 `volatile` 关键字可能对字段不起作用：

- 可变字段的长度超过可使用一条指令在当前体系结构上复制的最大大小。
- 最外层包含 `struct` 的长度 - 或如果它是可能嵌套的 `struct` 的成员 - 超过可使用一条指令在当前体系结构上复制的最大大小。

尽管处理器不会对不可缓存的内存访问重新排序，但必须将不可缓存的变量标记为 `volatile`，从而保证此编译器不会对内存访问重新排序。

声明为 `volatile` 的对象不在某些优化中使用，因为它们的值可以随时更改。 系统在请求易失对象时始终读取该对象的当前值，即使前面的指令要求从同一对象获取值也是如此。 此外，对象的值会立即在赋值时写入。

ISO 一致性

如果熟悉 C# volatile 关键字或熟悉早期版本的 Microsoft C++ 编译器 (MSVC) 中 `volatile` 的行为，请注意，在指定 `volatile` 编译器选项时，C++11 ISO 标准 关键字是不同的且在 MSVC 中受支持。 （对于 ARM，默认情况下将指定它。） C++11 ISO 标准代码中的 `volatile` 关键字仅用于硬件访问；请不要将其用于线程间通信。 对于线程间通信，请使用 <中的 std::atomic>T 等机制。

符合 ISO 的结尾

Microsoft 专用

在使用 **/volatile:ms** 编译器选项时 - 默认情况下，在面向 ARM 之外的体系结构时 - 编译器会生成额外代码来维护对可变对象的引用之间的排序，还维护对其他全局对象的引用的排序。 具体而言：

- 对可变对象进行写入（也称为可变编写）具有“发布”语义；也就是说，对指令序列中的可变对象进行写入之前发生的全局或静态对象的引用将在已编译的库中写入可变对象之前发生。
- 对可变对象进行读取（也称为可变读取）具有“获取”语义；也就是说，对指令序列中的可变对象进行读取之前发生的全局或静态对象的引用将在已编译的库中读取可变对象之前发生。

这使易失性对象可用于多线程应用程序中的内存锁和释放。

> **注意**
>
> 当它依赖于在使用 **/volatile:ms** 编译器选项时提供的增强保证时，代码是不可移植的。

结束 Microsoft 专用

另请参阅

关键字
const
固定和可变指针

Last updated on 2025/11/07

`decltype` （C++）

类型 `decltype` 说明符生成指定表达式的类型。 `decltype`类型说明符以及auto关键字主要用于编写模板库的开发人员。 使用 `auto` 和 `decltype` 声明其返回类型的函数模板取决于其模板参数的类型。 或者，使用 `auto` 并 `decltype` 声明一个函数模板，该模板包装对另一个函数的调用，然后返回包装函数的返回类型。

语法

> `decltype(` *`expression`* `)`

参数

`expression`
表达式。 有关详细信息，请参阅 表达式。

返回值

参数的类型 *`expression`* 。

注解

`decltype` Visual Studio 2010 或更高版本支持类型说明符，可用于本机或托管代码。
`decltype(auto)` Visual Studio 2015 及更高版本中支持 （C++14）。

编译器使用以下规则来确定参数的类型 *`expression`* 。

- *`expression`*如果参数是标识符或类成员访问， `decltype(expression)`则为命名的 *`expression`*实体的类型。 如果没有此类实体或 *`expression`* 参数为一组重载函数命名，编译器将生成错误消息。
- *`expression`*如果参数是对函数或重载运算符函数的调用， `decltype(expression)`则为函数的返回类型。 将忽略重载运算符周围的括号。
- *`expression`*如果参数是右值， `decltype(expression)`则为类型 *`expression`*。 *`expression`*如果参数是左值， `decltype(expression)`则为对类型的*`expression`*左值引用。

下面的代码示例演示了类型说明符的 `decltype` 一些用法。 首先，假设已编写以下语句。

```
C++
```

```
int var;
const int&& fx();
struct A { double x; };
const A* a = new A();
```

接下来，检查下表中四 `decltype` 个语句返回的类型。

展开表

声明	类型	注释
`decltype(fx());`	`const int&&`	对 .. 的 `const int` 右值引用
`decltype(var);`	`int`	变量 `var` 的类型。
`decltype(a->x);`	`double`	成员访问的类型。
`decltype((a->x));`	`const double&`	内部括号导致语句计算为表达式而不是成员访问。 由于 `a` 声明为 `const` 指针，因此类型是对的引用 `const double`。

`decltype` 和 `auto`

在 C++14 中，可以使用 `decltype(auto)` 不带尾随返回类型的函数模板来声明其返回类型取决于其模板参数的类型。

在 C++11 中，可以使用 `decltype` 尾随返回类型的类型说明符以及 `auto` 关键字来声明其返回类型依赖于其模板参数类型的函数模板。 例如，请考虑以下代码示例，其中函数模板的返回类型取决于模板参数的类型。 在代码示例中 *`UNKNOWN`* ，占位符指示无法指定返回类型。

C++

```
template<typename T, typename U>
UNKNOWN func(T&& t, U&& u){ return t + u; }
```

类型说明符的 `decltype` 引入使开发人员能够获取函数模板返回的表达式的类型。 使用稍后显示的 *替代函数声明语法* 、 `auto` 关键字和 `decltype` 类型说明符来声明 *延迟指定的* 返回类型。 后期指定的返回类型在编译声明时确定，而不是在编码时确定。

以下原型说明了备用函数声明的语法。 和 `const` `volatile` 限定符以及 `throw` 异常规范是可选的。 *`function_body`* 占位符表示一个复合语句，该语句指定函数的作用。 作为最佳编码做法，*`expression`* 语句中的 `decltype` 占位符应与语句中指定的 `return` 表达式（如果有）匹配 *`function_body`* 。

`auto` *`function_name`* `(` *`parameters`* 选择 `)` `const` 选择 `volatile` 选择 `->` `decltype(` *`expression`* `)` `noexcept` 选择 `{` *`function_body`* `};`

在下面的代码示例中，函数模板的后期指定返回类型 `myFunc` 由模板参数的类型 `t` 和 `u` 模板参数确定。 作为最佳编码做法，该代码示例还使用右值引用和 `forward` 函数模板，该模板支持 *完美的转发*。 有关详细信息，请参阅 Rvalue 引用声明符： `&&>`。

C++

```
//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` 和转发函数 （C++11）

转发函数包装对其他函数的调用。 请考虑一个函数模板，该模板将参数或涉及这些参数的表达式的结果转发到另一个函数。 此外，转发函数返回调用另一个函数的结果。 在此方案中，转发函数的返回类型应与包装函数的返回类型相同。

在此方案中，不能在没有类型说明符的情况下 `decltype` 编写适当的类型表达式。 类型 `decltype` 说明符启用泛型转发函数，因为它不会丢失有关函数是否返回引用类型的必需信息。 有关转发函数的代码示例，请参阅前面的 `myFunc` 函数模板示例。

例子

下面的代码示例声明函数模板 `Plus()` 的后期指定返回类型。 该 `Plus` 函数使用 `operator+` 重载处理其两个作数。 因此，加运算符 （`+`） 和函数的 `Plus` 返回类型的解释取决于函数参数的类型。

C++

```
// 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;
}
```

Output

```
Plus(i, 9) = 13
Plus(dx, dy) = 13.5
Hello, world!
x3.Dump() = 42
```

Visual Studio 2017 及更高版本： 编译器在声明模板而不是实例化时分析 `decltype` 参数。 因此，如果在参数中找到 `decltype` 非依赖专用化，则不会将其推迟到实例化时间;它会立即进行处理，此时会诊断任何生成的错误。

以下示例显示了在声明点引发的此类编译器错误：

C++

```cpp
#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");
```

要求

Visual Studio 2010 或更高版本。

`decltype(auto)` 需要 Visual Studio 2015 或更高版本。

Last updated on 2025/11/07

C++ 中的特性

C++ 标准定义了一组通用属性。 它还允许编译器供应商在其特定的命名空间中定义自己的属性。 但是，编译器只需识别标准中定义的属性。

在某些情况下，标准属性与编译器特定的 `__declspec` 参数重叠。 在 Microsoft C++ 中，可以使用 `[[deprecated]]` 属性而不使用 `__declspec(deprecated)`。 `[[deprecated]]` 属性可由任何符合标准的编译器识别。 对于所有其他 `__declspec` 参数（例如 `dllimport` 和 `dllexport`），到目前为止还没有等效的属性，因此必须继续使用 `__declspec` 语法。 属性不影响类型系统，也不会改变程序的含义。 编译器会忽略它们无法识别的属性值。

Visual Studio 2017 版本 15.3 和更高版本（适用于 和更高版本）：在属性列表的范围内，可以使用单个 /std:c++17 引入器为所有名称指定命名空间`using`：

C++

```
void g() {
    [[using rpr: kernel, target(cpu,gpu)]] // equivalent to [[ rpr::kernel,
rpr::target(cpu,gpu) ]]
    do task();
}
```

C++ 标准属性

在 C++11 中，属性提供一种使用附加信息来批注 C++ 构造（包括但不限于类、函数、变量和块）的标准化方式。 属性不一定特定于供应商。 编译器可以使用此信息来生成信息性消息，或者在编译属性代码时应用特殊的逻辑。 编译器会忽略它无法识别的任何属性，这意味着你无法使用此语法来定义自己的自定义属性。 属性括在双方括号中：

C++

```
[[deprecated]]
void Foo(int);
```

属性代表 `#pragma` 指令、 `__declspec()` (Visual C++) 或 `__attribute__` (GNU) 等供应商特定扩展的标准化替代项。 但是，在大多数情况下，你仍需使用供应商特定的构造。 标准目前指定了符合标准的编译器应识别的以下属性。

`[[carries_dependency]]`

`[[carries_dependency]]` 属性指定函数传播线程同步的数据依赖项顺序。 可将该属性应用于一个或多个参数，以指定传入的参数要将依赖项带入函数主体中。 可将该属性应用于函数本身，以指定返回值要将依赖项带出函数。 编译器可以使用此信息来生成更有效的代码。

`[[deprecated]]`

Visual Studio 2015 及更高版本： `[[deprecated]]` 属性指定函数不适合使用。 或者，它可能不存在于库接口的将来版本中。 `[[deprecated]]` 属性可应用于类、typedef 名称、变量、非静态数据成员、函数、命名空间、枚举、枚举器或模板专用化的声明。 当客户端代码尝试调用该函数时，编译器可以使用此属性来生成信息性消息。 当 Microsoft C++ 编译器检测到使用 `[[deprecated]]` 项时，会发出编译器警告 C4996。

`[[fallthrough]]`

Visual Studio 2017 和更高版本： （适用于 /std:c++17 和更高版本。） `[[fallthrough]]` 属性可在 switch 语句的上下文中用作向编译器（或阅读代码的任何人）显示的提示，指出失败行为是有意的。 Microsoft C++ 编译器当前不会对回退行为发出警告，因此此属性对编译器行为没有影响。

`[[likely]]`

Visual Studio 2019 版本 16.6 及更高版本： （适用于 /std:c++20 及更高版本。） `[[likely]]` 属性向编译器指定提示，指出属性化标签或语句的代码路径的执行可能性高于替代项。 在 Microsoft 编译器中， `[[likely]]` 属性将块标记为“热代码”，这会增加内部优化分数。 针对速度进行优化时，分数增加得更多；针对大小进行优化时，增加的幅度不太大。 净分数会影响内联、循环展开和向量化优化的可能性。 `[[likely]]` 和 `[[unlikely]]` 的效果类似于按配置优化，但仅限于当前转换单元的范围。 该属性尚未针对此属性实施块重新排序优化。

`[[maybe_unused]]`

Visual Studio 2017 版本 15.3 及更高版本： （适用于 /std:c++17 及更高版本。） `[[maybe_unused]]` 属性指定变量、函数、类、typedef、非静态数据成员、枚举或模板专用化可能是有意不使用的。 当未使用标记为 `[[maybe_unused]]` 的实体时，编译器不会发出警告。 未使用属性声明的实体以后可以使用属性来重新声明，反之亦然。 分析某个实体的首个标记为 的声明后，该实体被视为已标记，适用于当前转换单元的其余部分`[[maybe_unused]]`。

`[[nodiscard]]`

Visual Studio 2017 版本 15.3 和更高版本： （适用于 /std:c++17 和更高版本。）指定不应该丢弃函数的返回值。 引发警告 C4834，如以下示例中所示：

```
C++
```

```
[[nodiscard]]
int foo(int i) { return i * i; }

int main()
{
    foo(42); //warning C4834: discarding return value of function with 'nodiscard'
attribute
    return 0;
}
```

`[[noreturn]]`

`[[noreturn]]` 属性指定函数永不返回；换言之，它始终引发异常或退出。 编译器可以针对 `[[noreturn]]` 实体调整其编译规则。

`[[unlikely]]`

Visual Studio 2019 版本 16.6 及更高版本： （适用于 /std:c++20 及更高版本。） `[[unlikely]]` 属性向编译器指定提示，指出属性化标签或语句的代码路径的执行可能性低于替代项。 在 Microsoft 编译器中， `[[unlikely]]` 属性将块标记为“冷代码”，这会减少内部优化分数。 针对大小进行优化时，分数减少得更多；针对速度进行优化时，减少的幅度不太大。 净分数会影响内联、循环展开和向量化优化的可能性。 该属性尚未针对此属性实施块重新排序优化。

Microsoft 特定的属性

`[[gsl::suppress(<tag> [, justification: <narrow-string-literal>])]]`

`<tag>` 是一个字符串，用于指定要隐藏的规则的名称。 该 optional `justification` 字段允许您解释禁用或禁止显示警告的原因。 指定选项 `/analyze:log:includesuppressed` 后，此值将显示在 SARIF 输出中。 它的值是 UTF-8 编码的窄字符串文字。 该 `[[gsl::suppress]]` 属性在 Visual Studio 2022 版本 17.14 及更高版本中可用。

Microsoft 特定的 `[[gsl::suppress]]` 属性用于抑制在代码中强制实施准则支持库 (GSL) 规则的检查器发出的警告。 例如，考虑以下代码片段：

C++

```
int main()
{
    int arr[10]; // GSL warning C26494 will be fired
    int* p = arr; // GSL warning C26485 will be fired
```

```
    [[gsl::suppress("bounds.1", justification: "This attribute suppresses Bounds rule #1")]]
    {
        int* q = p + 1; // GSL warning C26481 suppressed
        p = q--; // GSL warning C26481 suppressed
    }
}
```

该示例引发了以下警告：

- C26494（类型规则 5：始终初始化对象。）
- C26485（边界规则 3：没有指针衰减的数组。）
- C26481（边界规则 1：不要使用指针算术。请改用范围。）

使用安装并激活的 CppCoreCheck 代码分析工具编译此代码时，会触发前两个警告。 但由于使用了该属性，第三个警告不会触发。 可以通过写入 `[[gsl::suppress("bounds")]]` 而不包括特定规则编号来抑制整个边界配置文件。 C++ Core Guidelines 可帮助你编写更好、更安全的代码。 使用 suppress 属性可以轻松关闭不需要的警告。

`[[msvc::flatten]]`

Microsoft 特定属性 `[[msvc::flatten]]` 与 `[[msvc::forceinline_calls]]` 非常相似，并且可以在相同位置以相同方式使用。 不同之处在于，`[[msvc::flatten]]` 会递归地 `[[msvc::forceinline_calls]]` 其应用范围内的所有调用，直到没有调用为止。 这可能会对函数的代码大小增长或编译器的吞吐量产生影响，必须手动管理。

`[[msvc::forceinline]]`

当放置在函数声明之前时，Microsoft 特定属性 `[[msvc::forceinline]]` 与 `__forceinline` 具有相同的含义。

`[[msvc::forceinline_calls]]`

Microsoft 特定属性 `[[msvc::forceinline_calls]]` 可以放置在语句或块之上或之前。 它会导致内联启发式尝试 `[[msvc::forceinline]]` 该语句或块中的所有调用：

C++

```
void f() {
    [[msvc::forceinline_calls]]
    {
        foo();
        bar();
```

```
    }
    ...
    [[msvc::forceinline_calls]]
    bar();

    foo();
}
```

对 `foo` 的第一次调用以及对 `bar` 的两次调用都被视为声明为 `__forceinline`。 对 `foo` 的第二次调用不会被视为 `__forceinline`。

`[[msvc::intrinsic]]`

`[[msvc::intrinsic]]` 属性对其应用到的函数有三个约束：

- 该函数不能具有递归性；它的主体必须只有一个 return 语句，该语句带有从参数类型到返回类型的 `static_cast`。
- 该函数只能接受单个参数。
- **`/permissive-`** 编译器选项是必需的。 （默认情况下，**`/std:c++20`** 及更高版本的选项意味着 **`/permissive-`**。）

Microsoft 特定的 `[[msvc::intrinsic]]` 属性告知编译器内联一个元函数，该元函数充当从参数类型到返回类型的命名转换。 当该属性出现在函数定义中时，编译器会用简单的强制转换替换对该函数的所有调用。 `[[msvc::intrinsic]]` 属性在 Visual Studio 2022 版本 17.5 预览版 2 及更高版本中可用。 此属性仅适用于其后面的特定函数。

示例

在此示例代码中，应用于 `[[msvc::intrinsic]]` 函数的 `my_move` 属性使编译器将该函数的调用替换为其主体中的内联静态强制转换：

C++

```cpp
template <typename T>
[[msvc::intrinsic]] T&& my_move(T&& t) { return static_cast<T&&>(t); }

void f() {
    int i = 0;
    i = my_move(i);
}
```

`[[msvc::musttail]]`

该 `[[msvc::musttail]]` 属性在 MSVC 生成工具版本 14.50 中引入，是强制实施尾调用优化的试验性 x64 仅Microsoft特定属性。 应用于限定返回语句时，它会指示编译器发出调用作为结尾调用。 如果编译器无法发出结尾调用，则会生成编译错误。 该 `[[msvc::musttail]]` 属性强制实施结尾调用，而不是将函数内联。

`[[msvc::musttail]]` 要求：

- 调用方和被调用方必须具有匹配的返回类型。
- 调用约定必须兼容。
- 尾部调用必须是调用函数中的最终作。
- 被调用方不能使用比调用函数更多的堆栈空间。
- 如果传递了四个以上的整数参数，调用函数必须为这些附加参数分配足够的堆栈空间。
- 使用 `/O2` 或 `/O2 /GL` 优化级别进行编译。

示例

Tail 调用是编译器优化，在函数调用是在返回之前执行的最后一个作时可能的。 重用当前函数的堆栈帧，而不是创建新的堆栈帧来调用函数。 这可减少堆栈使用率并提高性能，尤其是在递归方案中。

在以下代码中， `[[msvc::musttail]]` 应用于 `return increment(x)` 使控制直接 `increment` 传输到的控制的属性。 到达语句时 `increment`，其结果直接提供给调用 `incrementIfPositive` 方，即 `main`。 `return x+1;` 这将替换内联 `increment` 传入或调用 `incrementIfPositive` `increment` 和返回 `incrementIfPositive` 之前返回到 `main` `incrementIfPositive` 。 尾部调用优化无需 `incrementIfPositive` 在完成后重新获得控制 `increment` 。 这是一种性能优化，可减少堆栈使用率，在递归方案中特别有用。

C++

```
// compile with /O2
#include <iostream>

int increment(int x)
{
    return x + 1;
}

int incrementIfPositive(int x)
{
    if (x > 0)
    {
        [[msvc::musttail]]
        return increment(x);
    }
    return -1;
}
```

```
int main()
{
    int result = incrementIfPositive(42);
    if (result < 0)
    {
        return -1;
    }

    std::cout << result; // outputs 43
    return 0;
}
```

[[msvc::noinline]]

当放置在函数声明之前时，Microsoft 特定属性 `[[msvc::noinline]]` 与 `__declspec(noinline)` 具有相同的含义。

[[msvc::noinline_calls]]

Microsoft 特定属性 `[[msvc::noinline_calls]]` 的用法与 `[[msvc::forceinline_calls]]` 相同。 它可以放置在任何语句或块之前。 它不是强制内联该块中的所有调用，而是对应用到的范围禁用内联。

[[msvc::no_tls_guard]]

Microsoft 特定的 `[[msvc::no_tls_guard]]` 属性禁止在首次访问 DLL 中的线程局部变量时检查初始化。 默认情况下，在使用 Visual Studio 2019 版本 16.5 及更高版本构建的代码中启用这些检查。 此属性仅适用于其后面的特定变量。 若要全局禁用检查，请使用 /Zc:tlsGuards- 编译器选项。

Last updated on 2025/12/10

C++ 内置运算符、优先级和关联性

C++ 语言包括所有 C 运算符并添加多个新的运算符。 运算符指定对一个或多个操作数执行的计算。

优先级和结合性

运算符优先级指定了包含多个运算符的表达式中的运算顺序。 运算符关联性指定了在包含多个具有相同优先级的运算符的表达式中，操作数是与其左侧还是右侧的操作数组合。

替代拼写

C++ 为某些运算符指定了替代拼写。 在 C 中，替代拼写在 <iso646.h> 标头中作为宏提供。 在 C++ 中，这些替代项是关键字，<iso646.h> 或 C++ 等效的 <ciso646> 已弃用。 在 Microsoft C++ 中，需要 /permissive- 或 /Za 编译器选项才能启用替代拼写。

C++ 运算符的优先级和关联性表

下表显示 C++ 运算符的优先级和关联性（从最高优先级到最低优先级）。 优先级别编号相同的运算符具有等同的优先级别，除非由括号显式施加另一种关系。

展开表

运算符说明	运算符	替代项
第 1 组优先级，无关联性		
范围解析	::	
第 2 组优先级，从左到右关联		
成员选择（对象或指针）	. 或 ->	
数组下标	[]	
函数调用	()	
后缀递增	++	
后缀递减	--	
类型名称	typeid	
常量类型转换	const_cast	

运算符说明	运算符	替代项
动态类型转换	dynamic_cast	
重新解释的类型转换	reinterpret_cast	
静态类型转换	static_cast	
第 3 组优先级，从右到左关联		
对象或类型的大小	sizeof	
前缀递增	++	
前缀递减	--	
二进制反码	~	`compl`
逻辑“非”	!	`not`
一元求反	-	
一元加	+	
Address-of	&	
间接寻址	*	
创建对象	new	
销毁对象	delete	
强制转换	()	
第 4 组优先级，从左到右关联		
指向成员的指针（对象或指针）	.* 或 ->*	
第 5 组优先级，从左到右关联		
乘法	*	
除法	/	
取模	%	
第 6 组优先级，从左到右关联		
加法	+	
减法	-	
第 7 组优先级，从左到右关联		

运算符说明	运算符	替代项
左移	<<	
右移	>>	
第 8 组优先级，从左到右关联		
小于	<	
大于	>	
小于或等于	<=	
大于或等于	>=	
第 9 组优先级，从左到右关联		
等式	==	
不相等	!=	`not_eq`
第 10 组优先级，从左到右关联		
按位“与”	&	`bitand`
第 11 组优先级，从左到右关联		
按位“异或”	^	`xor`
第 12 组优先级，从左到右关联		
按位“与或”	|	`bitor`
第 13 组优先级，从左到右关联		
逻辑与	&&	`and`
第 14 组优先级，从左到右关联		
逻辑或	||	`or`
第 15 组优先级，从右到左关联		
条件	?:	
转让	=	
乘法赋值	*=	
除法赋值	/=	
取模赋值	%=	

运算符说明	运算符	替代项
加法赋值	+=	
减法赋值	-=	
左移赋值	<<=	
右移赋值	>>=	
按位“与”赋值	&=	`and_eq`
按位“与或”赋值	|=	`or_eq`
按位“异或”赋值	^=	`xor_eq`
引发表达式	throw	
第 16 组优先级，从左到右关联		
Comma	,	

另请参阅

运算符重载

Last updated on 2025/11/07

`alignof` 运算符

`alignof` 运算符将指定类型的对齐方式（以字节为单位）作为类型 `size_t` 的值返回。

语法

C++

```
alignof( type )
```

备注

例如：

展开表

表达式	值
`alignof( char )`	1
`alignof( short )`	2
`alignof( int )`	4
`alignof( long long )`	8
`alignof( float )`	4
`alignof( double )`	8

`alignof` 值与基本类型的 `sizeof` 的值相同。 但是，请考虑该示例：

C++

```
typedef struct { int a; double b; } S;
// alignof(S) == 8
```

在该示例中， `alignof` 值是结构中的最大元素的对齐需求。

同样，

C++

```
typedef __declspec(align(32)) struct { int a; } S;
```

`alignof(S)` 等于 `32`。

`alignof` 的用途之一是作为某个内存分配例程的参数。 例如，假定下面定义的结构 `S`，您可以调用名为 `aligned_malloc` 的内存分配例程以在特定对齐边界上分配内存。

C++

```
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));
```

有关修改对齐方式的详细信息，请参阅：

- pack
- align
- __unaligned
- /Zp（结构成员对齐）
- x64 结构对齐示例

有关 x86 和 x64 代码中的对齐方式的差异的详细信息，请参阅：

- 与 x86 编译器冲突

Microsoft 专用

`alignof` 和 **`__alignof`** 在 Microsoft 编译器中是同义词。 在 C++11 中成为标准的一部分之前，Microsoft 专用的 **`__alignof`** 运算符提供了此功能。 为实现最大的可移植性，应使用 **`alignof`** 运算符，而不是 Microsoft 专用的 **`__alignof`** 运算符。

为了与以前的版本兼容，除非指定了编译器选项 **`__alignof`**，否则 `/Za` 是 的同义词。

另请参阅

使用一元运算符的表达式
关键字

Last updated on 2025/11/07

`__uuidof` 运算符

Microsoft 专用

检索附加到表达式的 GUID。

语法

> `__uuidof (` *表达式* `)`

注解

此 *expression* 可以是类型名称、指针、引用或该类型的数组、专用于这些类型的模板或这些类型的变量。 只要编译器可使用参数查找附加的 GUID，此参数就有效。

将 0 或 NULL 作为自变量提供时是此内部函数的特例。 在此情况下，`__uuidof` 将返回由零组成的 GUID。

使用此关键字可将附加的 GUID 提取到：

- 由 uuid 扩展特性提供的对象。
- 使用 module 特性创建的库块。

> **注意**
>
> 在调试版本中，`__uuidof` 始终动态初始化对象（在运行时）。 在发布版本中，`__uuidof` 可静态初始化对象（在编译时）。

为了与以前的版本兼容，除非指定了编译器选项 `__uuidof`，否则 `/Za` 是 的同义词。

示例

以下代码（使用 ole32.lib 编译的）将显示使用 module 特性创建的库块的 uuid。

C++

```
// 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);
}
```

注释

如果库名不再在范围内，可使用 `__LIBID_` 代替 **`__uuidof`**。 例如：

C++

```
StringFromCLSID(__LIBID_, &lpolestr);
```

结束 Microsoft 专用

另请参阅

使用一元运算符的表达式
关键字

Last updated on 2025/11/07

加法运算符：`+` 和 `-`

语法

```
expression + expression
expression - expression
```

备注

相加运算符为：

- 加法 (+)
- 减法 (-)

这些二进制运算符具有从左至右的关联性。

相加运算符采用算术或指针类型的操作数。 加法 (+) 运算符的结果是操作数之和。 减法 (-) 运算符的结果是操作数之差。 如果一个操作数是指针或两个操作数都是指针，则它们必须是指向对象的指针，而不是指向函数的指针。 如果两个操作数都是指针，则结果没有意义，除非它们是指向同一数组中的对象的指针。

相加运算符采用 arithmetic、integral 和 scalar 类型的操作数。 下表定义了这些操作数。

用于相加运算符的类型

展开表

类型	含义
算术	整型和浮点类型统称为“算术”类型。
integral	所有大小（long、short）和枚举数的 char 和 int 类型为“整数”类型。
scalar	标量操作数是算术类型或指针类型的操作数。

这些运算符的合法组合为：

arithmeticarithmetic +

scalarintegral +

integralscalar +

arithmeticarithmetic -

scalarscalar -

请注意，加法和减法不是等效运算。

如果两个操作数都是 arithmetic 类型，则标准转换中介绍的转换适用于这两个操作数，并且结果为已转换的类型。

示例

C++

```
// 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;
}
```

指针加法

在加法运算中，如果其中一个操作数是指向对象数组的指针，则另一个操作数必须是整型。 结果为与原始指针类型相同的指针和指向另一个数组元素的指针。 以下代码片段阐述了此概念：

C++

```
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;
}
```

虽然将整数值 1 添加到 `pIntArray`，但这并不表示“将 1 添加到该地址”，而是指“调整指针使其指向数组中的下一个对象”，而该对象恰好是在 2 字节（或者 `sizeof( int )`）之外。

> **注意**
>
> 在 C++ 程序中很少找到 `pIntArray = pIntArray + 1` 形式的代码；若要实现递增，以下形式更可取： `pIntArray++` 或 `pIntArray += 1`。

指针减法

如果两个操作数都是指针，则减法运算的结果就是两个操作数之差（在数组元素中）。 减法表达式产生类型 `ptrdiff_t`（在标准包含文件 <stddef.h> 中定义）的带符号的整数结果。

其中一个操作数可以是整型，条件是该操作数是第二操作数。 减法的结果的类型与原始指针的类型相同。 减法的值是指向第 (n i) 个数组元素的指针，其中 n 是由原始指针指向的元素，而 i 是第二操作数的整数值 - 。

另请参阅

使用二元运算符的表达式
C++ 内置运算符、优先级和关联性
C 加法运算符

Last updated on 2025/11/07

地址运算符： `&`

语法

address-of-expression:
  `&` *cast-expression*

注解

一元地址运算符 （`&`） 返回其作数（即指向）作数的地址。 地址运算符的作数可以是函数设计符，也可以是引用非位字段的对象的左值。

地址运算符只能应用于某些左值表达式：无论是基本、结构、类或联合类型的变量，还是应用于下标数组引用。 在这些表达式中，可以将常量表达式（不包括运算符的常量表达式）添加到或减去表达式的地址。

应用于函数或左值时，表达式的结果是派生自作数类型的指针类型（右值）。 例如，如果作数的类型 `char`，则表达式的结果是指向 `char` 的类型指针。 应用于 `const` 或 `volatile` 对象的地址运算符的计算 `const type *` 结果为原始 `volatile type *` 对象的类型或计算 `type` 结果。

仅当明确引用哪个版本的函数时，才能获取重载函数的地址。 有关如何获取特定重载函数的地址的详细信息，请参阅 函数重载。

当地址运算符应用于限定名称时，结果取决于 *限定名称* 是否指定静态成员。 如果是这样，则结果是指向成员声明中指定的类型的指针。 对于非静态成员，结果是指向*由 qualified-class-name* 指示的类的成员*名称*的指针。 有关 *限定类名称*的详细信息，请参阅 主表达式。

示例：静态成员的地址

以下代码片段显示了运算符地址结果的不同之处，具体取决于类成员是否为静态：

C++

```
// 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
}
```

在此示例中，表达式`&PTM::fValue`生成类型而不是类型`float *``float PTM::*`，因为`fValue`是静态成员。

示例：引用类型的地址

将地址运算符应用于引用类型可提供与将运算符应用于所绑定引用的对象相同的结果。 例如：

C++

```
// 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
```

示例：函数地址作为参数

以下示例使用 address-of 运算符将指针参数传递给函数：

C++

```
// 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

```
25
```

另请参阅

具有一元运算符的表达式
C++内置运算符、优先级和关联性
左值引用声明符： &
间接运算符和地址运算符

Last updated on 2025/11/07

赋值运算符

语法

`expression` `assignment-operator` `expression`

`assignment-operator`：其中一个

`=` `*=` `/=` `%=` `+=` `-=` `<<=` `>>=` `&=` `^=` `|=`

注解

赋值运算符将值存储在左作数指定的对象中。 有两种类型的分配作：

- *简单赋值*，其中第二个作数的值存储在第一个作数指定的对象中。
- *复合赋值*，在存储结果之前执行算术、移位或按位运算。

下表中的所有赋值运算符（运算符除外 `=` ）都是复合赋值运算符。

赋值运算符表

展开表

操作员	含义
`=`	将第二个作数的值存储在第一个作数（简单赋值）指定的对象中。
`*=`	将第一个作数的值乘以第二个作数的值;将结果存储在第一个作数指定的对象中。
`/=`	将第一个作数的值除以第二个作数的值;将结果存储在第一个作数指定的对象中。
`%=`	获取由第二个作数的值指定的第一个作数的模数;将结果存储在第一个作数指定的对象中。
`+=`	将第二个作数的值添加到第一个作数的值;将结果存储在第一个作数指定的对象中。
`-=`	从第一个作数的值中减去第二个作数的值;将结果存储在第一个作数指定的对象中。
`<<=`	将第一个作数的值移出由第二个作数的值指定的位数;将结果存储在第一个作数指定的对象中。
`>>=`	将第一个作数的值向右移动由第二个作数的值指定的位数;将结果存储在第一个作数指定的对象中。
`&=`	获取第一个和第二个作数的按位 AND;将结果存储在第一个作数指定的对象中。
`^=`	获取第一个和第二个作数的按位排他 OR;将结果存储在第一个作数指定的对象中。

操作员	含义
`|=`	获取第一个和第二个作数的按位非独占 OR;将结果存储在第一个作数指定的对象中。

运算符关键字

三个复合赋值运算符具有关键字等效项。 它们分别是：

展开表

操作员	等效
`&=`	`and_eq`
`|=`	`or_eq`
`^=`	`xor_eq`

C++将这些运算符关键字指定为复合赋值运算符的替代拼写。 在 C 中，备用拼写以 iso646.h> 标头中的<宏的形式提供。 在C++中，替代拼写是关键字; <已弃用 iso646.h> 或C++等效 <ciso646> 的使用。 在Microsoft C++中， /permissive- 启用备用拼写需要或 /Za 编译器选项。

示例：

C++

```
// 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;
}
```

简单分配

简单赋值运算符 （`=`） 导致第二个作数的值存储在第一个作数指定的对象中。 如果两个对象都是算术类型，则右侧作数将转换为左侧的类型，然后存储值。

`const`对象和`volatile`类型可以分配给仅`volatile`属于或不是或不是`const`的类型`volatile`的 l 值。

类类型（和类型）的赋值由一个名为 a0/ a1/ /gt; 的函数执行。 此运算符函数的默认行为是执行对象的非静态数据成员和直接基类的成员复制分配;但是，可以使用重载运算符修改此行为。 有关详细信息，请参阅 运算符重载。 类类型还可以具有 *复制赋值* 和 *移动赋值* 运算符。 有关详细信息，请参阅 复制构造函数和复制赋值运算符 以及 移动构造函数和移动赋值运算符。

可以将给定基类中任何明确派生类的对象分配给基类的对象。 相反，因为存在从派生类到基类的隐式转换，而不是从基类转换为派生类。 例如：

C++

```
// 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
}
```

对引用类型的赋值的行为就像对引用指向的对象执行赋值一样。

对于类类型对象，赋值不同于初始化。 若要说明如何进行不同的分配和初始化，请考虑代码

C++

```
UserType1 A;
UserType2 B = A;
```

前面的代码显示初始值设定项;它调用该构造函数，该构造函数采用 `UserType2` 类型的 `UserType1`参数。 给定代码

```C++
UserType1 A;
UserType2 B;

B = A;
```

赋值语句

```C++
B = A;
```

可具有以下效果之一:

- 为其调用函数，提供的参数为 <a0/>。
- 调用显式转换函数 `UserType1::operator UserType2` （如果存在此类函数）。
- 调用构造函数，前提是存在此类构造函数，该构造函数 `UserType2::UserType2`采用参数 `UserType1` 并复制结果。

复合赋值

复合赋值 运算符显示在赋值运算符表中。 这些运算符采用 *e1op= e2* 形式，其中 *e1* 是不可 **`const`** 修改的 l 值， *e2* 为:

- 算术类型
- 指针（如果 *作* 为 `+` 或 `-`
- 存在 *e1* 类型的匹配`operator *op*=`重载的类型

内置 *e1ope2* 窗体的行为方式为 *e1 e1*`=`*op= e2*，但 *e1* 只评估一次。

对枚举类型的复合赋值将生成错误消息。 如果左侧作数是指针类型，则右作数必须是指针类型，或者必须是计算结果为 0 的常量表达式。 当左作数是整型作数时，右作数不得为指针类型。

内置赋值运算符的结果

内置赋值运算符返回赋值后左作数指定的对象值（以及复合赋值运算符的算术/逻辑运算）。 结果类型是左作数的类型。 赋值表达式的结果始终为 l 值。 这些运算符具有从右到左的关联性。 左作数必须是可修改的 l 值。

在 ANSI C 中，赋值表达式的结果不是 l 值。 这意味着 C 中不允许合法C++表达式 `(a += b) += c` 。

另请参阅

具有二进制运算符的表达式
C++内置运算符、优先级和关联性
C 赋值运算符

Last updated on 2025/11/07

按位 AND 运算符： `&`

语法

and-expression:

equality-expression

and-expression **&** *equality-expression*

注解

按位 AND 运算符 （`&`） 将第一个作数的每一位与第二个作数的对应位进行比较。 如果两个位均为 1，则对应的结果位将设置为 1。 否则，将对应的结果位设置为 0。

位 AND 运算符的两个作数必须具有整型类型。 标准转换中涵盖的常见算术转换将应用于作数。

运算符关键字 `&`

C++指定`bitand`为替代拼写。`&` 在 C 中，备用拼写作为 iso646.h> 标头中的<宏提供。 在 C++中，备用拼写是关键字; <已弃用 iso646.h> 或C++等效 <ciso646> 的使用。 在Microsoft C++中， /permissive- 启用备用拼写需要或 /Za 编译器选项。

示例：

C++

```
// expre_Bitwise_AND_Operator.cpp
// compile with: /EHsc
// Demonstrate bitwise AND
#include <iostream>
using namespace std;
int main() {
   unsigned short a = 0xCCCC;      // pattern 1100 ...
   unsigned short b = 0xAAAA;      // pattern 1010 ...

   cout  << hex << ( a & b ) << endl;   // prints "8888", pattern 1000 ...
}
```

另请参阅

C++内置运算符、优先级和关联性
C 位运算符

Last updated on 2025/11/07

按位异或运算符：`^`

语法

> *表达*`^`*表达*

注解

按位异或运算符 (`^`) 将其第一操作数的每个位与其第二操作数的相应位进行比较。 如果其中一个操作数中的位为 0，而另一个操作数中的位为 1，则相应的结果位设置为 1。 否则，将对应的结果位设置为 0。

该运算符的两个操作数必须为整型类型。 标准转换中所述的常用算术转换将应用于操作数。

有关在 C++/CLI 和 C++/CX 中交替使用 `^` 字符的详细信息，请参阅对象运算符 (^) 句柄（C++/CLI 和 C++/CX）。

^ 的运算符关键字

C++ 将 **`xor`** 指定为 `^` 的备选拼写。 在 C 中，备选拼写在 <iso646.h> 标头中作为宏提供。 在 C++ 中，备选拼写是关键字；已弃用 <iso646.h> 或 C++ 等效的 <ciso646>。 在 Microsoft C++ 中，需要 /permissive- 或 /Za 编译器选项才能启用备选拼写。

示例

C++

```
// 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 ...
}
```

另请参阅

C++ 内置运算符、优先级和关联性

Last updated on 2025/11/07

按位与或运算符：`|`

语法

> *expression1* `|` *expression2*

注解

按位"与或"运算符 (`|`) 将其第一操作数的每个位与第二操作数的相应位进行比较。 如果其中一个位是 1，则将对应的结果位设置为 1。 否则，将对应的结果位设置为 0。

该运算符的两个操作数必须为整型类型。 标准转换中所述的常用算术转换将应用于操作数。

`|` 的运算符关键字

C++ 将 `bitor` 指定为 `|` 的备选拼写。 在 C 中，备选拼写在 <iso646.h> 标头中作为宏提供。 在 C++ 中，备选拼写是关键字；已弃用 <iso646.h> 或 C++ 等效的 <ciso646>。 在 Microsoft C++ 中，需要 /permissive- 或 /Za 编译器选项才能启用备选拼写。

示例

C++

```
// 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 ...
}
```

另请参阅

C++ 内置运算符、优先级和关联性
C 按位运算符

Last updated on 2025/11/07

强制转换运算符： `()`

类型强制转换提供了一种方法，用于在特定情况下显式转换对象的类型。

语法

`cast-expression`:

`unary-expression`

`(` `type-name` `)` `cast-expression`

注解

任何一元表达式都被视为强制转换表达式。

编译器在 `cast-expression` 进行类型转换后视为类型 `type-name` 。 转换可用于将任何标量类型的对象转换为任何其他标量类型或从任何其他标量类型转换对象。 显式类型强制转换受确定隐式转换效果的相同规则的约束。 强制转换的其他限制可能是特定类型的实际大小或表示形式造成的。

例子

内置类型之间的标准强制转换：

C++

```
// 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;
}
```

在用户定义的类型中定义的强制转换运算符：

C++

```
// 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 = "Excitinggg";
    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);
}
```

Output

```
RawBytes truncated to 10 chars:   Exciting!!
Casted Bytes:   Exciting
Note that the cast changed the raw internal string
Raw Bytes after cast:   Exciting
```

另请参阅

具有一元运算符的表达式
C++内置运算符、优先级和关联性
显式类型转换运算符：()
强制转换运算符（C++）
强制转换运算符 （C）

Last updated on 2025/11/07

逗号运算符：`,`

允许对两个语句进行分组，其中有一个是预期的。

语法

```
expression , expression
```

备注

逗号运算符具有从左向右的关联性。 由逗号分隔的两个表达式将从左向右进行计算。 始终计算左操作数，并且在计算右操作数之前将完成所有副作用。

在某些上下文（如函数自变量列表）中，逗号可用作分隔符。 不要将该逗号用作分隔符与将其用作运算符的情况混淆；这两种用法完全不同。

请考虑表达式 `e1, e2`。 该表达式的类型和值是 e2 的类型和值；e1 的计算结果将被丢弃。 如果右操作数是左值，则结果为左值。

在通常将逗号用作分隔符的方案中（例如，在函数或聚合初始值设定项的自变量中），逗号运算符及其操作数必须包含在括号中。 例如：

C++

```
func_one( x, y + 2, z );
func_two( (x--, y + 2), z );
```

在上面的对 `func_one` 的函数调用中，会传递以逗号分隔的三个参数：`x`、`y + 2` 和 `z`。 在对 `func_two` 的函数调用中，圆括号强制编译器将第一个逗号解释为顺序计算运算符。 此函数调用将两个参数传递给 `func_two`。 第一个参数是顺序计算运算 `(x--, y + 2)` 的结果，具有表达式 `y + 2` 的值和类型；第二个参数为 `z`。

示例

C++

```
// 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);
}
```

Output

```
20
30
```

另请参阅

使用二元运算符的表达式
C++ 内置运算符、优先级和关联性
顺序评估运算符

Last updated on 2025/11/07

条件运算符： `?:`

语法

```
expression ? expression : expression
```

备注

条件运算符 (:) 是一个三元运算符（采用三个操作数）。 条件运算符按以下方式运行：

- 第一个操作数隐式转换为 `bool`。 计算该操作数，并在继续前完成所有副作用。
- 如果第一个操作数的计算结果为 `true` (1)，则计算第二个操作数。
- 如果第一个操作数的计算结果为 `false` (0)，则计算第三个操作数。

条件运算符的结果是操作数（无论是第二个还是第三个）的计算结果。 只有最后两个操作数之一在条件表达式中计算。

条件表达式具有从右到左的关联性。 第一个操作数必须是整数或指针类型。 以下规则适用于第二个和第三个操作数：

- 如果两个操作数是相同的类型，则结果也是该类型。
- 如果两个操作数都是算术或枚举类型，则执行常用算术转换（参见标准转换）来将它们转换为通用类型。
- 如果两个操作数都是指针类型，或者一个是指针类型，另一个是计算结果为 0 的常量表达式，则执行指针转换来将它们转换为通用类型。
- 如果两个操作数都是引用类型，则执行引用转换来将它们转换为通用类型。
- 如果两个操作数都是 void 类型，则通用类型是 void 类型。
- 如果两个操作数是相同的用户定义类型，则通用类型也是该类型。
- 如果操作数是不同的类型，而且至少有一个操作数是用户定义类型，则使用语言规则来确定通用类型。 （请参阅下面的警告。）

前面列表中没有的第二个和第三个操作数的任意组合都是非法的。 结果的类型是通用类型，如果第二个和第三个操作数是同一类型且都是左值，则结果为左值。

> ⚠ **警告**
>
> 如果第二个和第三个操作数的类型不相同，则会按 C++ 标准中的指定调用复杂类型转换规则。 这些转换可能会导致意外行为，包括构造和析构临时对象。 为此，我们强烈建议：(1) 避免将用户定义的类型用作带条件运算符的操作数；(2) 如果确实要使用用户定义的类型，务必将每个操作数显式转换为通用类型。

示例

C++

```cpp
// 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;
}
```

另请参阅

C++ 内置运算符、优先级和关联性
条件表达式运算符

Last updated on 2025/11/07

delete 运算符 (C++)

释放内存块。

语法

> [`::`] `delete` cast-expression
>
> [`::`] `delete []` cast-expression

备注

cast-expression 自变量必须是指向以前分配给使用 new 运算符创建的对象的内存块的指针。 `delete` 运算符的结果类型为 `void`，因此它不返回值。 例如：

```
C++
CDialog* MyDialog = new CDialog;
// use MyDialog
delete MyDialog;
```

对指向不使用 `delete` 分配的对象的指针使用 `new` 将产生不可预知的结果。 但是，可以对值为 0 的指针使用 `delete`。 此设置意味着，在失败时 `new` 返回 0 时，删除已失败的 `new` 操作的结果不会造成损害。 有关详细信息，请参阅 new 和 delete 运算符。

`new` 和 `delete` 运算符还可用于内置类型（包括数组）。 如果 `pointer` 指的是某一数组，请在 `[]` 前放置空括号（`pointer`）：

```
C++
int* set = new int[100];
//use set[]
delete [] set;
```

对对象使用 `delete` 运算符将释放其内存。 在删除对象后取消引用指针的程序可能会产生不可预知的结果或崩溃。

使用 `delete` 释放 C++ 类对象的内存时，将在释放该对象的内存之前调用该对象的析构函数（如果该对象具有析构函数）。

如果 `delete` 运算符的操作数是可修改的左值，则在删除该对象后未定义其值。

如果指定了 /sdl（启用其他安全检查）编译器选项，则删除对象后， `delete` 运算符的操作数设置为无效值。

使用 delete

delete 运算符有两个语法变体：一个针对单一对象，另一个针对对象数组。 以下代码片段演示了它们之间的差异：

C++

```
// expre_Using_delete.cpp
struct UDType
{
};

int main()
{
   // Allocate a user-defined object, UDObject, and an object
   //  of type double on the free store using the
   //  new operator.
   UDType *UDObject = new UDType;
   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.
   UDType (*UDArr)[7] = new UDType[5][7];
   // Use the array syntax to delete the array of objects.
   delete [] UDArr;
}
```

以下两种情况会生成未定义的结果：在对象中使用 delete（`delete []`）的数组形式，并在数组中使用 delete 的非数组形式。

示例

有关使用 **`delete`** 的示例，请参阅 new 运算符。

delete 的工作方式

delete 运算符将调用函数**运算符 delete**。

对于不是类类型（class、struct 或 union）的对象，将调用全局 delete 运算符。 对于类类型的对象，如果 delete 表达式以一元范围解析运算符（`::`）开始，则会在全局范围中解析解除分配函数的名称。 否则，delete 运算符将在释放内存之前为对象调用析构函数（如果指针不为 null）。 可为每个类定义 delete 运算符；如果给定类不存在这种定义，则会调用全局 delete 运算符。 如果删除表达式用于释放其静态对象具有虚拟析构函数的类对象，则将通过对象的动态类型的虚拟析构函数解析释放函数。

另请参阅

使用一元运算符的表达式
关键字
new 和 delete 运算符

Last updated on 2025/11/07

相等运算符：`==` 和 `!=`

语法

> *表达式* `==` *表达式*
> *表达式* `!=` *表达式*

注解

如果两个操作数的值相同，则等于运算符（`==`）返回 `true`；否则返回 `false`。
如果操作数的值不相同，则不等于运算符（`!=`）返回 `true`；否则返回 `false`。

在 C 和 C++ 中，`not_eq` 可用作 `!=` 的替代项。 有关详细信息，请参阅 not-eq。

示例

C++

```cpp
#include <iostream>

int main()
{
    int x = 1, y = 1, z = 2;

    if (x == y)
    {
        std::cout << "Equal\n";
    }

    if (x != z)
    {
        std::cout << "Not equal\n";
    }
}
```

输出

```
Equal
Not equal
```

另请参阅

not-eq
运算符重载
带二元运算符的表达式
C++ 内置运算符、优先级和关联性
C 关系和相等运算符

Last updated on 2025/11/07

显式类型转换运算符：`()`

C++ 允许使用与函数调用语法类似的语法进行显式类型转换。

语法

```
simple-type-name ( expression-list )
```

备注

后跟包含在括号中的 expression-list 的 simple-type-name 使用指定表达式构造指定类型的对象。 以下示例显示到类型 int 的显式类型转换：

```C++
int i = int( d );
```

以下示例显示了 `Point` 类。

示例

```C++
// 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();
}
```

输出

Output

```
x = 20, y = 10
x = 0, y = 0
```

尽管前面的示例演示了使用常量的显式类型转换，但在对对象执行转换时，此方法也同样有效。以下代码片段对此进行了演示：

C++

```
int i = 7;
float d;

d = float( i );
```

还可以使用“cast”语法指定显式类型转换。 使用 cast 语法重写的上一个示例是：

C++

```
d = (float)i;
```

当从单个值转换时，强制转换和函数样式转换都有相同的结果。 但是，在函数样式语法中，可以为转换指定多个自变量。 此差异对用户定义的类型非常重要。 请考虑 `Point` 类及其转换：

C++

```
struct Point
{
    Point( short x, short y ) { _x = x; _y = y; }
    ...
    short _x, _y;
};
...
Point pt = Point( 3, 10 );
```

前面使用函数样式转换的示例演示了如何将两个值（一个用于 x，另一个用于 y）转换为用户定义类型 `Point`。

> **⊗ 注意**
>
> 请谨慎使用显式类型转换，因其会重写 C++ 编译器的内置类型检查。

强制转换批注必须用于到没有 simple-type-name（例如，指针或引用类型）的类型的转换。 到可与 simple-type-name 一起表示的类型的转换可以使用任何一种形式写入。

在强制转换中的类型定义是非法的。

另请参阅

后缀表达式
C++ 内置运算符、优先级和关联性

Last updated on 2025/11/07

函数调用运算符： `()`

函数调用是由计算结果为函数或可调用对象后跟函数调用运算符`()`的表达式构成的一种 *`postfix-expression`*。 对象可以声明一个 `operator ()` 函数，该函数提供对象的函数调用语义。

语法

> *`postfix-expression`*:
>
> *`postfix-expression`* `(` *`argument-expression-list`* opt `)`

注解

函数调用运算符的参数来自 *`argument-expression-list`*一个逗号分隔的表达式列表。 这些表达式的值作为参数传递给函数。 *argument-expression-list* 可以为空。 在C++17之前，函数表达式和参数表达式的计算顺序未指定，并且可能按任何顺序发生。 在 C++17 及更高版本中，将在任何参数表达式或默认参数之前计算函数表达式。 参数表达式以不确定序列进行计算。

计算 *`postfix-expression`* 结果为要调用的函数。 它可以采用多种形式之一：

- 函数标识符，在当前范围或提供的任何函数参数的作用域中可见，
- 计算结果为函数、函数指针、可调用对象或对函数的引用的表达式，
- 成员函数访问器（显式或隐含）
- 指向成员函数的取消引用指针。

*`postfix-expression`*可以是重载的函数标识符或重载的成员函数访问器。 重载解析的规则决定了要调用的实际函数。 如果成员函数是虚拟的，则调用的函数在运行时确定。

一些示例声明：

- 函数返回类型 `T`。 示例声明为

  C++
  ```
  T func( int i );
  ```

- 指向返回类型的 `T`函数的指针。 示例声明为

  C++
  ```
  T (*func)( int i );
  ```

- 对返回类型的 `T` 函数的引用。 示例声明为

  C++
  ```
  T (&func)(int i);
  ```

- 指针到成员函数取消引用返回类型 `T`。 示例函数调用包括

  C++
  ```
  (pObject->*pmf)();
  (Object.*pmf)();
  ```

示例：

以下示例使用三个参数调用标准库函数 `strcat_s` ：

C++
```
// 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;
}
```

Output
```
Welcome to C++
```

函数调用结果

除非函数声明为引用类型，否则函数调用的计算结果为右值。 引用返回类型的函数的计算结果为左值。 这些函数可以在赋值语句的左侧使用，如下所示：

C++

```
// 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";
}
```

前面的代码定义一个调用 `Point` 的类，该类包含表示 *x* 和 *y* 坐标的私有数据对象。 必须修改这些数据对象及其检索值。 此程序只是此类的多个设计之一; `GetX` 使用和 `SetX` 函数 `GetY` `SetY` 是另一种可能的设计。

返回类类型、指向类类型的指针或对类类型的引用的函数可用作成员选择运算符的左作数。 以下代码是合法的：

C++

```
// 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 );
}
```

函数可以递归调用。 有关函数声明的详细信息，请参阅 Functions。 相关材料位于 翻译单元和链接中。

另请参阅

后缀表达式
C++内置运算符、优先级和关联性
函数调用

Last updated on 2025/11/07

间接运算符：`*`

语法

```
* cast-expression
```

备注

一元间接寻址运算符 (`*`) 取消引用指针；即它将指针值转换为一个 lvalue。 间接寻址运算符的操作数必须是指向类型的指针。 间接寻址表达式的结果是从中派生指针类型的类型。 此上下文中的 `*` 运算符的使用与它作为二元运算符的意义不同，后者是乘法运算符。

如果操作数指向函数，则结果是函数指示符。 如果它指向存储位置，则结果是指定存储位置的左值。

可以累计使用间接寻址运算符来取消引用指向指针的指针。 例如：

C++

```
// 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;
}
```

如果该指针的值无效，则结果是未定义的。 以下列表包含使指针值无效的一些最常见条件。

- 该指针为 null 指针。
- 该指针指定引用时不可见的本地项的地址。
- 该指针指定未针对所指向的对象类型正确对齐的地址。

- 该指针指定执行程序未使用的地址。

另请参阅

使用一元运算符的表达式
C++ 内置运算符、优先级和关联性
Address-of 运算符：&
间接寻址运算符和 Address-of 运算符

Last updated on 2025/11/07

左移和右移运算符： `<< >>`

按位移位运算符是右移运算符（`>>`），它将整数或枚举类型表达式的位向右移动，左移运算符（`<<`）将位移到左侧。 [1]

语法

shift-expression:

additive-expression

shift-expression `<<` *additive-expression*

shift-expression `>>` *additive-expression*

注解

> **重要**
>
> 对于 x86 和 x64 体系结构，以下说明和示例在 Windows 上有效。 对于 ARM 设备，在 Windows 上，左移和右移运算符的实现明显不同。 有关详细信息，请参阅 Hello ARM 博客文章的“Shift 运算符”部分。

左移

左移运算符使位 *shift-expression* 按指定的 *additive-expression* 位置数向左移。 移位作已空出的位位置是零填充的。 左移是一个逻辑移位（从末尾移出位的位将被丢弃，包括符号位）。 有关按位移位类型的详细信息，请参阅 按位移 位。

以下示例显示使用无符号数字的左移作。 该示例通过将值表示为位集来显示位点发生的情况。 有关详细信息，请参阅 bitset 类。

C++

```
#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
}
```

如果向左移一个有符号号码，使签名位受到影响，则结果未定义。 以下示例演示在将 1 位左移到符号位位置时会发生什么情况。

C++

```
#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
}
```

右移

右移运算符使位模式 `shift-expression` 按指定 `additive-expression`的位置数向右移动。 对于无符号数字，移位作已空出的位位置是零填充的。 对于带符号数字，符号位用于填充空位位置。 换句话说，如果数字为正数，则使用 0，如果数字为负数，则使用 1。

> ⓘ **重要**
>
> 有符号负数右移的结果取决于实现。 尽管Microsoft C++编译器使用符号位来填充空位位置，但不能保证其他实现也这样做。

此示例显示使用无符号号码的右移作：

C++

```cpp
#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
}
```

下一个示例显示带正符号数字的右移作。

C++

```cpp
#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
}
```

下一个示例显示带负带符号整数的右移作。

C++

```cpp
#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
}
```

班次和促销

Shift 运算符两侧的表达式必须是整型类型。 积分提升是根据 标准转换中所述的规则执行的。结果的类型与提升 `shift-expression` 的类型相同。

在下面的示例中，类型 `char` 变量提升为 `int`。

C++

```
#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
}
```

详细信息

如果为负数或`additive-expression`大于或等于 （提升） `shift-expression`中的位数，则未定义`additive-expression`班次运算的结果。 如果 `additive-expression` 为 0，则不执行 Shift作。

C++

```
#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

[1] 以下是 C++11 ISO 规范（INCITS/ISO/IEC 14882-2011[2012]）中的班次运算符的说明，第 5.8.2 节和 5.8.3 节。

左移`E2`位位置的值`E1 << E2` `E1`;空置位为零填充。 如果 `E1` 具有无符号类型，则结果的值为 **E1** × 2^{E2}，将模数减少一个大于结果类型中可表示的最大值。 否则，如果 `E1` 具有有符号类型和非负值，并且 **E1** × 2^{E2} 在结果类型的相应无符号类型中表示，则该值转换为结果类型是结果值;否则，行为是未定义的。

`E1 >> E2` `E1`值为右移`E2`位位置。 如果 `E1` 具有无符号类型或者 `E1` 具有有符号类型和非负值，则结果的值是 **E1**/2^{E2} 商的整部分。 如果 `E1` 具有有符号类型和负值，则生成的值为实现定义。

另请参阅

具有二进制运算符的表达式

C++内置运算符、优先级和关联性

Last updated on 2025/11/07

逻辑 AND 运算符： `&&`

语法

`logical-and-expression`:

`equality-expression`

`logical-and-expression` **`&&`** *`equality-expression`*

注解

如果两个作数为`true`并返回，则逻辑 AND 运算符 （`&&`） `true` 返回`false`。 作数在计算前隐式转换为类型 `bool` ，结果为类型 `bool`。 逻辑 AND 具有从左到右的关联性。

逻辑 AND 运算符的作数不需要具有相同类型，但它们必须具有布尔值、整型或指针类型。 作数通常是关系表达式或相等表达式。

第一个作数将完全计算，所有副作用在逻辑 AND 表达式的计算继续之前完成。

仅当第一个作数的计算结果为 `true` （非零） 时，才会计算第二个作数。 当逻辑 AND 表达式为`false`时，此计算消除了第二个作数的不必要计算。 可以使用此短路评估来防止 null 指针取消引用，如以下示例所示：

```
C++
char *pch = 0;
// ...
(pch) && (*pch = 'a');
```

如果 `pch` 为 null （0），则永远不会计算表达式的右侧。 此短路评估使分配无法通过 null 指针。

运算符关键字 `&&`

C++指定`and`为替代拼写。 `&&` 在 C 中，备用拼写作为 iso646.h> 标头中的<宏提供。 在 C++中，备用拼写是关键字; <已弃用 iso646.h> 或C++等效 <ciso646> 的使用。 在Microsoft C++中， /permissive- 启用备用拼写需要或 /Za 编译器选项。

示例：

```
C++
```

```cpp
// 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;
}
```

另请参阅

C++内置运算符、优先级和关联性
C 逻辑运算符

Last updated on 2025/11/07

逻辑非运算符： `!`

语法

> `!` 强制转换表达式

注解

逻辑非运算符 (`!`) 反转其操作数的含义。 操作数必须是算法或指针类型（或计算结果为算法或指针类型的表达式）。 操作数将隐式转换为类型 `bool`。 如果已转换的操作数是 `true`，则结果是 `false`；如果已转换的操作数是 `false`，则结果是 `true`。 结果的类型为 `bool`。

对于表达式 `e`，一元表达式 `!e` 与表达式 `(e == 0)` 等效，涉及重载运算符的情况除外。

! 的运算符关键字

C++ 将 `not` 指定为 `!` 的备选拼写。 在 C 中，备选拼写在 <iso646.h> 标头中作为宏提供。 在 C++ 中，备选拼写是关键字；已弃用 <iso646.h> 或 C++ 等效的 <ciso646>。 在 Microsoft C++ 中，需要 /permissive- 或 /Za 编译器选项才能启用备选拼写。

示例

C++

```cpp
// 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;
}
```

另请参阅

带一元运算符的表达式
C++ 内置运算符、优先级和关联性
一元算术运算符

Last updated on 2025/11/07

逻辑“或”运算符：`||`

语法

> logical-or-expression logical-and-expression `||`

注解

如果任一操作数或两个操作数为 `||`，则逻辑“或”运算符（`true`）返回布尔值 `true`；否则返回 `false`。 操作数在计算之前隐式转换为类型 `bool`，结果的类型为 `bool`。 逻辑“或”具有从左向右的关联性。

逻辑“或”运算符的操作数不需要具有相同的类型，但它们必须是布尔值、整数或指针类型。 操作数通常为关系或相等表达式。

第一个操作数将完全计算，并且在继续计算逻辑“或”表达式之前将完成所有副作用。

仅当第一个操作数的计算结果为 `false` 时计算第二个操作数，因为当逻辑“或”表达式为 `true` 时不需要计算。 这称作*“短路”*计算。

C++

```
printf( "%d" , (x == w || x == y || x == z) );
```

在上面的示例中，如果 `x` 与 `w`、`y` 或 `z` 相等，则 `printf` 函数的第二个自变量的计算结果为 `true`（之后增加至整数），并打印值 1。 否则，它的计算结果将为 `false`，并打印值 0。 只要其中一个条件的计算结果为 `true`，计算便会停止。

`||` 的运算符关键字

C++ 将 `or` 指定为 `||` 的备选拼写。 在 C 中，备选拼写在 <iso646.h> 标头中作为宏提供。 在 C++ 中，备选拼写是关键字；已弃用 <iso646.h> 或 C++ 等效的 <ciso646>。 在 Microsoft C++ 中，需要 /permissive- 或 /Za 编译器选项才能启用备选拼写。

示例

C++

```
// 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;
}
```

另请参阅

C++ 内置运算符、优先级和关联性
C 逻辑运算符

Last updated on 2025/11/07

成员访问运算符： `.` 和 `->`

语法

postfix-expression?

postfix-expression **.** **template** 选择 *id-expression*

postfix-expression **->** **template** 选择 *id-expression*

注解

成员访问运算符 `.` 和 `->` 用来引用 `struct`、 `union` 和 `class` 类型的成员。 成员访问表达式具有选定成员的值和类型。

有两种形式的成员访问表达式：

1. 在第一种形式中， *postfix-expression* 表示 `struct`、 `class` 或 `union` 类型的值，而 *id-expression* 为指定的 `struct`、 `union` 或 `class` 的成员命名。 运算的值是 *id-expression* 的值且是一个左值（如果 *postfix-expression* 是左值）。

2. 在第二种形式中， *postfix-expression* 表示指向 `struct`、 `union` 或 `class` 的指针，而 *id-expression* 为指定的 `struct`、 `union` 或 `class` 的成员命名。 该值是 `id-expression` 的值且是左值。 `->` 运算符取消引用该指针。 表达式 `e->member` 和 `(*(e)).member` （其中， *e* 表示指针）会产生相同的结果（重载运算符 `->` 或 `*` 时除外）。

示例

以下示例演示成员访问运算符的两种形式。

C++

```
// 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;
}
```

Output

```
2/1/1900
2/1/2000
```

另请参阅

后缀表达式
C++ 内置运算符、优先级和关联性
类和结构
结构和联合成员

Last updated on 2025/11/07

乘法运算符和取模运算符

语法

```
expression * expression
expression / expression
expression % expression
```

备注

乘法运算符为：

- 乘法 (`*`)
- 除法 (`/`)
- 取模（除法运算的余数）(`%`)

这些二进制运算符具有从左至右的关联性。

乘法运算符采用算术类型的操作数。 取模运算符 (%) 具有更严格的要求，即其操作数必须是整型。 （若要获取浮点除法的余数，请使用运行时函数 fmod。）标准转换中涵盖的转换应用于操作数，结果是转换后的类型。

除法运算符产生的结果为将第一个操作数乘以第二个操作数所获得的结果。

除法运算符产生的结果为将第一个操作数除以第二个操作数所获得的结果。

取模运算符产生下列表达式给定的余数，其中 e1 是第一个操作数，e2 是第二个操作数：e1 - (e1e2) * e2，其中两个操作数都是整型类型。 /

在除法或取模表达式中被 0 除的结果是不确定的，将会导致运行时错误。 因此，以下表达式生成未定义的错误结果：

```
C++
i % 0
f / 0.0
```

如果乘法、除法或取模表达式的两个操作数具有相同的符号，则结果为正。 否则，结果为负。 取模运算的符号的结果是实现定义的。

> ⓘ **注意**
>
> 由于在溢出或下溢条件不提供由乘法运算符执行的转换，因此，如果乘法操作的结果在转换后不能用操作数类型表示，则信息可能丢失。

Microsoft 专用

在 Microsoft C++ 中，取模表达式的结果的符号始终与第一个操作数的符号相同。

结束 Microsoft 专用

如果两个整数的减法计算不准确，并且只有一个操作数为负，则结果是最大的整数（在数量级上，忽略符号），该整数小于减法运算所生成的准确值。 例如，-11 / 3 的计算值为 -3.666666666。 整除的结果为 -3。

乘法运算符间的关系由标识 (e1e2) * e2 / e1e2e1 给定。 + % ==

示例

以下程序演示乘法运算符。 请注意，`10 / 3` 的任一操作数必须显式转换为类型 `float` 以避免截断，以便两个操作数的类型在除法运算前为 `float`。

C++

```
// 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;
}
```

另请参阅

使用二元运算符的表达式
C++ 内置运算符、优先级和关联性
C 乘法运算符

Last updated on 2025/11/07

`new` 运算符 （C++）

尝试分配和初始化指定或占位符类型的对象的对象或数组，并返回对象（或数组的初始对象）的适当类型化非零指针。

语法

`new-expression`:

`::` 选择 `new` `new-placement` 选择 `new-type-id` `new-initializer` 选择

`::` 选择 `new` `new-placement` 选择 `(` `type-id` `)` `new-initializer` 选择

`new-placement`:

`(` `expression-list` `)`

`new-type-id`:

`type-specifier-seq` `new-declarator` 选择

`new-declarator`:

`ptr-operator` `new-declarator` 选择

`noptr-new-declarator`

`noptr-new-declarator`:

`[` `expression` `]` `attribute-specifier-seq` 选择

`noptr-new-declarator` `[` `constant-expression` `]` `attribute-specifier-seq` 选择

`new-initializer`:

`(` `expression-list` 选择 `)`

`braced-init-list`

注解

如果失败， `new` 则返回零或引发异常。 有关详细信息，请参阅 " new 和 delete 运算符"。 可以通过编写自定义异常处理例程，并使用函数名称作为参数调用 _set_new_handler 运行时库函数来更改此默认行为。

有关如何在 C++/CLI 和 C++/CX 中的托管堆上创建对象的信息，请参阅 gcnew。

> **注意**

Microsoft C++组件扩展（C++/CX）支持 `new` 添加 vtable 槽项的关键字。 有关详细信息，请参阅 new （vtable 中的新槽）

当用于为C++类对象分配内存时 `new` ，在分配内存后调用对象的构造函数。

delete使用运算符解除分配由该运算符分配的`new`内存。 使用 `delete[]` 运算符删除由该运算符分配的 `new` 数组。

以下示例分配并释放大小 `dim` 为 10 的二维字符数组。 分配多维数组时，除第一个维度之外的所有维度必须是计算结果为正值的常量表达式。 最左侧的数组维度可以是计算结果为正值的任何表达式。 使用 `new` 运算符分配数组时，第一个维度可以为零; `new` 运算符返回唯一指针。

C++

```
char (*pchar)[10] = new char[dim][10];
delete [] pchar;
```

*`type-id`*不能包含`const`、 `volatile`类声明或枚举声明。 以下表达式格式不正确：

C++

```
volatile char *vch = new volatile char[20];
```

`new`运算符不会分配引用类型，因为它们不是对象。

`new`运算符不能用于分配函数，但可用于向函数分配指针。 以下示例分配并释放一个由七个指针构成的数组，这些指针指向返回整数的函数。

C++

```
int (**p) () = new (int (*[7]) ());
delete p;
```

如果使用不带任何额外参数的运算符`new`，并使用或/EHa/EHs选项进行编译/GX，则编译器在构造函数引发异常时生成调用运算符`delete`的代码。

以下列表描述了以下语法 `new`元素：

`new-placement`
提供一种在重载 `new`时传递额外参数的方法。

`type-id`
指定要分配的类型;它可以是内置类型或用户定义的类型。 如果类型规范很复杂，可以用括号括起来以强制绑定顺序。 该类型可以是一个占位符（`auto`），其类型由编译器确定。

`new-initializer`
为初始化的对象提供值。 不能为数组指定初始值设定项。 仅当类具有默认构造函数时，该 `new` 运算符才会创建对象的数组。

`noptr-new-declarator`
指定数组的边界。 分配多维数组时，除第一个维度外的所有维度必须是计算结果 `std::size_t` 为可转换为正值的常量表达式。 最左侧的数组维度可以是计算结果为正值的任何表达式。 应用于 `attribute-specifier-seq` 关联的数组类型。

示例：分配和释放字符数组

下面的代码示例分配字符数组和类 `CName` 的对象，然后释放它们。

C++

```
// 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;
}
```

示例： `new` 运算符

如果使用运算符的 `new` 放置形式（具有比大小更多的参数的窗体），则如果构造函数引发异常，编译器不支持运算符的 `delete` 放置形式。 例如：

C++

```
// 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 can't occur.

      A* pa2 = new(__FILE__, __LINE__) A(20);
   } catch (...) {
   }
}

int main() {
   A a;
}
```

初始化使用 分配的对象

运算符的语法`new`中包含可选*new-initializer*字段。 此字段允许使用用户定义的构造函数初始化新对象。 有关如何完成初始化的详细信息，请参阅 初始值设定项。 以下示例演示如何对运算符使用初始化表达式 `new` ：

C++

```
// 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 };
    // ...
}
```

在此示例中，使用运算符分配 `new` 对象 `CheckingAcct`，但未指定默认初始化。 因此，将调用类 `Acct()` 的默认构造函数。 然后，以相同的方式分配对象，只不过该对象 `SavingsAcct` 显式初始化为 34.98。 由于 34.98 的类型， `double` 因此调用采用该类型的参数的构造函数来处理初始化。 最后，非类类型 `HowMuch` 初始化为 43.0。

如果对象是类类型且该类具有构造函数（如前面的示例中），则仅当满足以下条件之一时，该对象才能由 `new` 运算符初始化：

- 初始值设定项中提供的参数与构造函数的参数匹配。
- 该类具有一个默认构造函数（一个无需参数调用的构造函数）。

使用 `new` 运算符分配数组时，无法执行显式每元素初始化;仅调用默认构造函数（如果存在）。 有关详细信息，请参阅 默认参数。

如果内存分配失败（`operator new` 返回值 0），则不执行初始化。 此行为可防止尝试初始化不存在的数据。

与函数调用一样，未定义初始化表达式的计算顺序。 此外，不应依赖在内存分配发生之前完全计算这些表达式。 如果内存分配失败， `new` 并且运算符返回零，则初始值设定项中的某些表达式可能无法完全计算。

使用

当定义对象的范围退出时，不会销毁使用 `new` 运算符分配的对象。 `new` 由于运算符返回指向它分配的对象的指针，因此程序必须定义一个具有适当作用域的指针来访问和删除这些对象。 例如：

C++

```
// 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.
}
```

在示例中指针 `AnotherArray` 超出范围后，无法再删除该对象。

`new` 的工作原理

`new-expression` （包含运算符的 `new` 表达式）执行三项作：

- 查找并保留要分配的对象或对象的存储。 完成此阶段后，将分配正确的存储量，但尚不是对象。
- 初始化对象（s）。 初始化完成后，将为分配的存储提供足够的信息作为对象。
- 返回指向派生自 *`new-type-id`* 或 *`type-id`* 的指针类型的对象的指针的指针。 程序使用此指针访问新分配的对象。

运算符 `new` 调用函数 `operator new`。 对于任何类型的数组，对于不是 `class` `struct`、或 `union` 类型的对象， `::operator new` 将调用全局函数来分配存储。 类类型对象可以基于每个类定义自己的 `operator new` 静态成员函数。

当编译器遇到 `new` 运算符来分配类型的 `T` 对象时，它会调用 `T::operator new( sizeof(T) )` 或未定义 `::operator new( sizeof(T) )` 用户定义的 `operator new` 对象。 操作员如何 `new` 为对象分配正

确的内存量。

> ⓘ **注意**
>
> 要的参数 `operator new` 的类型 `std::size_t`。 此类型在 direct.h、malloc.h>、<memory.h>、search.h>、<stddef.h、<<stdio.h、<stdlib.h>>>、<string.h 和 <time.h>> 中定义。 <><

语法中的选项允许指定 *`new-placement`* （请参阅运算符语法new）。 参数 *`new-placement`* 只能用于用户定义的实现 `operator new`;它允许将额外的信息传递给 `operator new`。 具有字段的表达式，例如，如果类 T 具有成员，则转换为 ;否则为 < a1/>。

字段的 *`new-placement`* 最初意图是允许在用户指定的地址分配依赖于硬件的对象。

> ⓘ **注意**
>
> 虽然前面的示例只显示字段中的一个参数 *`new-placement`* ，但对可以传递给 `operator new` 这种方式的额外参数数没有限制。

`operator new`即使已为类类型`T`定义，也可以显式使用全局运算符`new`，如以下示例所示：

```C++
T *TObject = ::new TObject;
```

范围解析运算符 （`::`） 强制使用全局 `new` 运算符。

另请参阅

具有一元运算符的表达式
关键字
new 和 delete 运算符

Last updated on 2025/11/07

二进制求补运算符：`~`

语法

```
C++
~ cast-expression
```

备注

二进制反码运算符（`~`）（有时称为“按位反码”运算符）将生成其操作数的按位二进制反码。 即，操作数中为 1 的每个位在结果中为 0。 相反，操作数中为 0 的每个位在结果中为 1。 二进制反码运算符的操作数必须为整型。

~ 的运算符关键字

C++ 将 `compl` 指定为 `~` 的备选拼写。 在 C 中，备选拼写在 <iso646.h> 标头中作为宏提供。 在 C++ 中，备选拼写是关键字；已弃用 <iso646.h> 或 C++ 等效的 <ciso646>。 在 Microsoft C++ 中，需要 /permissive- 或 /Za 编译器选项才能启用备选拼写。

示例

```
C++
// 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;
}
```

在此示例中，分配给 `y` 的新值是无符号值 0xFFFF 或 0x0000 的二进制反码。

整型提升是对整型操作数执行的。 操作数提升到的类型是结果类型。 有关整型提升的详细信息，请参阅标准转换。

另请参阅

带一元运算符的表达式
C++ 内置运算符、优先级和关联性
一元算术运算符

Last updated on 2025/11/07

指针到成员运算符： `.* ->*`

语法

pm-expression:

cast-expression

pm-expression `.*` *cast-expression*

pm-expression `->*` *cast-expression*

注解

指针到成员运算符 `.*` ，并为 `->*` 表达式左侧指定的对象返回特定类成员的值。 右侧必须指定类的成员。 以下示例演示如何使用这些运算符：

C++

```
// 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;
}
```

输出

Output

```
m_func1
m_func1
1
2
```

在前面的示例中，指向成员 `pmfn` 的指针用于调用成员函数 `m_func1`。 指向成员 `pmd` 的另一个指针用于访问 `m_num` 该成员。

二进制运算符 `.*` 将其第一个作数（必须是类类型的对象）与第二个作数（必须是指针到成员类型）组合在一起。

二进制运算符 `->*` 将第一个作数（必须是指向类类型的对象的指针）与第二个作数（必须是指针到成员类型）组合在一起。

在包含运算符的 `.*` 表达式中，第一个作数必须是类类型的指针，并且必须可供访问，指向第二个作数中指定的成员的指针或可访问的类型明确派生自该类并可供该类访问。

在包含运算符的 `->*` 表达式中，第一个作数必须是第二个作数中指定的类型的“指向类类型的指针”的类型，或者它必须明确派生自该类的类型。

示例：

请考虑以下类和程序片段：

C++

```cpp
// 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.
}
```

或`->*`指针到成员运算符的结果`.*`是指向成员的指针声明中指定的类型的对象或函数。 因此，在前面的示例中，表达式 `ADerived.*pmfnFunc1()` 的结果是指向返回 `void`的函数的指针。 如果第二个作数为 l 值，则此结果是 l 值。

> **注意**
>
> 如果指针到成员运算符之一的结果是函数，则结果只能用作函数调用运算符的作数。

另请参阅

C++内置运算符、优先级和关联性

Last updated on 2025/11/07

后缀增量和减量运算符：`++` 和 `--`

语法

```
postfix-expression ++
postfix-expression --
```

备注

C++ 提供了前缀和后缀递增和递减运算符；本节仅介绍后缀递增和递减运算符。 有关详细信息，请参阅 前缀增量和递减运算符。 两者之间的差异在于，在后缀表示法中，运算符显示在 *后缀表达式*之后，而在前缀表示法中，运算符显示在 *表达式之前*。 以下示例演示后缀递增运算符：

```C++
i++;
```

应用后缀递增运算符 (`++`) 的效果是操作数的值增加一个适当类型的单位。 同样，应用后缀递减运算符 (`--`) 的效果是操作数的值减少一个适当类型的单元。

值得注意的是，后缀递增或递减表达式的计算结果为应用各自的运算符之前的表达式的值。 递增或递减运算在计算操作数之后发生。 仅当在较大的表达式的上下文中发生后缀递增或递减运算时才会出现此问题。

当后缀运算符应用于函数参数时，在参数的值传递给函数之前，不能保证该值是递增还是递减。 有关详细信息，请参阅 C++ 标准中的 1.9.17 节。

将后缀递增运算符应用于指向类型 `long` 的对象数组的指针实际上会将指针的内部表示形式增加 4。 此行为会导致以前引用数组的第 n 个元素的指针引用第 (n+1) 个元素。

后缀递增运算符和后缀递减运算符的操作数必须是算术或指针类型的可修改的（非 `const`）左值。 结果的类型与 postfix-expression 的类型相同，但不再是左值。

Visual Studio 2017 版本 15.3 及更高版本（在 模式和更高版本中可用）： 后缀递增或递减运算符的操作数可能不是 /std:c++17 类型。 `bool`

以下代码演示了后缀递增运算符：

```C++
```

```
// 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;
}
```

Output

```
10
11
```

不支持对枚举类型执行后递增和后递减操作：

C++

```
enum Compass { North, South, East, West };
Compass myCompass;
for (myCompass = North; myCompass != West; myCompass++); // Error
```

另请参阅

后缀表达式
C++ 内置运算符、优先级和关联性
C 后缀增量和减量运算符

Last updated on 2025/11/07

前缀增量和减量运算符：`++` 和 `--`

语法

```
++ unary-expression
-- unary-expression
```

备注

前缀递增运算符 (`++`) 向其操作数增加 1；此递增值是表达式的结果。 操作数必须是类型不为 `const` 的 l-value。 结果是与操作数相同类型的左值。

前缀递减运算符 (`--`) 与前缀递增运算符类似，只不过操作数将减少 1，并且结果是递减值。

Visual Studio 2017 版本 15.3 及更高版本（在 模式和更高版本中可用）：递增或递减运算符的操作数可能不是 /std:c++17 类型。 `bool`

前缀和后缀递增和递减运算符均会影响其操作数。 它们之间的主要差异是递增或递减在表达式的计算中出现的顺序。 有关详细信息，请参阅 后缀增量和递减运算符。 在前缀形式中，增量或递减发生在表达式计算中使用该值之前，因此表达式的值与作数的值不同。 在后缀形式中，将在表达式计算中使用值之后进行递增或递减，因此表达式的值与操作数的值相同。 例如，以下程序将打印“`++i = 6`”：

C++

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

using namespace std;

int main() {
   int i = 5;
   cout << "++i = " << ++i << endl;
}
```

整型或浮动类型的操作数将按整数值 1 递增或递减。 结果的类型与操作数类型相同。 指针类型的操作数将按其所寻址对象的大小递增或递减。 递增的指针将指向下一个对象；递减的指针将指向上一个对象。

由于增量和减量运算符具有副作用，因此在预处理器宏中使用带递增或递减运算符的表达式时会产生意外结果。 请看以下示例：

C++

```cpp
// 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 );
}
```

宏将扩展为：

C++

```cpp
k = ((++i)<(j))?(j):(++i);
```

如果 `i` 大于或等于 `j` 或者比 `j` 小 1，则将递增两次。

> **注意**
>
> 由于 C++ 内联函数会消除副作用（如此处描述的副作用），并允许语言执行更全面的类型检查，因此在很多情况下 C++ 内联函数较宏更为可取。

另请参阅

使用一元运算符的表达式
C++ 内置运算符、优先级和关联性
前缀增量和减量运算符

Last updated on 2025/11/07

关系运算符：`<`、`>`、`<=` 和 `>=`

语法

```
expression < expression
expression > expression
expression <= expression
expression >= expression
```

备注

二进制关系运算符确定下列关系：

- 小于号 (`<`)
- 大于号 (`>`)
- 小于或等于 (`<=`)
- 大于或等于 (`>=`)

关系运算符具有从左到右的关联性。 关系运算符的两个操作数必须是算术或指针类型。 它们将生成 `bool` 类型的值。 如果表达式中的关系为 false，则返回的值为 `false` (0)；否则返回的值为 `true` (1)。

示例

C++

```cpp
// 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;
}
```

前面的示例中的表达式必须括在括号中，因为流插入运算符（`<<`）的优先级高于关系运算符。 因此，未括在括号中的第一个表达式的计算结果将为：

```
C++
(cout << "The true expression 3 > 2 yields: " << 3) < (2 << "\n");
```

标准转换中介绍的常用算术转换将应用于算术类型的操作数。

比较指针

在比较两个指向同一类型的对象的指针时，结果由程序的地址空间中所指向的对象的位置决定。 也可以将指针与计算结果为 0 的常数表达式或 `void *` 类型的指针进行比较。 如果针对 `void *` 类型的指针执行指针比较，则另一个指针将隐式转换为 `void *` 类型。 然后进行比较。

不能比较两个类型不同的指针，除非：

- 一个类型是派生自另一个类型的类类型。
- 至少有一个指针显式转换（强制转换）为类型 `void *`。 （对于该转换，另一个指针将隐式转换为类型 `void *`。）

两个指向同一对象的相同类型的指针一定是相等的。 如果比较两个指向对象的非静态成员的指针，则以下规则将适用：

- 如果类类型不是 `union`，并且如果两个成员未通过 access-specifier（例如，`public`、`protected` 或 `private`）分隔开，则指向最后声明的成员的指针将大于指向之前声明的成员的指针。
- 如果两个成员通过 access-specifier 分隔开，则结果是不确定的。
- 如果类类型是 `union`，则指向该 `union` 中不同的数据成员的指针是相等的。

如果两个指针指向同一数组的元素或指向超出数组末尾 1 的元素，则指向带较高下标的对象的指针会更高。 仅当指针引用同一数组中的对象或超出数组末尾 1 的位置时，才能保证指针比较有效。

另请参阅

使用二元运算符的表达式
C++ 内置运算符、优先级和关联性
C 关系和相等运算符

Last updated on 2025/11/07

范围解析运算符： `::`

范围解析运算符 `::` 用于识别和消除不同作用域中使用的标识符。 有关范围的详细信息，请参阅 范围。

语法

> *`qualified-id`*:
>
> *`nested-name-specifier`* **`template`** 选择 *`unqualified-id`*

> *`nested-name-specifier`*:
>
> `::`
>
> *`type-name`* `::`
>
> *`namespace-name`* `::`
>
> *`decltype-specifier`* `::`
>
> *`nested-name-specifier`* *`identifier`* `::`
>
> *`nested-name-specifier`* **`template`** 选择 *`simple-template-id`* `::`

> *`unqualified-id`*:
>
> *`identifier`*
>
> *`operator-function-id`*
>
> *`conversion-function-id`*
>
> *`literal-operator-id`*
>
> `~` *`type-name`*
>
> `~` *`decltype-specifier`*
>
> *`template-id`*

注解

`identifier` 可以是变量、函数或枚举值。

用于 `::` 类和命名空间

以下示例演示如何将范围解析运算符与命名空间和类一起使用：

```
C++
```

```
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;
}
```

没有范围限定符的作用域解析运算符引用全局命名空间。

C++

```
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;
}
```

可以使用范围解析运算符来标识成员， `namespace` 或标识在指令中 `using` 指定成员命名空间的命名空间。 在下面的示例中，你可以使用 `NamespaceC` 它来限定 `ClassB`，即使 `ClassB` 在命名空间 `NamespaceB` 中声明，因为 `NamespaceB` 是由指令提名的 `NamespaceC` `using`。

C++

```
namespace NamespaceB {
    class ClassB {
    public:
        int x;
    };
```

```
}

namespace NamespaceC{
    using namespace NamespaceB;
}

int main() {
    NamespaceB::ClassB b_b;
    NamespaceC::ClassB c_b;

    b_b.x = 3;
    c_b.x = 4;
}
```

可以使用范围解析运算符链。 在以下示例中， `NamespaceD::NamespaceD1` 标识嵌套命名空间 `NamespaceD1`，并 `NamespaceE::ClassE::ClassE1` 标识嵌套类 `ClassE1`。

C++

```
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  ;
}
```

用于 `::` 静态成员

必须使用范围解析运算符来调用类的静态成员。

C++

```
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();
}
```

用于 `::` 限定范围的枚举

作用域解析运算符还与作用域 枚举枚举声明的值一起使用，如以下示例所示：

C++

```
enum class EnumA{
    First,
    Second,
    Third
};

int main() {
    EnumA enum_value = EnumA::First;
}
```

另请参阅

C++内置运算符、优先级和关联性
命名空间

Last updated on 2025/11/07

`sizeof` 运算符

产生与 `char` 类型的大小有关的操作数大小。

> ⓘ **注意**
>
> 有关 `sizeof ...` 运算符的信息，请参阅**省略号和可变参数模板**。

语法

```
sizeof unary-expression
sizeof ( type-name )
```

备注

`sizeof` 运算符的结果是 `size_t` 类型，它是包含文件 <stddef.h> 中定义的整数类型。 利用此运算符，你可以避免在程序中指定依赖于计算机的数据大小。

`sizeof` 的操作数可以是下列项之一：

- 类型名称。 若要将 `sizeof` 用于类型名称，则该名称必须用括号括起。
- 一个表达式。 与表达式一起使用时，可以使用或不使用括号指定 `sizeof`。 不计算表达式。

当 `sizeof` 运算符应用到 `char` 类型的对象时，它将生成 1。 当 `sizeof` 运算符应用到数组时，它将产生该数组的字节总数，而非由数组标识符表示的指针的大小。 若要获取由数组标识符表示的指针的大小，请将它作为参数传递给使用 `sizeof` 的函数。 例如：

示例

C++

```
#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;
}
```

示例输出

Output

```
The size of a char is: 1
The length of Hello, world! is: 14
The size of the pointer is 4
```

当 `sizeof` 运算符应用于 `class`、`struct` 或 `union` 类型时，结果是该类型对象中的字节数，加上为对齐字边界上的成员而添加的任何填充。 结果不一定对应于通过将各个成员的存储需求相加计算出的大小。 /Zp 编译器选项和 pack pragma 会影响成员的对齐边界。

`sizeof` 运算符永远不会产生 0，即使对于空类也是如此。

`sizeof` 运算符不能用于以下操作数：

- 函数。 （但是，`sizeof` 可应用于指向函数的指针。）
- 位域。
- 未定义的类。
- `void` 类型。
- 动态分配的数组。
- 外部数组。
- 不完整类型。
- 带括号的不完整类型的名称。

将 `sizeof` 运算符应用于引用时，结果与将 `sizeof` 应用于对象本身一样。

如果某个未确定大小的数组是结构的最后一个元素，则 `sizeof` 运算符将返回没有该数组的结构的大小。

`sizeof` 运算符通常用于使用以下形式的表达式计算数组中的元素数：

```
C++
sizeof array / sizeof array[0]
```

另请参阅

使用一元运算符的表达式
关键字

Last updated on 2025/11/07

下标运算符：`[]`

语法

```
postfix-expression [ expression ]
```

备注

后跟下标运算符 `[ ]` 的后缀表达式（也可为主表达式）指定数组索引。

有关 C++/CLI 中托管数组的详细信息，请参阅数组。

通常，postfix-expression 表示的值是一个指针值（如数组标识符），expression 是一个整数值（包括枚举类型）。 但是，从语法上来说，只需要一个表达式是指针类型，另一个表达式是整型。 因此整数值可以位于 postfix-expression 位置，指针值可以位于 expression 的方括号中或下标位置。 可考虑使用以下 代码段：

C++

```
int nArray[5] = { 0, 1, 2, 3, 4 };
cout << nArray[2] << endl;            // prints "2"
cout << 2[nArray] << endl;            // prints "2"
```

在前面的示例中，表达式 `nArray[2]` 与 `2[nArray]` 相同。 原因是下标表达式 `e1[e2]` 的结果由以下所示给定：

`*((e2) + (e1))`

该表达式生成的地址不是 e1 地址中的 e2 字节。 相反，该地址将进行缩放以生成数组 e2 中的下一个对象。 例如：

C++

```
double aDbl[2];
```

`aDb[0]` 和 `aDb[1]` 的地址相距 8 字节，`double` 类型的对象的大小。 根据对象类型进行的缩放将由 C++ 语言自动完成，并在其中讨论了指针类型的操作数的加减法的相加运算符中定义。

下标表达式还可以有多个下标，如下所示：

expression1[*expression2*] [*expression3*] ...

下标表达式从左至右关联。 首先计算最左侧的下标表达式 expression1 [expression2]。 通过添加 expression1 和 expression2 得到的地址构成一个指针表达式；然后 expression3 将添加到此指针表达式，从而构成一个新的指针表达式，依此类推，直到添加最后一个下标表达式。 在计算了最后的 subscripted 表达式后，将应用间接寻址运算符 (`*`)，除非最终指针值将为数组类型寻址。

具有多个下标的表达式引用多维数组的元素。 多维数组是其元素为数组的数组。 例如，三维数组的第一个元素是一个具有两个维度的数组。 以下示例声明并初始化字符的简单二维数组：

C++

```
// 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;
}
```

正下标和负下标

数组的第一个元素是元素 0。 C++ 数组的范围是从 array[0] 到 array[size – 1]。 但是，C++ 支持正负下标。 负下标必须在数组边界内；否则结果不可预知。 以下代码显示了正数组和负数组下标：

C++

```
#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
}
```

上一行中的负下标可能产生运行时错误，因为它在内存中指向比数组的原点低 256 `int` 的地址。 指针 `midArray` 会初始化为 `intArray` 的中点；因此可以对其使用正数组和负数组索引（但较危险）。 数组下标错误不会产生编译时错误，但它们会产生不可预知的结果。

下标运算符是可交换的。 因此，只要没有重载下标运算符（请参阅*重载运算符*），表达式 array[index] 和 index[array] 就一定等效。 第一种形式是最常见的编码做法，但它们都有效。

另请参阅

后缀表达式
C++ 内置运算符、优先级和关联性
数组
一维数组
多维数组

Last updated on 2025/11/07

`typeid` 运算符

语法

```
typeid(type-id)
typeid(expression)
```

备注

`typeid` 运算符允许在运行时确定对象的类型。

`typeid` 的结果是 `const type_info&`。 该值是对表示 type-id `type_info` 或 expression 的类型的对象的引用，具体取决于所使用的 `typeid` 的形式。 有关详细信息，请参阅 type_info 类。

`typeid` 运算符不适用于托管类型（抽象声明符或实例）。 有关获取指定类型的 Type 的信息，请参阅 typeid。

`typeid` 运算符在应用于多态类类型的 lvalue 时执行运行时检查，其中对象的实际类型不能由提供的静态信息确定。 此类情况是：

- 对类的引用
- 使用 `*` 取消引用的指针
- 带下标的指针（`[ ]`）。 （将下标与指向多态类型的指针一起使用是不安全的。）

如果 expression 指向基类类型，但该对象实际上是派生自该基类的类型，则派生类的 `type_info` 引用是结果。 expression 必须指向多态类型（具有虚函数的类）。 否则，结果是 expression `type_info` 中引用的静态类的 。 此外，必须取消引用指针，这样使用的对象就是它所指向的对象。 如果不取消引用指针，结果将是指针的 `type_info`，而不是它指向的内容。 例如：

C++

```cpp
// expre_typeid_Operator.cpp
// compile with: /GR /EHsc
#include <iostream>
#include <typeinfo>

class Base {
public:
   virtual void vvfunc() {}
```

```
};

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;
}
```

如果 expression 正在取消引用某个指针，并且该指针的值是零，`typeid` 将引发 bad_typeid 异常。 如果指针不指向有效的对象，则会引发 `__non_rtti_object` 异常。 它指示试图分析触发了错误的 RTTI，因为该对象在某种程度上是无效的。 （例如，它是一个错误的指针，或者代码不是使用 /GR 编译的。）

如果 expression 既不是指针，也不是对对象的基类的引用，则结果是表示 expression `type_info` 的静态类型的 引用。 表达式的 static type 将引用在编译时已知的表达式的类型。 在计算表达式的静态类型时，将忽略执行语义。 此外，在确定表达式的静态类型时，将忽略引用（如果可能）：

C++

```
// expre_typeid_Operator_2.cpp
#include <typeinfo>

int main()
{
   typeid(int) == typeid(int&); // evaluates to true
}
```

`typeid` 也可以在模板中使用，以确定模板参数的类型：

C++

```
// 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 );
}
```

另请参阅

运行时类型信息
关键字

Last updated on 2025/11/07

一元加号和非运算符：`+` 和 `-`

语法

```
+ cast-expression
- cast-expression
```

`+` 运算符

一元加运算符（`+`）的结果是其操作数的值。 一元加运算符的操作数必须是一个算术类型。

整型提升是对整型操作数执行的。 结果类型是操作数将提升到的类型。 因此，表达式 `+ch`（其中 `ch` 的类型为 `char`）的结果类型为 `int`；值不会进行修改。 有关提升是如何完成的详细信息，请参阅标准转换。

`-` 运算符

一元求反运算符（`-`）生成其操作数的负数。 一元求反运算符的操作数必须是算术类型。

将对整型操作数执行整型提升，并且结果类型将是操作数将提升到的类型。 有关提升是如何执行的详细信息，请参阅标准转换。

通过从 2^n 中减去操作数的值来执行无符号数量的一元求反运算，其中 n 是给定的无符号类型的对象的位数。

另请参阅

使用一元运算符的表达式
C++ 内置运算符、优先级和关联性

Last updated on 2026/01/08

表达式 (C++)

本节描述 C++ 表达式。 表达式是用于实现以下一个或多个目的而使用的运算符和操作数的序列：

- 计算来自操作数的值。
- 指定对象或函数。
- 产生“副作用”。 （副作用是除表达式计算之外的任何操作 - 例如，修改对象的值。）

在 C++ 中，可以重载运算符，并且其含义可以是用户定义的。 但是，不能修改其优先级以及它们采用的操作数的数目。 本节描述了使用语言提供而不是重载的运算符的语法和语义。 除了表达式的类型和表达式的语义之外，还包括以下主题：

- 主要表达式
- 作用域解析运算符
- 后缀表达式
- 带一元运算符的表达式
- 带二元运算符的表达式
- 条件运算符
- 常量表达式
- 转换运算符
- 运行时类型信息

其他节中有关运算符的主题：

- C++ 内置运算符、优先级和关联性
- 重载运算符
- typeid （C++/CLI）

> ⓘ **注意**
>
> 无法重载内置类型的运算符；它们的行为是预定义的。

另请参阅

C++ 语言参考

Last updated on 2025/11/07

表达式的类型

C++ 表达式分为多个类别：

- 主要表达式。 这些是从中构成所有其他表达式的构造块。
- 后缀表达式。 这些是后跟运算符的主表达式 - 例如，数组下标或后缀递增运算符。
- 带一元运算符的表达式。 一元运算符仅作用于表达式中的某个操作数。
- 带二元运算符的表达式。 二元运算符在表达式中操作两个操作数。
- 带条件运算符的表达式。 条件运算符是 C++ 语言中唯一的三元运算符，它操作三个操作数。
- 常量表达式。 常量表达式完全由常量数据组成。
- 带显式类型转换的表达式。 可以在表达式中使用显式类型转换或“强制转换”。
- 带指向成员的指针运算符的表达式。
- 强制转换。 可在表达式中使用类型安全“强制转换”。
- 运行时类型信息。 在程序执行期间确定对象的类型。

另请参阅

表达式

Last updated on 2025/11/07

主要表达式

主表达式是更复杂的表达式的构建基块。 它们可能是范围解析运算符（`::`）限定的文本、名称和名称。 主表达式可能具有以下任何形式：

primary-expression
  literal
  this
  name
  :: *name* **(** *expression* **)**

A *literal* 是常量主表达式。 其类型取决于其规范的形式。 有关指定文本的详细信息，请参阅文本 。

关键字 **this** 是指向类实例的指针。 它在非静态成员函数中可用，并指向从中调用函数的类的实例。 **this** 关键字不能在类成员函数的正文之外使用。

有关指针类型 **this** 的详细信息，请参阅 this 指针。

范围解析运算符 （`::`） 后跟名称是主表达式。 此类名称必须是全局范围内的名称，而不是成员名称。 名称的声明确定表达式的类型。 它是一个 l 值（即，如果声明名称为 l-value，它可以显示在赋值表达式的左侧）。 范围解析运算符允许引用全局名称，即使该名称隐藏在当前作用域中也是如此。 有关如何使用范围解析运算符的示例，请参阅 Scope 。

括在括号中的表达式是主表达式。 其类型和值与未分析表达式的类型和值相同。 如果未分析的表达式是 l 值，则为 l 值。

主要表达式的示例包括：

C++

```
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
```

这些示例都被视为各种形式的 *名称*，例如主要表达式：

C++

```
MyClass // an 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
```

另请参阅

表达式类型

Last updated on 2025/11/07

省略号和可变参数模板

本文介绍了如何将省略号（`...`）与 C++ 可变参数模板一起使用。 省略号在 C 和 C++ 中有多种用法。 其中包括函数的变量参数列表。 C 运行时库中的 `printf()` 函数是最著名的示例之一。

“可变参数模板”是支持任意数量的参数的类或函数模板。 此机制对 C++ 库开发人员特别有用：可以将其应用于类模板和函数模板，从而提供广泛的类型安全且非凡的功能和灵活性。

语法

可变参数模板以两种方式使用省略号。 在参数名称的左侧，表示“参数包”，在参数名称的右侧，将参数包扩展为单独的名称。

下面是*可变参数类模板*定义语法的一个基本示例：

C++

```
template<typename... Arguments> class classname;
```

对于参数包和扩展，您可根据您的偏好在省略号周围添加空白，如以下示例所示：

C++

```
template<typename ...Arguments> class classname;
```

或者此示例：

C++

```
template<typename ... Arguments> class classname;
```

本文使用第一个示例中显示的约定（省略号已附加到 **`typename`**）。

在前面的示例中，*`Arguments`* 是参数包。 类 `classname` 可以接受可变数量的参数，如以下示例所示：

C++

```
template<typename... Arguments> class vtclass;

vtclass< > vtinstance1;
vtclass<int> vtinstance2;
vtclass<float, bool> vtinstance3;
vtclass<long, std::vector<int>, std::string> vtinstance4;
```

通过使用可变参数类模板定义，还可以至少需要一个参数：

```C++
template <typename First, typename... Rest> class classname;
```

下面是*可变参数函数模板*语法的基本示例：

```C++
template <typename... Arguments> returntype functionname(Arguments... args);
```

然后，将展开 `Arguments` 参数包以供使用，如下一部分所示。

可以采用其他形式的可变参数函数模板语法，包括但不限于以下示例：

```C++
template <typename... Arguments> returntype functionname(Arguments&... args);
template <typename... Arguments> returntype functionname(Arguments&&... args);
template <typename... Arguments> returntype functionname(Arguments*... args);
```

也允许使用说明符（如 `const`）：

```C++
template <typename... Arguments> returntype functionname(const Arguments&... args);
```

与可变参数模板类定义一样，可以生成要求至少一个参数的函数：

```C++
template <typename First, typename... Rest> returntype functionname(const First&
first, const Rest&... args);
```

可变参数模板使用 `sizeof...()` 运算符（与旧的 `sizeof()` 运算符无关）：

```C++
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...);
}
```

有关省略号位置的更多信息

此前，本文介绍了以以下形式定义参数包和扩展的省略号位置：“参数名称的左侧表示参数包，参数名称的右侧将参数包扩展为单独的名称”。 虽然从技术上讲的确如此，但在转换为代码的过程中可能会造成混淆。 请注意以下几点：

- 在模板参数列表（`template <parameter-list>`）中，`typename...` 引入了模板参数包。
- 在参数声明子句（`func(parameter-list)`）中，“顶级”省略号引入了函数参数包，并且省略号定位很重要：

  C++
  ```
  // 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);
  ```

- 其中省略号显示在参数名称的后面，并且您具有一个参数包扩展。

示例

说明可变参数函数模板机制的一个好方法是在重写 `printf` 的某些功能时使用它：

C++
```
#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);
}
```

输出

Output

```
1
10, 20
100, 200, 300
first, 2, third, 3.14159
```

注意

合并可变参数函数模板的大多数实现使用某种形式的递归，但与传统递归略有不同。 传统的递归涉及使用同一签名调用自身的函数。 （它可能会重载或模板化，但每次选择相同的签名。）可变参数递归涉及通过使用变化（基本都是减少）数量的参数来调用可变参数函数模板，从而每次都清除不同的签名。 仍然需要“基案例”，但递归的性质不同。

Last updated on 2025/11/07

后缀表达式

后缀表达式包含主表达式或者其中的后缀运算符跟在主表达式之后的表达式。 下表列出了后缀运算符。

后缀运算符

展开表

运算符名称	运算符表示法
下标运算符	[.]
函数调用运算符	[.]
显式类型转换运算符	type-name ( )
成员访问运算符	。 or ->
后缀递增运算符	++
后缀递减运算符	--

以下语法描述了可能的后缀表达式：

```
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 )
```

上面的 postfix-expression 可能是*主表达式*或另一个后缀表达式。 后缀表达式从左到右进行分组，这允许表达式按如下方式链接起来：

C++

```
func(1)->GetValue()++
```

在上面的表达式中，`func` 是主表达式，`func(1)` 是函数后缀表达式，`func(1)->GetValue` 是指定类成员的后缀表达式，`func(1)->GetValue()` 是另一个函数后缀表达式，整个表达式是增加 GetValue 的返回值的后缀表达式。 该表达式的整体含义是作为自变量传递 1 的 "call func，并作为返回值获取一个指向类的指针。 然后调用此类上的 `GetValue()`，接着递增返回的值。

上面列出的表达式是赋值表达式，这意味着这些表达式的结果必须为右值。

后缀表达式形式

```
C++
simple-type-name ( expression-list )
```

指示构造函数的调用。 如果 simple-type-name 是基本类型，则表达式列表必须是单个表达式，并且该表达式指示表达式的值将转换为基础类型。 此类强制转换表达式模仿构造函数。 由于此形式允许使用相同的语法来构造基本类型和类，因此它在定义模板类时特别有用。

cast-keyword 是 、`dynamic_cast` 或 `static_cast` 之一`reinterpret_cast`。 更多信息可在 dynamic_cast、static_cast 和 reinterpet_cast 中找到。

`typeid` 运算符被视为后缀表达式。 请参阅 typeid 运算符。

形式和实际自变量

调用程序将“实参”中的信息传递给被调用的函数。被调用的函数使用相应的“形参”访问该信息。

当调用函数时，将执行以下任务：

- 计算所有实参（调用方提供的参数）。 没有计算这些自变量的隐含顺序，但所有自变量都会计算，并且所有副作用都会在进入该函数前完成。
- 使用每个形参在表达式列表中对应的实参来初始化该形参。 （形参是在函数头中声明并在函数体中使用的参数。）转换就像通过初始化完成的一样 - 标准和用户定义的转换都是在将实参转换为正确类型时执行的。 以下代码从概念上演示了所执行的初始化：

  ```
  C++
  void Func( int i ); // Function prototype
  ...
  Func( 7 );          // Execute function call
  ```

  调用前的概念性初始化为：

  ```
  C++
  int Temp_i = 7;
  Func( Temp_i );
  ```

  请注意，初始化就像使用等号语法（而不是括号语法）一样执行。 在将值传递到函数之前制作了 `i` 的副本。 有关详细信息，请参阅 初始值设定项 和 转换。

因此，如果函数原型（声明）对 `long` 类型的参数进行调用，并且调用程序提供了 `int` 类型的实参，则会使用到 `long` 类型的标准类型转换提升该实参（请参阅标准转换）。

如果提供了一个实际自变量，但它没有到形式自变量类型的标准的或用户定义的转换，则是一个错误。

对于类类型的实际自变量，将通过调用类的构造函数初始化形式自变量。 （有关这些特殊类成员函数的详细信息，请参阅构造函数。）

- 执行函数调用。

以下程序片段演示了函数调用：

C++

```
// 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 )
{
}
```

当从 main 调用 `func` 时，将使用 `param1` （`i` 将转换为 `i` 类型以对应使用标准转换的正确类型）的值初始化形参 `long`，并使用 `param2` （`j` 将转换为使用标准转换的 `j` 类型）的值初始化形参 `double`。

自变量类型的处理

不能在函数主体内更改声明为 `const` 类型的形参。 函数可以更改类型不是 `const` 的任何参数。但是，更改对于函数而言是本地进行的，且不会影响实参的值，除非实参是对非 `const` 类型的对象的引用。

以下函数阐释了其中的一些概念：

C++

```
// 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;
}
```

省略号和默认参数

通过使用下列两种方法之一，可以声明函数以接受比函数定义中指定的自变量更少的自变量：省略号 (`...`) 或默认自变量。

省略号表示可能需要参数，但声明中未指定数目和类型。 这通常是较差的 C++ 编程做法，因为它使您无法获得 C++ 的一个优点，即类型安全。 不同的转换将应用于使用省略号声明的函数，而不是应用于那些形参和实参类型的已知函数：

- 如果实参的类型为 `float`，则在函数调用前将其提升为 `double` 类型。
- 使用整数提升将任何 `signed char` 或 `unsigned char`、`signed short` 或 `unsigned short`、枚举类型或位字段转换为 `signed int` 或 `unsigned int`。
- 类类型的所有参数都作为数据结构通过值进行传递；副本是由二进制复制创建的，而不是通过调用类的复制构造函数（如果存在）创建的。

如果使用省略号，则必须在参数列表中最后声明它。 有关传递可变数量的参数的详细信息，请参阅《运行时库参考》中对 va_arg, va_start, and va_list 的讨论。

有关 CLR 编程中默认参数的信息，请参阅变量参数列表 (...) (C++/CLI)。

如果函数调用中没有提供值，则可通过默认参数指定参数应采用的值。 以下代码片段演示默认自变量的工作方式。 有关指定默认参数的限制的详细信息，请参阅默认参数。

C++

```
// expre_Ellipsis_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;
}
```

上面的程序声明一个采用两个参数的函数 `print`。 而第二个参数 terminator 具有默认值 `"\n"`。 在 `main` 中，对 `print` 的前两个调用允许默认的第二个参数提供新行以终止打印的字符串。 第三个调用为第二个参数指定显式值。 该程序的输出为

Output

```
hello,
world!
good morning, sunshine.
```

另请参阅

表达式类型

Last updated on 2025/11/07

具有一元运算符的表达式

一元运算符只对表达式中的一个作数执行作。 一元运算符如下所示：

- 间接运算符 （*）
- 地址运算符 （&）
- 一元加运算符 （+）
- 一元求反运算符 （-）
- 逻辑求反运算符 （!）
- 一者的补数运算符 （~）
- 前缀递增运算符 （++）
- 前缀递减运算符 （--）
- 强制转换运算符 （）
- sizeof 运算符
- alignof 运算符
- noexcept表达式
- new 运算符
- delete 运算符

这些运算符具有从右到左的关联性。 一元表达式通常涉及后缀或主表达式之前的语法。

语法

`unary-expression`:

`postfix-expression`

`++` `cast-expression`

`--` `cast-expression`

`unary-operator` `cast-expression`

`sizeof` `unary-expression`

`sizeof` `(` `type-id` `)`

`sizeof` `...` `(` `identifier` `)`

`alignof` `(` `type-id` `)`

`noexcept-expression`

`new-expression`

`delete-expression`

`unary-operator`：其中一个

`*` `&` `+` `-` `!` `~`

注解

任何 `postfix-expression` 都被视为一个 `unary-expression`，因为任何 `primary-expression` 被视为一个 `postfix-expression`，任何 `primary-expression` 都被视为一个 `unary-expression` 。 有关详细信息，请参阅 后缀表达式 和 主表达式。

`cast-expression`这是一个`unary-expression`具有可选强制转换以更改类型。 有关详细信息，请参阅 Cast 运算符： ()。

`noexcept-expression`该参数是一个`constant-expression` `noexcept-specifier`参数。 有关详细信息，请参阅 noexcept。

引用`new-expression` `new`运算符。 引用`delete-expression` `delete`运算符。 有关详细信息，请参阅 new 运算符 和 delete 运算符。

另请参阅

表达式类型

Last updated on 2025/11/07

使用二元运算符的表达式

二元运算符在表达式中操作两个操作数。 二元运算符为：

- 乘法运算符
  - 乘 (*)
  - 除 (/)
  - 取模 (%)
- 相加运算符
  - 加 (+)
  - 减法 (–)
- 移位运算符
  - 右移 (>>)
  - 左移 (<<)
- 关系运算符和相等运算符
  - 小于号 (<)
  - 大于号 (>)
  - 小于或等于 (<=)
  - 大于或等于 (>=)
  - 等于 (==)
  - 不等于 (!=)
- 位运算符
  - 按位“与”(&)
  - 按位“异或”(^)
  - 按位“与或”(|)
- 逻辑运算符
  - 逻辑“与”(&&)

    - 逻辑“或”(||)
- 赋值运算符
    - 赋值 (=)
    - 加法赋值 (+=)
    - 减法赋值 (-=)
    - 乘法赋值 (*=)
    - 除法赋值 (/=)
    - 取模赋值 (%=)
    - 左移赋值 (<<=)
    - 右移赋值 (>>=)
    - 按位“与”赋值 (&=)
    - 按位“异或”赋值 (^=)
    - 按位“与或”/赋值 (|=)
- 逗号运算符 (,)

另请参阅

表达式类型

Last updated on 2025/11/07

C++ 常量表达式

常量值是指不会更改的值。 C + + 提供了两个关键字，它们使你能够表达不打算修改对象的意图，还可让你实现该意图。

C++ 需要常量表达式（计算结果为常量的表达式）以便声明：

- 数组边界
- case 语句中的选择器
- 位域长度规范
- 枚举初始值设定项

常量表达式中合法的唯一操作数是：

- 文字
- 枚举常量
- 声明为使用常量表达式初始化的常量的值
- `sizeof` 表达式

必须将非整型常量（显式或隐式）转换为常量表达式中合法的整型。 因此，以下代码是合法的：

C++

```
const double Size = 11.0;
char chArray[(int)Size];
```

到整型的显式转换在常量表达式中是合法的；所有其他类型和派生类型是非法的（在用作 `sizeof` 运算符的操作数时除外）。

逗号运算符和赋值运算符不能用于常量表达式。

另请参阅

表达式类型

Last updated on 2025/11/07

表达式的语义

表达式根据其运算符的优先级和分组来计算。 （词汇约定中的运算符优先级和关联性显示了 C++ 运算符对表达式施加的关系。）

计算顺序

请看以下示例：

C++

```
// 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";
}
```

Output

```
38
38
54
```

表达式计算顺序

上图中显示的表达式的计算顺序取决于运算符的优先级和关联性：

1. 乘法 (*) 在此表达式中具有最高优先级；因此子表达式 `b * c` 首先计算。
2. 加法 (+) 具有第二高的优先级，因此， `a` 将与 `b` 和 `c` 的乘积相加。
3. 左移 (<<) 在表达式中的优先级最低，但出现了两次。 由于左移运算符从左到右分组，因此先计算左子表达式，再计算右子表达式。

当使用括号为子表达式分组时，它们将更改表达式的计算优先级和顺序，如下图所示。

带括号的表达式计算顺序

类似于上图的表达式的计算纯粹是为了展示副作用 - 在本例中是将信息转移到标准输出设备。

表达式中的表示法

在指定操作数时，C++ 语言指定某些兼容性。 下表显示了需要 type 类型操作数的运算符可接受的操作数类型。

运算符可接受的操作数类型

展开表

应为类型	允许的类型
type	`const` *类型* `volatile` *类型* *type&* `const` *type&* `volatile` *type&* `volatile const` *类型* `volatile const` *type&*
type	type `const` *类型 ** `volatile` *类型 ** `volatile const` *类型 **
`const` *类型*	type `const` *类型* `const` *type&*
`volatile` *类型*	type `volatile` *类型* `volatile` *type&*

由于上述规则始终可以组合使用，因此，可以在指针所需的位置提供指向可变对象的 const 指针。

不明确的表达式

某些表达式的意义不明确。 当在同一表达式中多次修改对象的值时，这些表达式最常见。 当语言没有定义表达式的计算顺序时，这些表达式依赖于特定的顺序计算。 请考虑以下示例：

```
int i = 7;

func( i, ++i );
```

C++ 语言不保证计算函数调用的参数的顺序。 因此，在前面的示例中，`func` 的参数可以接受值 7 和 8 或 8 和 8，取决于参数是从左到右还是从右到左计算。

C++ 序列点（Microsoft 专用）

在连续的“序列点”之间，表达式只能修改对象的值一次。

C++ 语言定义当前未指定序列点。 Microsoft C++ 对涉及 C 运算符但不涉及重载运算符的任何表达式使用与 ANSI C 相同的序列点。 当重载运算符时，语义从运算符排序更改为函数调用排序。 Microsoft C++ 使用以下序列点：

- 逻辑“与”运算符（`&&`）的左操作数。 完全计算逻辑“与”运算符的左操作数，并在继续之前完成所有副作用。 不保证一定会计算逻辑“与”运算符的右操作数。
- 逻辑“或”运算符（`||`）的左操作数。 完全计算逻辑“或”运算符的左操作数，并在继续之前完成所有副作用。 不保证一定会计算逻辑“或”运算符的右操作数。
- 逗号运算符的左操作数。 完全计算逗号运算符的左操作数，并在继续之前完成所有副作用。 始终计算逗号运算符的两个操作数。
- 函数调用运算符。 计算函数调用表达式以及函数的所有自变量（包括默认自变量），并在进入函数之前完成所有副作用。 在自变量或函数调用表达式之间没有指定的计算顺序。
- 条件运算符的第一个操作数。 完全计算条件运算符的第一个操作数，并在继续之前完成所有副作用。
- 完整的初始化表达式的末尾，如声明语句中的初始化的末尾。
- 表达式语句中的表达式。 表达式语句由可选表达式后跟分号 (;) 组成。 表达式为其副作用完全计算。
- 选择（if 或 switch）语句中的控制表达式。 完全计算该表达式，并在执行依赖于选择的代码之前完成所有副作用。
- while 或 do 语句的控制表达式。 完全计算该表达式，并在执行 while 或 do 循环的下一次迭代中的任何语句之前完成所有副作用。

- for 语句的所有三个表达式。 完全计算每个表达式，并在移动到下一个表达式之前完成所有副作用。
- return 语句中的表达式。 完全计算该表达式，并在控制权返回到调用函数之前完成所有副作用。

另请参阅

表达式

Last updated on 2025/11/07

强制转换

在 C++ 中，如果类派生自包含一个或多个虚拟函数的基类，则指向该基类类型的指针可用于调用派生类对象中的虚拟函数。 包含虚函数的类有时被称为“多态类”。

类层次结构

`C`类型的对象可以如下所示进行可视化：

具有子对象 B 和 A 的类 C

给定`C`类的一个实例，存在`B`子对象和`A`子对象。 包括 `C` 和 `A` 子对象的 `B` 实例是“完整对象。”

由于派生类完全包含它派生自的所有基类的定义，因此将指针转换至其基类中的任何一个都是安全的（也称为向上转换）。 鉴于指向基类的指针，可以安全地将指针强制转换为派生类的实例（也称为向下转换）。

通过使用运行时类型信息，可以检查指针实际是否指向完整对象，并可以安全转换以指向其层次结构中的另一个对象。 dynamic_cast运算符执行运行时检查以确保操作安全。 最好设计类层次结构，以便可以使用虚拟函数来避免向下转换的需要。 但是，如果必须向下转换，请使用`dynamic_cast`确保操作安全。

对于非多态类型的转换，可以使用static_cast运算符（本主题说明静态和动态强制转换之间的差异，以及何时适合使用它们）。

以下示例演示了 `dynamic_cast` 和 `static_cast` 的用法：

C++

```
#include <iostream>

class Base {
public:
    virtual void print() { std::cout << "Base\n"; }
};

class Derived1 : public Base {
public:
    void print() override { std::cout << "Derived1\n"; }
};

class Derived2 : public Base {
public:
```

```
    void print() override { std::cout << "Derived2\n"; }
};

class MostDerived : public Derived1, public Derived2 {
public:
    void print() override { std::cout << "MostDerived\n"; }
};

int main() {
    MostDerived md;
    Base* b1 = static_cast<Derived1*>(&md); // Upcast to Derived1 is safe
    Base* b2 = static_cast<Derived2*>(&md); // Upcast to Derived2 is safe

    // Downcast to MostDerived is ambiguous and unsafe
    // MostDerived* md1 = static_cast<MostDerived*>(b1); // This won't compile
    // MostDerived* md2 = static_cast<MostDerived*>(b2); // This won't compile

    // Correct way to downcast in this situation
    MostDerived* md1 = dynamic_cast<MostDerived*>(b1); // This is safe
    MostDerived* md2 = dynamic_cast<MostDerived*>(b2); // This is safe

    md1->print(); // Prints "MostDerived"
    md2->print(); // Prints "MostDerived"

    return 0;
}
```

本部分涵盖了以下主题：

- 强制转换运算符
- 运行时类型信息

另请参阅

表达式

ⓘ **注意：** 作者在 AI 的帮助下创作了此文章。 了解详细信息

Last updated on 2025/11/07

强制转换运算符

有几种特定于 C++ 语言的转换运算符。 这些运算符用于删除旧式 C 语言转换中的一些多义性和危险继承。 这些运算符是：

- dynamic_cast 用于多态类型的转换。
- static_cast 用于非多态类型的转换。
- const_cast 用于删除 `const`、`volatile` 和 `__unaligned` 特性。
- reinterpret_cast 用于对位进行简单的重新解释。
- safe_cast 在 C++/CLI 中用于生成可验证的 MSIL。

在万不得已时使用 `const_cast` 和 `reinterpret_cast`，因为这些运算符与旧的样式转换带来的危险相同。 但是，若要完全替换旧的样式转换，仍必须使用它们。

另请参阅

强制转换

Last updated on 2025/11/07

dynamic_cast 运算符

将操作数 `expression` 转换为 `type-id` 类型的对象。

语法

```
dynamic_cast < type-id > ( expression )
```

备注

`type-id` 必须是针对以前定义的类类型的指针或引用，或者是“指向 void 的指针”。 如果 `expression` 是指针，则 `type-id` 类型必须为指针，或者如果 `type-id` 是引用，则为左值。

请参阅 static_cast，了解静态和动态强制转换之间的区别，以及什么情况下适合使用这两种转换。

托管代码中的 **`dynamic_cast`** 行为有两项中断性变更：

- 对指向装箱枚举的基础类型的指针的 **`dynamic_cast`** 将在运行时失败，返回 0 而不是转换后的指针。
- 当 **`dynamic_cast`** 是指向值类型的内部指针时，`type-id` 将不再引发异常；相反，强制转换在运行时失败。 强制转换返回 0 个针值，而不是引发。

如果 `type-id` 是指向 `expression` 的明确可访问的直接或间接基类的指针，则结果是指向 `type-id` 类型的唯一子对象的指针。 例如：

C++

```
// 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
}
```

这种类型的转换称为“向上转换”，因为它将指针向上移动一个类层次结构，从派生类移至它从中派生的类。 向上转换是一种隐式转换。

如果 `type-id` 是 void*，则进行运行时检查以确定 `expression` 的实际类型。 结果是指向由 `expression` 指向的完整对象的指针。 例如：

C++

```
// 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
}
```

如果 `type-id` 不是 `void*`，则进行运行时检查，看看是否可以将 `expression` 指向的对象转换为 `type-id` 指向的类型。

如果 `expression` 的类型是 `type-id` 类型的基类，则进行运行时检查，看看 `expression` 是否实际指向 `type-id` 类型的完整对象。 如果是，则结果是指向 `type-id` 类型的完整对象的指针。 例如：

C++

```
// 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
}
```

这种类型的转换称为“向下转换”，因为它将指针向下移动一个类层次结构，从给定类移至从中派生的类。

如果有多重继承，可能会导致不明确。 来看看下图中显示的类层次结构。

对于 CLR 类型，如果可以隐式执行转换，则 `dynamic_cast` 会导致无操作，或者执行 MSIL `isinst` 指令，该指令执行动态检查并在转换失败时返回 `nullptr`。

以下示例使用 `dynamic_cast` 来确定类是否是特定类型的实例：

C++

```
// 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);
}
```

指向 `D` 类型对象的指针可以安全地强制转换为 `B` 或 `C`。 但是，如果 `D` 强制转换为指向 `A` 对象的指针，会产生 `A` 的哪个实例？ 这将导致不明确的强制转换错误。 若要解决此问题，可以执行两个明确的强制转换。 例如：

C++

```
// 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
}
```

使用虚拟基类时，可能会导致更多不明确的情况。 来看看下图中显示的类层次结构。

显示虚拟基类的类层次结构

在此层次结构中，`A` 是一个虚拟基类。 对于类 `E` 的实例和指向 `A` 子对象的指针，指向 `dynamic_cast` 的指针的 `B` 将因不明确性而失败。 必须首先转换回完整的 `E` 对象，然后以明确的方式备份层次结构，才能访问正确的 `B` 对象。

来看看下图中显示的类层次结构。

显示重复基类的类层次结构

在给定 `E` 类型对象和指向 `D` 子对象的指针的情况下，若要从 `D` 子对象导航到最左侧的 `A` 子对象，可以进行三次转换。 可以执行从 `dynamic_cast` 指针到 `D` 指针的 `E` 转换，然后执行从 `dynamic_cast` 到 `E` 的转换（可以是 `B`，也可以是隐式转换），最后执行从 `B` 到 `A` 的隐式转换。 例如：

C++

```
// 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
}
```

`dynamic_cast` 运算符还可用于执行“交叉强制转换”。使用同一类层次结构时，只要完整对象为 `B` 类型，就可以强制转换指针，例如从 `D` 子对象强制转换为 `E` 子对象。

就交叉强制转换而言，只需执行两个步骤即可完成从指向 `D` 的指针到指向最左侧 `A` 子对象的指针的转换。 可以执行从 `D` 到 `B` 的交叉强制转换，然后从 `B` 到 `A` 的隐式转换。 例如：

C++

```
// 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
}
```

一个空指针值通过 `dynamic_cast` 转换为目标类型的空指针值。

使用 `dynamic_cast < type-id > ( expression )` 时，如果 `expression` 无法安全地转换为 `type-id` 类型，运行时检查将导致强制转换失败。 例如：

C++

```
// 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
}
```

未能强制转换为指针类型的值是空指针。 如果对引用类型的强制转换失败，会引发 bad_cast 异常。 如果 `expression` 没有指向或引用有效对象，则会引发 `__non_rtti_object` 异常。

有关 异常的说明，请参阅 `__non_rtti_object`。

示例

下面的示例创建了基类（结构 A）指针，指向一个对象（结构 C）。 这一点，加上有虚函数，可以实现运行时多形性。

此示例还调用层次结构中的非虚拟函数。

C++

```cpp
// 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);

   // fails because B knows nothing about C
    B BonStack;
    Globaltest(BonStack);
}
```

Output

```
in C
test2 in B
in GlobalTest
Can't cast to C
```

另请参阅

强制转换运算符
关键字

Last updated on 2025/11/07

bad_cast 异常

由于强制转换为引用类型失败， 运算符引发 bad_cast 异常`dynamic_cast`。

语法

```
catch (bad_cast)
   statement
```

备注

bad_cast 的接口为：

```C++
class bad_cast : public exception
```

以下代码包含失败的 `dynamic_cast` 引发 bad_cast 异常的示例。

```C++
// 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();
```

```
    }
}
```

由于强制转换的对象 (Shape) 不是派生自指定的强制转换类型 (Circle)，因此引发异常。 若要避免此异常，请将下列声明添加到 `main`：

C++

```
Circle circle_instance;
Circle& ref_circle = circle_instance;
```

然后在 `try` 块中反转强制转换的意义，如下所示：

C++

```
Shape& ref_shape = dynamic_cast<Shape&>(ref_circle);
```

成员

构造函数

展开表

构造函数	说明
bad_cast	`bad_cast` 类型的对象的构造函数。

Functions

展开表

函数	说明
what	待定

运算符

展开表

运算符	说明
operator=	用于将一个 `bad_cast` 对象赋予另一个对象的赋值运算符。

bad_cast

`bad_cast` 类型的对象的构造函数。

```C++
bad_cast(const char * _Message = "bad cast");
bad_cast(const bad_cast &);
```

operator=

用于将一个 `bad_cast` 对象赋予另一个对象的赋值运算符。

```C++
bad_cast& operator=(const bad_cast&) noexcept;
```

what

```C++
const char* what() const noexcept override;
```

另请参阅

dynamic_cast 运算符
关键字
现代 C++ 处理异常和错误的最佳做法

Last updated on 2025/11/07

static_cast 运算符

仅根据表达式中存在的类型，将 expression 转换为 type-id 类型。

语法

```
static_cast <type-id> ( expression )
```

备注

在标准 C++ 中，不进行运行时类型检查来帮助确保转换的安全。 在 C++/CX 中，将执行编译时和运行时检查。 有关更多信息，请参见 强制转换中定义的接口的私有 C++ 特定实现。

`static_cast` 运算符可用于将指向基类的指针转换为指向派生类的指针等操作。 此类转换并非始终安全。

通常使用 `static_cast` 转换数值数据类型，例如将枚举型转换为整型或将整型转换为浮点型，而且你能确定参与转换的数据类型。 `static_cast` 转换安全性不如 `dynamic_cast` 转换，因为 `static_cast` 不执行运行时类型检查，而 `dynamic_cast` 执行该检查。 对不明确的指针的 `dynamic_cast` 将失败，而 `static_cast` 的返回结果看似没有问题；这是危险的。 尽管 `dynamic_cast` 转换更加安全，但是 `dynamic_cast` 只适用于指针或引用，而且运行时类型检查也是一项开销。 有关详细信息，请参阅 dynamic_cast 运算符。

在下面的示例中，因为 `D* pd2 = static_cast<D*>(pb);` 可能有不在 `D` 内的字段和方法，所以行 `B` 不安全。 但是，因为 `B* pb2 = static_cast<B*>(pd);` 始终包含所有 `D`，所以行 `B` 是安全的转换。

C++

```cpp
// 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.
}
```

与 dynamic_cast 不同，`static_cast` 的 `pb` 转换不执行运行时检查。 由 `pb` 指向的对象可能不是 `D` 类型的对象，在这种情况下使用 `*pd2` 会是灾难性的。 例如，调用 `D` 类（而非 `B` 类）的成员函数可能会导致访问冲突。

`dynamic_cast` 和 `static_cast` 运算符可以在整个类层次结构中移动指针。 然而，`static_cast` 完全依赖于转换语句提供的信息，因此可能不安全。 例如：

C++

```
// 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);
}
```

如果 `pb` 确实指向 `D` 类型的对象，则 `pd1` 和 `pd2` 将获取相同的值。 如果 `pb == 0`，它们也将获取相同的值。

如果 `pb` 指向 `B` 类型的对象，而非指向完整的 `D` 类，则 `dynamic_cast` 足以判断返回零。 但是，`static_cast` 依赖于程序员的断言，即 `pb` 指向 `D` 类型的对象，因而只是返回指向那个假定的 `D` 对象的指针。

因此，`static_cast` 可以反向执行隐式转换，而在这种情况下结果是不确定的。 这需要程序员来验证 `static_cast` 转换的结果是否安全。

该行为也适用于类以外的类型。 例如，`static_cast` 可用于将 int 转换为 `char`。 但是，得到的 `char` 可能没有足够的位来保存整个 `int` 值。 同样，这需要程序员来验证 `static_cast` 转换的结果是否安全。

`static_cast` 运算符还可用于执行任何隐式转换，包括标准转换和用户定义的转换。 例如：

C++

```
// 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);
}
```

`static_cast` 运算符可以将整数值显式转换为枚举类型。 如果整型值不在枚举值的范围内，生成的枚举值是不确定的。

`static_cast` 运算符将空指针值转换为目标类型的空指针值。

任何表达式都可以通过 `static_cast` 运算符显式转换为 void 类型。 目标 void 类型可以选择性地包含 `const`、 `volatile` 或 `__unaligned` 特性。

`static_cast` 运算符无法强制转换掉 `const`、 `volatile` 或 `__unaligned` 特性。 有关移除这些特性的详细信息，请参阅 const_cast Operator。

C++/CLI：由于在一个重定位垃圾回收器顶部执行无检查转换存在的危险，你只应在确信代码将正常运行的时候，在性能关键代码中使用 `static_cast`。 如果必须在发布模式下使用 `static_cast`，请在调试版本中用 safe_cast 替换它以确保成功。

另请参阅

强制转换运算符
关键字

Last updated on 2025/11/07

const_cast 运算符

从某个类删除 `const`、`volatile` 和 `__unaligned` 属性。

语法

```
const_cast <type-id> (expression)
```

备注

指向任何对象类型的指针或指向数据成员的指针可显式转换为完全相同的类型（`const`、`volatile` 和 `__unaligned` 限定符除外）。 对于指针和引用，结果将引用原始对象。 对于指向数据成员的指针，结果将引用与指向数据成员的原始（未强制转换）的指针相同的成员。 根据引用对象的类型，通过生成的指针、引用或指向数据成员的指针的写入操作可能产生未定义的行为。

你不能使用 `const_cast` 运算符直接重写常量变量的常量状态。

`const_cast` 运算符将空指针值转换为目标类型的空指针值。

示例

C++

```cpp
// 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();
}
```

在包含 `const_cast` 的行中， `this` 指针的数据类型为 `const CCTest *`。 `const_cast` 运算符会将 `this` 指针的数据类型更改为 `CCTest *`，以允许修改成员 `number`。 强制转换仅对其所在的语句中的其余部分持续。

另请参阅

强制转换运算符
关键字

Last updated on 2025/11/07

reinterpret_cast 运算符

允许将任何指针转换为任何其他指针类型。 也允许将任何整数类型转换为任何指针类型以及反向转换。

语法

```
reinterpret_cast < type-id > ( expression )
```

备注

滥用 `reinterpret_cast` 运算符可能很容易带来风险。 除非所需转换本身是低级别的，否则应使用其他强制转换运算符之一。

`reinterpret_cast` 运算符可用于 `char*` 到 `int*` 或 `One_class*` 到 `Unrelated_class*` 之类的转换，这本身并不安全。

`reinterpret_cast` 的结果不能安全地用于除强制转换回其原始类型以外的任何用途。 在最好的情况下，其他用途也是不可移植的。

`reinterpret_cast` 运算符无法强制转换掉 `const`、 `volatile` 或 `__unaligned` 特性。 有关移除这些特性的详细信息，请参阅 const_cast Operator。

`reinterpret_cast` 运算符将空指针值转换为目标类型的空指针值。

`reinterpret_cast` 的一个实际用途是在哈希函数中，即，通过让两个不同的值几乎不以相同的索引结尾的方式将值映射到索引。

C++

```
#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
```

`reinterpret_cast` 允许将指针视为整数类型。 结果随后将按位移位并与自身进行“异或”运算以生成唯一的索引（具有唯一性的概率非常高）。 该索引随后被标准 C 样式强制转换截断为函数的返回类型。

另请参阅

强制转换运算符
关键字

Last updated on 2025/11/07

运行时类型信息

运行时类型信息 (RTTI) 是一种允许在程序执行过程中确定对象的类型的机制。 RTTI 已添加到 C++ 语言中，因为许多类库供应商将自行实现此功能。 这会导致库之间出现不兼容的情况。 因此，显而易见的是，需要语言级别的对运行时类型信息的支持。

为了清楚起见，此 RTTI 的讨论几乎完全是针对指针展开的。 但讨论的概念也适用于引用。

有三个针对运行时类型信息的 C++ 语言元素：

- dynamic_cast 运算符。

  用于多态类型的转换。

- typeid 运算符。

  用于标识对象的确切类型。

- type_info 类。

  用于保留由 `typeid` 运算符返回的类型信息。

另请参阅

强制转换

Last updated on 2025/11/07

bad_typeid 异常

当 的操作数是 NULL 指针时，typeid 运算符将引发 bad_typeid 异常`typeid`。

语法

```
catch (bad_typeid)
   statement
```

备注

bad_typeid 的接口为：

```C++
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;
};
```

以下示例演示引发 bad_typeid 异常的 `typeid` 运算符。

```C++
// 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;
   }
}
```

输出

Output

```
Object is NULL
```

另请参阅

运行时类型信息
关键字

Last updated on 2025/11/07

type_info 类

type_info 类描述编译器在程序中生成的类型信息。 此类的对象可以有效存储指向类型的名称的指针。 type_info 类还可存储适合比较两个类型是否相等或比较其排列顺序的编码值。 类型的编码规则和排列顺序是未指定的，并且可能因程序而异。

必须包含 `<typeinfo>` 头文件才能使用 type_info 类。 type_info 类的接口是：

C++

```
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;
};
```

不能直接实例化 type_info 类的对象，因为该类只有一个私有复制构造函数。 构造（临时）type_info 对象的唯一方式是使用 typeid 运算符。 由于赋值运算符也是专用的，因此不能复制或给类 type_info 的对象赋值。

`type_info::hash_code` 可定义适合将 typeinfo 类型的值映射到索引值的分布的哈希函数。

运算符 `==` 和 `!=` 分别用于与其他 type_info 对象比较是否相等和不相等。

类型的排列顺序与继承关系之间没有关联。 使用 `type_info::before` 成员函数可确定类型的排序。 不能保证 `type_info::before` 在不同的程序中（甚至是多次运行同一程序时）会产生相同的结果。 通过这种方式， `type_info::before` 类似于 `(&)` 运算符的地址。

`type_info::name` 成员函数可将 `const char*` 返回到以 null 结尾的字符串，该字符串表示类型的用户可读名称。 将缓存所指向的内存，应该从不直接释放它。

`type_info::raw_name` 成员函数特定于 Microsoft。 它将 `const char*` 返回到以 null 结尾的字符串，该字符串表示对象类型的修饰名称。 该名称以其修饰的形式存储以节省空间。 因此，此函数比 `type_info::name` 更快，因为它不需要取消修饰名称。 `type_info::raw_name` 函数返回的字符串在比较运算符中很有用，但它不可读。 如果需要用户可读的字符串，请改用 `type_info::name`。

仅当指定了 /GR（启用运行时类型信息）编译器选项时，才会为多态类生成类型信息。

另请参阅

运行时类型信息

Last updated on 2025/11/07

语句 (C++)

C++ 语句是控制操作对象的方式和顺序的程序元素。 本节包括：

- 概述
- 标记语句
- 语句类别
  - 表达式语句。 这些语句评估表达式的副作用或其返回值。
  - Null 语句。 可以在 C++ 语法需要语句但未采取任何措施的位置提供这些语句。
  - 复合语句。 这些语句是用大括号 ({ }) 括起来的语句组。 可在能够使用单个语句的任何位置使用它们。
  - 选择语句。 这些语句执行一个测试；如果测试的计算结果为 true（非零），则它们将执行代码的一部分。 如果测试的计算结果为 false，则它们可能执行代码的另一部分。
  - 迭代语句。 这些语句用于重复执行代码块，直到满足指定的终止条件。
  - 跳转语句。 这些语句可以立即将控制权转移到函数中的其他位置或从函数中返回控制权。
  - 声明语句。 声明将一个名称引入程序中。

有关异常处理语句的信息，请参阅异常处理。

另请参阅

C++ 语言参考

Last updated on 2025/11/07

C++ 语句概述

C++ 语句将按顺序执行，除非表达式语句、选择语句、迭代语句或跳转语句特意修改了顺序。

语句可以是以下类型之一：

> `labeled-statement`
> `expression-statement`
> `compound-statement`
> `selection-statement`
> `iteration-statement`
> `jump-statement`
> `declaration-statement`
> `try-throw-catch`

在大多数情况下，C++ 语句的语法与 ANSI C89 语句的语法相同。 两者之间的主要区别在于：在 C89 中只允许在块的开头进行声明；而 C++ 通过添加 `declaration-statement` 有效地消除了此限制。 这样，您就能够在程序中可以计算预计算初始化值的某个时点引入变量。

通过在块中的声明变量，您还可以对这些变量的范围和生存期进行精确的控制。

有关语句的文章描述了以下 C++ 关键字：

break
case
catch
continue
default
do

else
__except
__finally
for
goto

if
__if_exists
__if_not_exists
__leave
return

switch
throw
__try
try
while

另请参阅

语句

Last updated on 2025/11/07

带标签的语句

标签用于将程序控制权直接转交给特定语句。

语法

> `labeled-statement`?
>
> `identifier` `:` `statement`
>
> `case` `constant-expression` `:` `statement`
>
> `default` `:` `statement`

标签的范围为整个函数，已在其中声明该标签。

备注

有三种标记语句。 它们全都使用冒号 (`:`) 将某种标签与语句隔开。 `case` 和 `default` 标签特定于 case 语句。

C++

```
#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
}
```

标签和 `goto` 语句

源程序中 *`identifier`* 标签的外观声明了一个标签。 仅 goto 语句可将控制转移到 *`identifier`* 标签。 以下代码片段阐释了 **`goto`** 语句和 *`identifier`* 标签的使用：

标签无法独立出现，必须总是附加到语句。 如果标签需要独立出现，则必须在标签后放置一个 null 语句。

标签具有函数范围，并且不能在函数中重新声明。 但是，相同的名称可用作不同函数中的标签。

C++

```
// 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.
```

`case` 语句中的标签

在 **`case`** 关键字后显示的标签不能在 **`switch`** 语句外部显示。 （此限制也适用于 **`default`** 关键字。）以下代码片段显示 **`case`** 标签的正确用法：

C++

```
// 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;

   case WM_CLOSE:
      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;
}
```

另请参阅

C++ 语句概述
switch 语句 (C++)

Last updated on 2025/11/07

表达式语句

表达式语句导致计算表达式。 出于表达式语句的原因，不会发生控制或迭代的传输。

表达式语句的语法就是

语法

```
[expression ] ;
```

备注

在执行下一个语句前，将计算表达式语句中的所有表达式并完成所有副作用。 最常用的表达式语句是赋值和函数调用。 由于表达式是可选的，因此单独的分号被视为空表达式语句，称为 null 语句。

另请参阅

语句概述

Last updated on 2025/11/07

Null 语句

“null 语句”是缺少 expression 的表达式语句。 当语言的语法调用语句而不是表达式计算时，它很有用。 它包括分号。

Null 语句通常用作迭代语句中的占位符或用作在复合语句或函数的末尾放置标签的语句。

以下代码片段说明如何将一个字符串复制到另一个字符串，并包含 null 语句：

C++

```
// 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()
{
}
```

另请参阅

表达式语句

Last updated on 2025/11/07

复合语句（块）

复合语句包含封闭在大括号 ({ }) 中的零个或多个语句。 可以在任何期望语句出现的位置使用复合语句。 复合语句通常称为“块”。

语法

```
{ [ statement-list ] }
```

备注

以下示例使用复合语句作为 语句的 statement 部分（有关语法的详细信息，请参阅 `if`）：

```C++
if( Amount > 100 )
{
    cout << "Amount was too large to handle\n";
    Alert();
}
else
{
    Balance -= Amount;
}
```

> **注意**
>
> 由于声明是一个语句，因此声明可以是 statement-list 内的某个语句。 因此，复合语句内声明的名称（而不是显式声明为静态的名称）具有局部范围和（对于对象）生存期。 有关处理带局部范围的名称的详细信息，请参阅**范围**。

另请参阅

语句概述

Last updated on 2025/11/07

选择语句 (C++)

C++ 选择语句 if 和 switch 提供了一种有条件地执行代码节的方式。

利用 __if_exists 和 __if_not_exists 语句，你可根据是否存在某个符号来有条件地包括代码。

有关每个语句的语法，请参阅单独的主题。

另请参阅

语句概述

Last updated on 2025/11/07

if-else 语句 (C++)

项目 • 2024/11/21

if-else 语句控制条件分支。 仅当 `condition` 计算结果为非零值（或 `true`）时，才会执行 `if-branch` 中的语句。 如果 `condition` 的值为非零，则执行以下语句，并跳过以下可选 `else` 语句后面的语句。 否则，将跳过以下语句，如果存在 `else`，则随后执行 `else` 后面的语句。

计算结果为非零的 `condition` 表达式包括：

- `true`
- 非 null 指针，
- 任何非零算术值，或
- 一种类类型，用于定义对算术、布尔值或指针类型的明确转换。 （有关转换的信息，请参阅标准转换。）

语法

`init-statement`?

  `expression-statement`

  `simple-declaration`

`condition`?

  `expression`

  `attribute-specifier-seq`$_{opt}$ `decl-specifier-seq` `declarator` `brace-or-equal-initializer`

`statement`?

  `expression-statement`

  `compound-statement`

`expression-statement`?

  `expression`$_{opt}$ `;`

`compound-statement`?

  `{` opt `statement-seq` `}`

`statement-seq`?

  `statement`

  `statement-seq` `statement`

`if-branch`?

`statement`

`else-branch`?

`statement`

`selection-statement`?

`if` `constexpr`$_{opt}$[17] `(` `init-statement`$_{opt}$[17] `condition` `)` `if-branch`

`if` `constexpr`$_{opt}$[17] `(` `init-statement`$_{opt}$[17] `condition` `)` `if-branch` `else` `else-branch`

[17] 从 C++17 开始，此可选元素可用。

if-else 语句

在 `if` 语句的所有形式中，可计算 `condition`，它具有除了结构以外的任何值，包括所有副作用。 控制从 `if` 语句传递给程序中的下一个语句，除非已执行的 `if-branch` 或 `else-branch` 包含 break、continue 或 goto。

`if...else` 语句的 `else` 子句与同一范围内没有相应 `else` 语句的最接近的上一个 `if` 语句相关联。

示例

此示例代码演示了多个正在使用的 `if` 语句，包括使用和不使用 `else`：

C++

```
// if_else_statement.cpp
#include <iostream>

using namespace std;

int main()
{
    int x = 10;

    if (x < 11)
    {
        cout << "x < 11 is true!\n";  // executed
    }
    else
    {
        cout << "x < 11 is false!\n"; // not executed
    }

    // no else statement
```

```
    bool flag = false;
    if (flag == true)
    {
        x = 100; // not executed
    }

    int *p = new int(25);
    if (p)
    {
        cout << *p << "\n"; // outputs 25
    }
    else
    {
        cout << "p is null!\n"; // executed if memory allocation fails
    }
}
```

输出：

输出

```
x < 11 is true!
25
```

带有初始值设定项的 if 语句

从 C++17 开始，`if` 语句还可能包含声明和初始化命名变量的 *`init-statement`* 表达式。当变量仅在 if-statement 范围内需要时，请使用 if-statement 的此形式。 **特定于 Microsoft**：此形式从 Visual Studio 2017 版本 15.3 开始提供，至少需要 /std:c++17 编译器选项。

示例

C++

```
// Compile with /std:c++17

#include <iostream>
#include <mutex>
#include <map>
#include <string>
#include <algorithm>

using namespace std;

map<int, string> m{ {1, "one"}, {2, "two"}, {10,"ten"} };
mutex mx;
bool shared_flag = true; // guarded by mx
```

```
int getValue() { return 42; }

int main()
{
    if (auto it = m.find(10); it != m.end())
    {
        cout << it->second << "\n";
    }

    if (int x = getValue(); x == 42)
    {
        cout << "x is 42\n";
    }

    if (lock_guard<mutex> lock(mx); shared_flag)
    {
        cout << "setting shared_flag to false\n";
        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";
    }
}
```

输出：

Output

```
ten
x is 42
setting shared_flag to false
Error! Token must not be a keyword
```

if constexpr 语句

从 C++17 开始，可以使用函数模板中的 `if constexpr` 语句做出编译时分支决策，而无需求助于多个函数重载。 **特定于 Microsoft**：此形式从 Visual Studio 2017 版本 15.3 开始提供，至少需要 /std:c++17 编译器选项。

示例

此示例演示如何根据发送到模板的类型有条件地编译模板：

C++

```
// Compile with /std:c++17
#include <iostream>

template<typename T>
auto Show(T t)
{
    //if (std::is_pointer_v<T>) // Show(a) results in compiler error for
return *t. Show(b) results in compiler error for return t.
    if constexpr (std::is_pointer_v<T>) // This statement goes away for
Show(a)
    {
        return *t;
    }
    else
    {
        return t;
    }
}

int main()
{
    int a = 42;
    int* pB = &a;

    std::cout << Show(a) << "\n"; // prints "42"
    std::cout << Show(pB) << "\n"; // prints "42"
}
```

`if constexpr` 语句在编译时计算，编译器仅生成与发送到函数模板的参数类型匹配的 **`if`** 分支的代码。 如果注释掉 **`if constexpr`** 语句并取消注释 **`if`** 语句，编译器将为这两个分支生成代码。 这意味着你收到错误：

- 如果调用 `ShowValue(a);`，即使 **`if`** 语句为 false 且代码从不执行，你也会在 `return *t` 上收到错误，因为 `t` 不是指针。
- 如果调用 `ShowValue(pB);`，即使 **`if`** 语句为 true 且代码从不执行，你也会在 `return t` 上收到错误，因为 `t` 是指针。

使用 `if constexpr` 可解决此问题，因为只有与发送到函数模板的参数类型匹配的语句才会被编译。

输出：

输出

```
42
42
```

另请参阅

选择语句
关键字
switch语句 (C++)

反馈

此页面是否有帮助? 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

__if_exists 语句

`__if_exists` 语句测试指定的标识符是否存在。 如果该标识符存在，则执行指定的语句块。

语法

```
__if_exists ( identifier ) {
statements
};
```

参数

identifier
要测试其存在性的标识符。

statements
如果标识符存在，则执行一条或多条语句。

备注

> ⊗ **注意**
>
> 若要实现最可靠的结果，请使用以下约束下的 `__if_exists` 语句。

- 只将 `__if_exists` 语句应用于简单类型而不是模板。
- 将 `__if_exists` 语句应用于类的内部或外部的标识符。 不要将 `__if_exists` 语句应用于局部变量。
- 仅在函数的主体中使用 `__if_exists` 语句。 在函数主体的外部， `__if_exists` 语句仅能测试完全定义的类型。
- 在测试重载函数时，不能测试特定形式的重载。

`__if_exists` 语句的补集是 __if_not_exists 语句。

示例

请注意，此示例使用了模板，不建议这样做。

C++

```cpp
// 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;
}
```

输出

Output

```
In X<T>::Dump()
In A::Dump()
In X<T>::Dump()
T::Dump does not exist
g_bFlag = 1
C::f exists
```

另请参阅

选择语句
关键字
__if_not_exists 语句

Last updated on 2025/11/07

__if_not_exists 语句

`__if_not_exists` 语句测试指定的标识符是否存在。 如果该标识符不存在，则执行指定的语句块。

语法

```
__if_not_exists ( identifier ) {
statements
};
```

参数

identifier
要测试其存在性的标识符。

statements
如果标识符不存在，则执行一条或多条语句。

注解

> ⊗ **注意**
>
> 若要实现最可靠的结果，请使用以下约束下的 `__if_not_exists` 语句。

- 只将 `__if_not_exists` 语句应用于简单类型而不是模板。
- 将 `__if_not_exists` 语句应用于类的内部或外部的标识符。 不要将 `__if_not_exists` 语句应用于局部变量。
- 仅在函数的主体中使用 `__if_not_exists` 语句。 在函数主体的外部，`__if_not_exists` 语句仅能测试完全定义的类型。
- 在测试重载函数时，不能测试特定形式的重载。

`__if_not_exists` 语句的补充是 __if_exists 语句。

示例

有关如何使用 `__if_not_exists` 的示例，请参阅 __if_exists 语句。

另请参阅

选择语句
关键字
__if_exists 语句

Last updated on 2025/11/07

`switch` 语句 (C++)

允许根据整型表达式的值在多个代码段中进行选择。

语法

> *selection-statement*?
>
> **switch** (*init-statement* $_{\text{opt}}$ $^{\text{C++17}}$ *condition*) *statement*

> *init-statement*?
>
> *expression-statement*
>
> *simple-declaration*

> *condition*?
>
> *expression*
>
> *attribute-specifier-seq* $_{\text{opt}}$ *decl-specifier-seq* *declarator* *brace-or-equal-initializer*

> *labeled-statement*?
>
> **case** *constant-expression* **:** *statement*
>
> **default** **:** *statement*

备注

switch 语句使控件根据 *labeled-statement* 的值转移到其语句正文中的一个 *condition*。

condition 必须是整数型或明确转换为整数型的类类型。 整型提升按标准转换中所述进行。

switch 语句正文由一系列 **case** 标签和一个可选的 **default** 标签组成。 A *labeled-statement* 是其中一个标签和后面的语句。 标记语句不是语法需求，但如果它们不存在，**switch** 语句是无意义的。 *constant-expression* 语句中没有两个 **case** 值可能计算为相同的值。 **default** 标签只能出现一次。 **default** 语句通常放在末尾，但它可以出现在 **switch** 语句正文中的任何位置。 **case** 或 **default** 标签只能显示在 **switch** 语句内部。

每个 *constant-expression* 标签的 **case** 都转换为与类型 *condition* 相同的常量值。 然后，与 *condition* 比较相等情况。 控件将传递到 **case** *constant-expression* 值后的第一个语句，该语句与 *condition* 的值匹配。 下表中显示了生成的行为。

`switch` 语句行为

展开表

条件	操作
转换后的值与提升的控制表达式的值匹配。	控制将转移到跟在该标签后面的语句。
没有常量与 `case` 标签中的常量匹配；存在 `default` 标签。	控件将转移到 `default` 标签。
没有常量与 `case` 标签中的常量匹配；不存在 `default` 标签。	控件将转移到 `switch` 语句之后的语句。

如果找到匹配表达式，则可以继续执行 `case` 或 `default` 标签。 break 语句用于停止执行并将控制转移到 `switch` 语句之后的语句。 如果没有 `break` 语句，将执行从匹配的 `case` 标签到 `switch` 语句末尾之间的每个语句，包括 `default`。 例如：

C++

```
// switch_statement1.cpp
#include <stdio.h>

int main() {
   const char *buffer = "Any character stream";
   int uppercase_A, lowercase_a, other;
   char c;
   uppercase_A = lowercase_a = other = 0;

   while ( c = *buffer++ )   // Walks buffer until NULL
   {
      switch ( c )
      {
         case 'A':
            uppercase_A++;
            break;
         case 'a':
            lowercase_a++;
            break;
         default:
            other++;
      }
   }
   printf_s( "\nUppercase A: %d\nLowercase a: %d\nTotal: %d\n",
      uppercase_A, lowercase_a, (uppercase_A + lowercase_a + other) );
}
```

在上面的示例中，如果 `uppercase_A` 是大写 `c`，则 `'A'` 将递增。 `break` 之后的 `uppercase_A++` 语句会终止 `switch` 语句正文的执行并将控制转移到 `while` 循环。 如果没有 `break` 语句，执行将“落入”到下一个标记的语句，以便 `lowercase_a` 和 `other` 也会递增。 `break` 的 `case 'a'` 语句

也能达到类似目的。 如果 `c` 是小写的 `'a'`，则会递增 `lowercase_a`，并且 **`break`** 语句将终止 **`switch`** 语句正文。 如果 `c` 不是 `'a'` 或 `'A'`，则将执行 **`default`** 语句。

Visual Studio 2017 及更高版本（在 /std:c++17 模式和更高版本中可用）： 属性 `[[fallthrough]]` 在 C++17 标准中指定。 只能在 **`switch`** 语句中使用它。 这是提醒编译器或者任何读代码的人回退行为是故意的。 Microsoft C++ 编译器当前不会对回退行为发出警告，因此此属性对编译器行为没有影响。 在该示例中，属性将应用于未终止的标记语句中的空语句。 换句话说，需要分号。

C++

```
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 15.3 及更高版本（在 /std:c++17 模式和更高版本中提供）： 语句 **`switch`** 可能具有以分号结尾的 *`init-statement`* 子句。 它引入并初始化一个变量，该变量的作用域限制为语句块 **`switch`**：

C++

```
    switch (Gadget gadget(args); auto s = gadget.get_status())
    {
    case status::good:
        gadget.zip();
        break;
    case status::bad:
        throw BadGadget();
    }
```

`switch` 语句的内部块可以包含带有初始化的定义，前提是可以*访问*到它们 - 即所有可能的执行路径都不会绕过它们。 使用这些声明引入的名称具有局部范围。 例如：

C++

```
// 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;
    }
}
```

`switch` 语句可以嵌套。 嵌套时， `case` 或 `default` 标签将与封装它们的最近的 `switch` 语句关联。

Microsoft 特定行为

Microsoft C++ 未限制 `case` 语句中 `switch` 值的数量。 该数量仅受可用内存的限制。

另请参阅

选择语句
关键字

Last updated on 2025/11/07

迭代语句 (C++)

根据一些循环终止条件，迭代语句会导致语句（或复合语句）被执行零次或多次。 当这些语句是复合语句时，除非遇到 break 语句或 continue 语句，否则将按顺序执行它们。

C++ 提供四个迭代语句 - while、do、for 和 range-based for。 它们都将进行迭代环，直到其终止表达式的计算结果为零 (false)，或直到使用 `break` 语句强制执行循环终止。 下表汇总了这些语句及其操作；后面各节详细讨论了它们。

迭代语句

展开表

语句	计算位置	初始化	增量
`while`	循环的顶部	否	否
`do`	循环的底部	否	否
`for`	循环的顶部	是	是
基于范围的 for	循环的顶部	是	是

迭代语句的语句部分不能为声明。 但是，它可以是包含声明的复合语句。

另请参阅

语句概述

Last updated on 2025/11/07

While 语句 (C++)

重复执行语句，直到表达式计算结果为零。

语法

```
while ( expression )
   statement
```

备注

表达式的测试在每次执行循环之前开始，因此， 循环执行零次或多次`while`。 表达式必须是整型类型、指针类型或可以明确转换为整型或指针类型的类类型。

当执行语句正文中的 `while`、goto 或 return 时， 循环也可以终止。 使用 continue 可在不退出 `while` 循环的情况下终止迭代。 `continue` 将控制转移到 `while` 循环的下一次迭代。

以下代码使用 `while` 循环剪裁字符串中的尾随下划线：

C++

```
// 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));
}
```

终止条件在循环顶部进行评估。 如果没有尾随下划线，循环永远不会执行。

另请参阅

迭代语句
关键字
do-while 语句 (C++)
for 语句 (C++)
基于范围的 for 语句 (C++)

Last updated on 2025/11/07

do-while 语句 (C++)

反复执行 statement，直到指定的终止条件 (expression) 的计算结果为零。

语法

```
do
   statement
while ( expression ) ;
```

备注

终止条件的测试将在每次执行循环后进行；因此 do-while 循环将执行一次或多次，具体取决于终止表达式的值。 do-while 语句还可在语句体中执行 break、goto 或 return 语句时终止。

expression 必须具有算法或指针类型。 执行过程如下所示：

1. 执行语句体。
2. 接着，计算 expression。 如果 expression 为 false，则 do-while 语句将终止，控制权将传递到程序中的下一条语句。 如果 expression 为 true（非零），则将从第 1 步开始重复此过程。

示例

以下示例演示了 do-while 语句：

C++

```cpp
// do_while_statement.cpp
#include <stdio.h>
int main()
{
    int i = 0;
    do
    {
        printf_s("\n%d",i++);
    } while (i < 3);
}
```

另请参阅

迭代语句
关键字
While 语句 (C++)
for 语句 (C++)
基于范围的 for 语句 (C++)

Last updated on 2025/11/07

`for` 语句 （C++）

重复执行语句，直到条件变为 false。 有关基于`for`范围的语句的信息，请参阅基于 for Range 的语句（C++）。 有关 C++/CLI `for each` 语句的信息，请参阅for eachin：

语法

> `for` (*`init-expression`* ; *`cond-expression`* ; *`loop-expression`*)
> *`statement`*

注解

使用 `for` 语句构造必须执行指定次数的循环。

该 `for` 语句由三个可选部分组成，如下表所示。

for 循环元素

展开表

语法名称	执行时	DESCRIPTION
`init-expression`	在语句的任何其他元素 `for` 之前，*`init-expression`* 只执行一次。 然后控制传递给 *`cond-expression`*.	通常用于初始化循环索引。 它可以包含表达式或声明。
`cond-expression`	在执行每个迭代 *`statement`*之前，包括第一次迭代。 仅在计算结果为 true（非零）时才 执行<	一个表达式，该表达式的计算结果为整型类型或类类型，该类类型与整型类型有明确转换。 通常用于测试循环终止条件。
`loop-expression`	每次迭代 *`statement`*结束时。 执行后 *`loop-expression`* ， *`cond-expression`* 将评估。	通常用于递增循环索引。

以下示例演示了 `for` 使用语句的不同方法。

C++

```
#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: 01
}
```

`init-expression` 可以 `loop-expression` 包含用逗号分隔的多个语句。 例如：

C++

```
#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` 可以通过其他方式递增或递减或修改。

C++

```
#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
```

执行内部循环break时，循环`for`将终止（返回或goto（到循环外部`for`标记的语句）。`statement` continue循环中的`for`语句仅终止当前迭代。

如果 `cond-expression` 省略，则将其视为 `true`循环，并且 `for` 循环不会终止，没有 a `break`、`return`或 `goto` 内部 `statement`。

尽管语句的 `for` 三个字段通常用于初始化、测试终止和递增，但它们不限于这些用途。 例如，下面的代码打印数字 0 到 4。 在本例中， `statement` 为 null 语句：

C++

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

int main()
{
    int i;
    for( i = 0; i < 5; cout << i << '\n', i++){
        ;
    }
}
```

`for` 循环和 C++ Standard

C++标准表示，循环中 `for` 声明的变量应在循环结束后超出范围 `for` 。 例如：

C++

```
for (int i = 0 ; i < 5 ; i++) {
   // do something
}
// i is now out of scope under /Za or /Zc:forScope
```

默认情况下，在 /Ze循环中 `for` 声明的变量将保留在范围内，直到 `for` 循环的封闭范围结束。

/Zc:forScope 启用在 for 循环中声明的变量的标准行为，而无需指定 `/Za`。

还可以按如下所示使用循环的范围差异 `for` 重新声明变量 `/Ze` ：

C++

```
// 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++ ) {}
}
```

此行为更接近于模拟循环中 `for` 声明的变量的标准行为，这要求在循环完成后在循环中 `for` 声明的变量超出范围。 在循环中 `for` 声明变量时，编译器会在内部将其提升为循环封闭范围内的局部变量 `for` 。 即使已经有具有相同名称的局部变量，也会升级它。

另请参阅

迭代语句
关键字
while 语句 （C++）
do-while 语句 （C++）
基于 for 范围的语句（C++）

Last updated on 2025/11/07

基于范围的 for 语句 (C++)

对 `statement` 中的每个元素按顺序重复执行 `expression`。

语法

> **`for (`** *for-range-declaration* **`:`** *表达* **`)`**
> *语句*

备注

使用基于范围的 **`for`** 语句构造必须在“范围”中执行的循环，它定义为可循环访问的任何内容 - 例如，`std::vector` 或其范围由 `begin()` 和 `end()` 定义的任何其他 C++ 标准库序列。 `for-range-declaration` 部分中声明的名称是 **`for`** 语句的本地名称，且无法在 `expression` 或 `statement` 中重新声明它。 请注意，在语句的 auto 部分中，`for-range-declaration` 关键字是首选的。

Visual Studio 2017 中的新增功能：基于范围的 **`for`** 循环不再需要 `begin()` 和 `end()` 返回相同类型的对象。 这使得 `end()` 能够返回类似于 Ranges-V3 方案中定义的范围所使用的那种 sentinel 对象。 有关详细信息，请参阅通用化基于范围的 For 循环 和 GitHub 上的 range-v3 库 。

此代码说明了如何使用基于范围的 **`for`** 循环来循环访问数组和矢量：

C++

```
// 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;
}
```

输出如下：

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
```

在 `for` 中执行下列项之一时，基于范围的 `statement` 循环将终止：针对基于范围的 break 循环外部的标记语句的 return、goto 或 `for`。 基于范围的 continue 循环中的 `for` 语句仅终止当前迭代。

请记住这些有关基于范围的 `for` 的情况：

- 自动识别数组。

- 识别拥有 `.begin()` 和 `.end()` 的容器。
- 对于任何其他内容，使用依赖于自变量的查找 `begin()` 和 `end()`。

另请参阅

auto
迭代语句
关键字
while语句 (C++)
do-while语句 (C++)
for语句 (C++)

Last updated on 2025/11/07

跳转语句 (C++)

C++ 跳转语句执行控制的即时本地转换。

语法

```
break;
continue;
return [expression];
goto identifier;
```

备注

有关 C++ 跳转语句的说明，请参阅下列主题。

- break 语句
- continue 语句
- return 语句
- goTo 语句

另请参阅

语句概述

Last updated on 2025/11/07

`break` 语句 (C++)

`break` 语句可终止执行最近的封闭循环或其所在条件语句。 控制权将传递给该语句结束之后的语句（如果有的话）。

语法

C++

```
break;
```

备注

`break` 语句与条件 switch 语句以及 do、for 和 while 循环语句一起使用。

在 `switch` 语句中， `break` 语句将导致程序执行 `switch` 语句之外的下一语句。 如果没有 `break` 语句，将执行从匹配的 `case` 标签到 `switch` 语句末尾之间的每个语句，其中包括 `default` 子句。

在循环中， `break` 语句将终止执行最近的 `do`、 `for` 或 `while` 封闭语句。 控制权将传递给终止语句之后的语句（如果有的话）。

在嵌套语句中， `break` 语句只终止直接包围它的 `do`、 `for`、 `switch` 或 `while` 语句。 你可以使用 `return` 或 `goto` 语句从较深嵌套的结构转移控制权。

示例

以下代码演示如何在 `break` 循环中使用 `for` 语句。

C++

```
#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';
    }
}
```

Output

```
1
2
3
1
2
3
```

以下代码演示如何在 `break` 循环和 `while` 循环中使用 `do`。

C++

```
#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);
}
```

Output

```
0
1
2
3
0
1
2
3
```

以下代码演示如何在 switch 语句中使用 `break`。 如果你要分别处理每个用例，则必须在每个用例中使用 `break`；如果不使用 `break`，则执行下一用例中的代码。

C++

```cpp
#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";
    }
}
```

另请参阅

跳转语句
关键字
continue 语句

Last updated on 2025/11/07

continue 语句 (C++)

强制转移对最小封闭 do、for 或 while 循环的控制表达式的控制。

语法

```
continue;
```

备注

将不会执行当前迭代中的所有剩余语句。 确定循环的下一次迭代，如下所示：

- 在 `do` 或 `while` 循环中，下一个迭代首先会重新计算 `do` 或 `while` 语句的控制表达式。
- 在 `for` 循环中（使用语法 `for( <init-expr> ; <cond-expr> ; <loop-expr> )`），将执行 `<loop-expr>` 子句。 然后，重新计算 `<cond-expr>` 子句，并根据结果确定该循环结束还是进行另一个迭代。

下面的示例演示了如何使用 `continue` 语句跳过代码部分并启动循环的下一个迭代。

示例

C++

```
// 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");
}
```

Output

```
before the continue
before the continue
before the continue
after the do loop
```

另请参阅

跳转语句
关键字

Last updated on 2025/11/07

return 语句 (C++)

终止函数的执行并返回对调用函数的控制（或对操作系统的控制，如果您从 `main` 函数转移控制）。 紧接在调用之后在调用函数中恢复执行。

语法

```
return [expression];
```

备注

`expression` 子句（如果存在）将转换为函数声明中指定的类型，就像正在执行初始化一样。 从该类型的表达式到 `return` 类型的函数的转换会创建临时对象。 有关如何以及何时创建临时对象的详细信息，请参阅临时对象。

`expression` 子句的值将返回调用函数。 如果省略该表达式，则函数的返回值是不确定的。 构造函数和析构函数以及 `void` 类型的函数无法在 `return` 语句中指定表达式。 所有其他类型的函数必须在 `return` 语句中指定表达式。

当控制流退出封闭函数定义的块时，结果将与执行不带表达式的 `return` 语句所获得的结果一样。 这对于声明为返回值的函数无效。

一个函数可以包含任意数量的 `return` 语句。

以下示例将一个表达式与 `return` 语句一起使用来获取两个整数中的最大者。

示例

C++

```
// 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 ));
}
```

另请参阅

跳转语句
关键字

Last updated on 2025/11/07

goto 语句 (C++)

`goto` 语句无条件地将控制转移到由指定标识符标记的语句。

语法

```
goto identifier;
```

备注

由 `identifier` 指定的标记语句必须位于当前函数中。 所有 `identifier` 名称都是内部命名空间的成员，因此不会干扰其他标识符。

语句标签仅对 `goto` 语句有意义；否则，将忽略语句标签。 不能重新声明标签。

不允许 `goto` 语句将控制转移到跳过该位置范围内任何变量的初始化的位置。 以下示例引发 C2362：

C++

```
int goto_fn(bool b)
{
    if (!b)
    {
        goto exit;  // C2362
    }
    else
    { /*...*/ }

    int error_code = 42;

exit:
    return error_code;
}
```

有一种较好的编程风格，是尽可能使用 `break`、 `continue` 和 `return` 语句而不是 `goto` 语句。 但是，由于 `break` 语句仅退出循环的一个级别，可能必须使用 `goto` 语句来退出深度嵌套的循环。

有关标签和 `goto` 语句的详细信息，请参阅带标签的语句。

示例

在此示例中，当 `goto` 等于 3 时，`stop` 语句将控制转移到标记为 `i` 的点。

C++

```
// 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 );
}
```

Output

```
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
```

另请参阅

跳转语句
关键字

Last updated on 2025/11/07

控制的转移

可以在 `goto` 语句中使用 `case` 语句或 `switch` 标签来指定分支超出初始值设定项的程序。 此类代码是非法的，除非包含初始值设定项的声明在跳转语句发生的块所封闭的块中。

下面的示例显示了声明和初始化对象 `total`、 `ch` 和 `i` 的循环。 也存在将控制权传递过初始值设定项的错误 `goto` 语句。

C++

```
// 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;
   }
}
```

在前面的示例中， `goto` 语句尝试将控制权传递过 `i` 的初始化。 但是，如果已声明但未初始化 `i`，则该传递是合法的。

在用作 `total` 语句的 statement 的块中声明的对象 `ch` 和 在使用 `while` 语句退出此块时将被销毁`break`。

Last updated on 2025/11/07

命名空间 (C++)

命名空间是一个声明性区域，为其内部的标识符（类型、函数和变量等的名称）提供一个范围。 命名空间用于将代码组织到逻辑组中，还可用于避免名称冲突，尤其是在基本代码包括多个库时。 命名空间范围内的所有标识符彼此可见，而没有任何限制。 命名空间之外的标识符可通过使用每个标识符的完全限定名（例如 `std::vector<std::string> vec;`）来访问成员，也可通过单个标识符的 using 声明（`using std::string`）或命名空间中所有标识符的 using 指令（`using namespace std;`）来访问成员。 头文件中的代码应始终使用完全限定的命名空间名称。

下面的示例演示了一个命名空间声明和命名空间之外的代码可访问其成员的三种方法。

C++

```
namespace ContosoData
{
    class ObjectManager
    {
    public:
        void DoSomething() {}
    };
    void Func(ObjectManager) {}
}
```

使用完全限定名：

C++

```
ContosoData::ObjectManager mgr;
mgr.DoSomething();
ContosoData::Func(mgr);
```

使用 using 声明，以将一个标识符引入范围：

C++

```
using ContosoData::ObjectManager;
ObjectManager mgr;
mgr.DoSomething();
```

使用 using 指令，以将命名空间中的所有内容引入范围：

C++

```
using namespace ContosoData;

ObjectManager mgr;
```

```
mgr.DoSomething();
Func(mgr);
```

using 指令

通过 `using` 指令，可使用 `namespace` 中的所有名称，而不需要 namespace-name 为显式限定符。 如果在一个命名空间中使用多个不同的标识符，则在实现文件中使用 using 指令（即 *.cpp）；如果仅使用一个或两个标识符，则考虑 using 声明，以仅将这些标识符而不是命名空间中的所有标识符引入范围。 如果本地变量的名称与命名空间变量的名称相同，则隐藏命名空间变量。 使命名空间变量具有与全局变量相同的名称是错误的。

> **注意**
>
> using 指令可以放置在 .cpp 文件的顶部（在文件范围内），或放置在类或函数定义内。
>
> 一般情况下，避免将 using 指令放置在头文件 (*.h) 中，因为任何包含该标头的文件都会将命名空间中的所有内容引入范围，这将导致非常难以调试的名称隐藏和名称冲突问题。 在头文件中，始终使用完全限定名。 如果这些名称太长，可以使用命名空间别名将其缩短。（请参阅下文。）

声明命名空间和命名空间成员

通常情况下，在头文件中声明一个命名空间。 如果函数实现位于一个单独的文件中，则限定函数名称，如本示例所示。

C++

```cpp
// contosoData.h
#pragma once
namespace ContosoDataServer
{
    void Foo();
    int Bar();
}
```

contosodata.cpp 中的函数实现应使用完全限定名，即使将一个 `using` 指令放置于文件的顶部也是如此：

C++

```cpp
#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;}
```

可以在单个文件中的多个块中声明命名空间，也可在多个文件中声明命名空间。 编译器在预处理过程中将各部分联接在一起，产生的命名空间中包含所有部分中声明的所有成员。 一个相关示例是在标准库中的每个头文件中声明的 std 命名空间。

指定的命名空间的成员可以在定义的名称的显式限定所声明的命名空间的外部进行定义。 但是，定义必须出现在命名空间中的声明位置之后，该命名空间包含在声明的命名空间中。 例如：

C++

```
// 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();
}
```

当跨多个头文件声明命名空间成员，并且未以正确的顺序包含这些标头时，可能出现此错误。

全局命名空间

如果未在显式命名空间中声明某个标识符，则该标识符属于隐式全局命名空间的一部分。 通常情况下，如果可能，尝试避免在全局范围内进行声明，入口点 main 函数除外，它必须位于全局命名空间中。 若要显式限定全局标识符，请使用没有名称的范围解析运算符，如 `::SomeFunction(x);` 中所示。 这将使标识符与任何其他命名空间中具有相同名称的任何内容区分开来，并且还有助于使其他人更轻松地了解你的代码。

Std 命名空间

所有 C++ 标准库类型和函数都在 `std` 命名空间或嵌套在 `std` 内的命名空间中进行声明。

嵌套命名空间

可以嵌套命名空间。 普通的嵌套命名空间具有对其父级成员的非限定访问权限，而父成员不具有对嵌套命名空间的非限定访问权限（除非它被声明为内联），如下面的示例所示：

C++

```
namespace ContosoDataServer
{
    void Foo();

    namespace Details
    {
        int CountImpl;
        void Ban() { return Foo(); }
    }

    int Bar(){...};
    int Baz(int i) { return Details::CountImpl; }
}
```

普通嵌套命名空间可用于封装不属于父命名空间的公共接口的一部分的内部实现详细信息。

内联命名空间 (C++11)

与普通嵌套命名空间不同，内联命名空间的成员会被视为父命名空间的成员。 这一特性使针对重载函数的依赖于自变量的查找可以对父命名空间和嵌套内联命名空间中具有重载的函数起作用。 它还可让你在内联命名空间中声明的模板的父命名空间中声明专用化。 下面的示例演示在默认情况下，外部代码如何绑定到内联命名空间：

C++

```
// 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"); }
    }
}

// main.cpp
```

```
#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;
}
```

下面的示例演示如何在内联命名空间中声明的模板的父命名空间中声明专用化：

C++

```
namespace Parent
{
    inline namespace new_ns
    {
         template <typename T>
         struct C
         {
             T member;
         };
    }
     template<>
     class C<int> {};
}
```

可以将内联命名空间用作版本控制机制，以管理对库的公共接口的更改。 例如，可以创建单个父命名空间，并将接口的每个版本封装到嵌套在父命名空间内的其自己的命名空间中。 保留最新或首选的版本的命名空间限定为内联，并因此以父命名空间的直接成员的形式公开。 调用 Parent::Class 的客户端代码将自动绑定到新代码。 通过使用指向包含该代码的嵌套命名空间的完全限定路径，选择使用较旧版本的客户端仍可以对其进行访问。

Inline 关键字必须应用到编译单元中命名空间的第一个声明中。

下面的示例演示一个接口的两个版本，每个版本位于一个嵌套命名空间中。 通过 `v_20` 接口对 `v_10` 命名空间进行了某些修改，且该命名空间被标记为内联。 使用新库并调用 `Contoso::Funcs::Add` 的客户端代码将调用 v_20 版本。 尝试调用 `Contoso::Funcs::Divide` 的代码现在将获取一个编译时错误。 如果它们确实需要该函数，则仍可以通过显式调用 `v_10` 访问 `Contoso::v_10::Funcs::Divide` 版本。

C++

```
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);
        };
    }
}
```

命名空间别名

命名空间名称必须是唯一的，这意味着通常它们不应太短。 如果名称的长度使代码难以阅读，或在不能使用 using 指令的头文件中进行键入单调乏味，则可以使用用作实际名称的缩写的命名空间别名。 例如：

C++

```
namespace a_very_long_namespace_name { class Foo {}; }
namespace AVLNN = a_very_long_namespace_name;
void Bar(AVLNN::Foo foo){ }
```

匿名或未命名的命名空间

可以创建显式命名空间，但不为其提供一个名称：

C++

```
namespace
{
    int MyFunc(){}
}
```

这称为未命名的命名空间或匿名命名空间，在你想要使变量声明对于其他文件中的代码不可见（即为它们提供内部链接），而不必创建已命名的命名空间时非常有用。 同一文件中的所有代码都可以看到未命名的命名空间中的标识符，但这些标识符以及命名空间本身在该文件外部（或更准确地说，在翻译单元外部）不可见。

另请参阅

声明和定义

Last updated on 2025/11/07

枚举（C++）

枚举是一种用户定义的类型，它由一组称为 *枚举数*的命名整型常量组成。

> **注意**
>
> 本文介绍 ISO 标准C++语言 `enum` 类型和C++11 中引入的范围（或强类型） `enum class` 类型。 有关 C++/CLI 和 C++/CX 中的或`private enum class`类型的信息`public enum class`，请参阅enum class（C++/CLI 和 C++/CX）。

语法

enum-name:

  identifier

enum-specifier:

  enum-head **{** *enumerator-list* 选择 **}**

  enum-head **{** *enumerator-list* **,** **}**

enum-head:

  enum-key *attribute-specifier-seq* 选择 *enum-head-name* 选择 *enum-base* 选择

enum-head-name:

  nested-name-specifier 选择 *identifier*

opaque-enum-declaration:

  enum-key *attribute-specifier-seq* 选择 *enum-head-name* *enum-base* 选择 **;**

enum-key:

  enum

  enum class

  enum struct

enum-base:

  : *type-specifier-seq*

enumerator-list:

  enumerator-definition

  enumerator-list **,** *enumerator-definition*

`enumerator-definition`:

`enumerator`

`enumerator` `=` `constant-expression`

`enumerator`:

`identifier` `attribute-specifier-seq` 选择

用法

```
C++
// 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
```

参数

`identifier`

为枚举提供的类型名称。

`type`

枚举器的基础类型;所有枚举器具有相同的基础类型。 可以是任何整型。

`enum-list`

枚举中的枚举器的逗号分隔列表。 范围中的每个枚举器或变量名称都必须是唯一的。 但是，这些值可以重复。 在未范围枚举中，范围是周围的范围;在作用域枚举中，范围本身。 `enum-list` 在作用域枚举中，列表可能为空，实际上定义了一个新的整型类型。

`class`

通过在声明中使用此关键字，可以指定枚举的范围，并且必须提供枚举 `identifier` 。 还可以使用 `struct` 关键字代替 `class`，因为它们在此上下文中具有语义上的等效性。

枚举器范围

枚举提供上下文来描述表示为命名常量的值范围。 这些命名常量也称为 *枚举器*。 在原始 C 和 C++类型中，未限定的 `enum` 枚举器在整个声明范围 `enum` 中可见。 在作用域枚举中，枚举器名称必须由类型名称限定 `enum` 。 以下示例演示了这两种枚举之间的基本差异：

C++

```
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
        { /*...*/
        }
    }
}
```

枚举中的每个名称都分配一个整数值，该值对应于枚举中值的顺序。 默认情况下，第一个值分配为 0、下一个值分配 1 等，但可以显式设置枚举器的值，如下所示：

C++

```
enum Suit { Diamonds = 1, Hearts, Clubs, Spades };
```

为枚举器 `Diamonds` 赋值 `1`。 如果后续枚举器未指定显式值，则接收上一个枚举器的值加上一个。 在前面的示例中， `Hearts` 值为 2、 `Clubs` 将具有 3 等。

每个枚举器都被视为常量，并且必须在定义范围（对于未作用域枚举）或自身范围内`enum`（对于作用域枚举）内具有唯一的名称`enum`。 提供给名称的值不必是唯一的。 例如，请考虑此未作用域枚举 `Suit`的声明：

C++

```
enum Suit { Diamonds = 5, Hearts, Clubs = 4, Spades };
```

`Diamonds` `Hearts`、、 `Clubs`和 `Spades` 5 的值分别为 5、6、4 和 5。 请注意，5 多次使用;即使它可能不是预期，它也是允许的。 对于作用域枚举，这些规则是相同的。

强制转换规则

可以隐式转换为无作用域枚举常量，但`int`永远不会隐式转换为`int`枚举值。 以下示例演示了尝试分配 `hand` 非 `Suit`值时会发生什么情况：

C++

```
int account_num = 135692;
Suit hand;
hand = account_num; // error C2440: '=' : cannot convert from 'int' to 'Suit'
```

强制转换需要转换为 `int` 作用域内或无作用域枚举器。 但是，可以将未范围枚举器提升为没有强制转换的整数值。

C++

```
int account_num = Hearts; //OK if Hearts is in an unscoped enum
```

以这种方式使用隐式转换可能会导致意外的副作用。 为了帮助消除与未范围枚举关联的编程错误，范围枚举值是强类型化的。 作用域枚举器必须由枚举类型名称（标识符）限定，不能隐式转换，如以下示例所示：

C++

```
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
    }
}
```

请注意，该行 `hand = account_num;` 仍然会导致出现未范围枚举的错误，如前所述。 允许使用显式强制转换。 但是，对于作用域枚举，下一个语句 `account_num = Suit::Hearts;` 中的尝试转换不再允许在没有显式强制转换的情况下使用。

没有枚举器的枚举

Visual Studio 2017 版本 15.3 及更高版本 （可用于 /std:c++17 及更高版本）： 通过定义具有显式基础类型且没有枚举器的枚举（常规或作用域），实际上可以引入没有隐式转换到任何其他类型的新整型类型。 通过使用此类型而不是其内置基础类型，可以消除因无意隐式转换而导致的细微错误的可能性。

C++

```
enum class byte : unsigned char { };
```

新类型是基础类型的精确副本，因此具有相同的调用约定，这意味着它可以在 API 之间使用，而不会造成任何性能损失。 使用直接列表初始化初始化类型变量时，不需要强制转换。 以下示例演示如何在各种上下文中初始化没有枚举器的枚举：

C++

```
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;
}
```

另请参阅

C 枚举声明
关键字

Last updated on 2025/11/07

`union`

项目 • 2024/09/25

> **备注**
>
> 在 C++17 及更高版本中， `std::variant` class 是 union 的一个类型安全替代项。

`union` 是一个用户定义类型，其中所有成员都共享同一个内存位置。 此定义意味着在任何给定时间，union 都不能包含来自其成员列表的多个对象。 这还意味着，无论 union 具有多少成员，它始终仅使用足以存储最大成员的内存。

如果具有大量对象但内存有限，union 可用于节省内存。 但是，需要格外小心才能正确使用 union。 你负责确保始终访问分配的同一成员。 如果任何成员类型具有不常用构造函数，则必须编写代码来显式构造和销毁该成员。 使用 union 之前，应考虑是否可以使用基 class 和派生的 class 类型来更好地表示尝试解决的问题。

语法

> `union` *`tag`* opt `{` *`member-list`* `};`

参数

`tag`
为 union 提供的类型名称。

`member-list`
union 可以包含的成员。

声明 union

利用 `union` 关键字开始 union 的声明，并用大括号包含成员列表：

```C++
// 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
}
```

使用 union

在上面的示例中，任何访问 union 的代码都需要了解保存数据的成员。 解决此问题的最常见解决方案被称为“可区分 union”。 它将 union 括在 struct 中，还加入一个 enum 成员，用于指示当前存储在 union 中的成员类型。 下面的示例演示了基本模式：

C++

```
#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) {}

void Initialize(std::queue<Input>& inputs)
{
    Input first;
    first.type = WeatherDataType::Temperature;
    first.temp = { 101, 1418855664, 91.8, 108.5, 67.2 };
    inputs.push(first);

    Input second;
    second.type = WeatherDataType::Wind;
    second.wind = { 204, 1418859354, 14, 27 };
    inputs.push(second);
}

int main(int argc, char* argv[])
{
    // Container for all the data records
    queue<Input> inputs;
    Initialize(inputs);
    while (!inputs.empty())
    {
        Input const 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;
}
```

在上面的示例中，`Input` struct 中的 union 没有名称，因此被称为“匿名 union”。 它的成员是可以直接访问的，就像它们是 struct 的成员一样。 有关如何使用匿名 union 的详细信息，请参阅“匿名 union”部分。

上面的示例展示的问题也可以通过以下方法解决：使用派生自公共基 class 的 class 类型。 基于容器中每个对象的运行时类型对代码进行分支。 代码将更易于维护和理解，但是也可能比使用 union 更慢。 此外，通过 union 可以存储不相关的类型。 union 让你可以动态地更改存储值的类型，而无需更改 union 变量本身的类型。 例如，可以创建其元素存储不同类型的不同值的 `MyUnionType` 异类数组。

很容易误用示例中的 `Input` struct。 由用户负责正确使用鉴别器来访问保存数据的成员。 你可以通过使 union 成为 **`private`** 并提供特殊访问函数（如下一个示例所示）来防止误用。

不受限制的 union (C++11)

在 C++03 及更低版本中，union 可以包含具有 class 类型的非静态数据成员，只要该类型没有用户提供的构造函数、析构函数或赋值运算符即可。 在 C++11 中，消除了这些限制。 如果在 union 中包含这样一个成员，编译器会自动将不是用户提供的任何特殊成员函数标记为 **`deleted`**。 如果 union 是 class 或 struct 中的匿名 union，则 class 或 struct 的非用户提供的任何特殊成员函数都会标记为 **`deleted`**。 以下示例展示了如何处理此情况。 union 的某个成员具有需要此特殊处理的成员：

C++

```
// 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;
}
```

union 无法存储引用。 union 也不支持继承。 这意味着不能使用 union 作为基 class，也不能继承自另一个 class 或具有虚拟函数。

初始化 union

可以通过分配包含在括号中的表达式，在相同语句中声明并初始化 union。 计算该表达式并将其分配给 union 的第一个字段。

C++

```
#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
*/
```

`NumericType` union 排列在内存中（从概念上讲），如下图所示：

匿名 union

匿名 union 是在没有 `class-name` 或 `declarator-list` 的情况下声明的。

> `union { member-list }`

匿名 union 中声明的名称可直接使用，就像非成员变量一样。 这意味着匿名 union 中声明的名称必须在周边范围中是唯一的。

匿名 union 受以下限制的约束：

- 如果是在文件或命名空间范围内声明的，还必须将其声明为 `static`。
- 它只能有 `public` 成员；在一个匿名的 union 中拥有 `private` 成员和 `protected` 成员会产生错误。
- 它不能具有成员函数。

另请参阅

类和结构
关键字
class
struct

反馈

此页面是否有帮助？ 是 否

提供产品反馈 | 在 Microsoft Q&A 获取帮助

函数 (C++)

函数是执行某种操作的代码块。 函数可以选择性地定义使调用方可以将实参传递到函数中的输入形参。 函数可以选择性地返回值作为输出。 函数可用于在单个可重用块中封装常用操作（理想情况是使用可清晰地描述函数行为的名称）。 以下函数从调用方接受两个整数并返回其总和；a 和 b 是 类型的参数`int`。

```
C++
int sum(int a, int b)
{
    return a + b;
}
```

可以从程序中任意数量的位置调用函数。 传递给函数的值是自变量，其类型必须与函数定义中的参数类型兼容。

```
C++
int main()
{
    int i = sum(10, 32);
    int j = sum(i, 66);
    cout << "The value of j is" << j << endl; // 108
}
```

对于函数长度没有实际限制，不过良好的设计应以执行单个明确定义的任务的函数为目标。 复杂算法应尽可能分解成易于理解的更简单函数。

在类范围中定义的函数称为成员函数。 在 C++ 中（与其他语言不同），函数还可以在命名空间范围中定义（包括隐式全局命名空间）。 这类函数称为*free 函数*或*非成员函数*；它们在标准库中广泛使用。

函数可能会重载，这意味着如果函数的不同版本在形参的数量和/或类型上有所不同，它们可能会共享相同的名称。 有关详细信息，请参阅函数重载。

函数声明的各个部分

最小函数*声明*包含返回类型、函数名和参数列表（可能为空），以及向编译器提供更多说明的可选关键字。 以下示例是函数声明：

```
C++
int sum(int a, int b);
```

函数定义包含声明以及函数体（这是大括号之间的所有代码）：

C++

```
int sum(int a, int b)
{
    return a + b;
}
```

后接分号的函数声明可以出现在程序中的多个位置处。 它必须在每个翻译单元中对该函数的任何调用之前出现。 根据单个定义规则 (ODR)，函数定义必须仅在程序中出现一次。

函数声明的必需部分有：

1. 返回类型，指定函数将返回的值的类型，如果不返回任何值，则为 `void`。 在 C++11 中，`auto` 是有效返回类型，可指示编译器从返回语句推断类型。 在 C++14 中，还允许使用 `decltype(auto)`。 有关详细信息，请参阅下面的“返回类型中的类型推导”。

2. 函数名，必须以字母或下划线开头，不能包含空格。 一般而言，标准库函数名中的前导下划线指示私有成员函数，或不是供你的代码使用的非成员函数。

3. 参数列表（一组用大括号限定、逗号分隔的零个或多个参数），指定类型以及可以用于在函数体内访问值的可选局部变量名。

函数声明的可选部分有：

1. `constexpr`，指示函数的返回值是常量值，可以在编译时进行计算。

   C++

   ```
   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. 其链接规范，`extern` 或 `static`。

   C++

   ```
   //Declare printf with C linkage.
   extern "C" int printf( const char *fmt, ... );
   ```

   有关详细信息，请参阅翻译单元和链接。

3. `inline`，指示编译器将对函数的每个调用替换为函数代码本身。 在某个函数快速执行并且在性能关键代码段中重复调用的情况下，内联可以帮助提高性能。

   C++
   ```
   inline double Account::GetBalance()
   {
       return balance;
   }
   ```

   有关详细信息，请参阅内联函数。

4. `noexcept` 表达式，指定函数是否可以引发异常。 在下面的示例中，函数在`is_pod`表达式计算结果为`true`时不引发异常。

   C++
   ```
   #include <type_traits>

   template <typename T>
   T copy_object(T& obj) noexcept(std::is_pod<T>) {...}
   ```

   有关详细信息，请参阅 noexcept。

5. （仅限成员函数）cv 限定符，指定函数是 `const` 还是 `volatile`。

6. （仅限成员函数）`virtual`、`override` 或 `final`。 `virtual`指定可以在派生类中重写函数。 `override` 表示派生类中的函数在重写虚函数。 `final`表示函数不能在任何进一步的派生类中进行重写。 有关详细信息，请参阅虚函数。

7. （仅限成员函数）应用于成员函数的`static`表示函数不与类的任何对象实例关联。

8. （仅限非静态成员函数）ref 限定符，向编译器指定隐式对象参数（`*this`）是右值引用与左值引用时要选择的函数重载。 有关详细信息，请参阅函数重载。

函数定义

函数定义由声明和函数体组成，括在大括号中，其中包含变量声明、语句和表达式。 以下示例显示了完整的函数定义：

C++
```
int foo(int i, std::string s)
{
   int value {i};
   MyClass mc;
```

```
        if(strcmp(s, "default") != 0)
        {
             value = mc.do_something(i);
        }
        return value;
    }
```

函数体内声明的变量称为局部变量。 它们会在函数退出时超出范围；因此，函数应永远不返回对局部变量的引用！

C++

```
    MyClass& boom(int i, std::string s)
    {
        int value {i};
        MyClass mc;
        mc.Initialize(i,s);
        return mc;
    }
```

const 和 constexpr 函数

可以将成员函数声明为 `const` 以指定该函数不允许更改类中任何数据成员的值。 通过将成员函数声明为 `const`，可以帮助编译器强制执行 const-correctness。 如果有人错误地尝试使用声明为 `const` 的函数来修改对象，则会引发编译器错误。 有关详细信息，请参阅 const。

如果函数生成的值可在编译时确定，则将函数声明为 `constexpr`。 constexpr 函数通常比常规函数执行速度更快。 有关详细信息，请参阅 constexpr。

函数模板

函数模板类似于类模板；它基于模板自变量生成具体功能。 在许多情况下，模板能够推断类型参数，因此无需显式指定它们。

C++

```
template<typename Lhs, typename Rhs>
auto Add2(const Lhs& lhs, const Rhs& 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
```

有关详细信息，请参阅函数模板

函数形参和实参

函数具有零种或多种类型的逗号分隔参数列表，其中每个参数都具有可以用于在函数体内访问它的名称。 函数模板可以指定更多类型或值参数。 调用方传递实参（其类型与形参列表兼容的具体值）。

默认情况下，参数通过值传递给函数，这意味着函数会收到所传递的对象的副本。 对于大型对象，创建副本可能成本高昂，并非始终必要。 若要使自变量通过引用（特别是左值引用）进行传递，请向参数添加引用限定符：

```
C++
void DoSomething(std::string& input){...}
```

当函数修改通过引用传递的参数时，它会修改原始对象，而不是本地副本。 若要防止函数修改这类实参，请将形参限定为 const&：

```
C++
void DoSomething(const std::string& input){...}
```

C++ 11：若要显式处理通过右值引用或通过左值引用传递的自变量，请对参数使用双与号以指示通用引用：

```
C++
void DoSomething(const std::string&& input){...}
```

只要关键字 `void` 是自变量声明列表中的第一个也是唯一一个成员，那么在参数声明列表中使用单个关键字 `void` 声明的函数就没有自变量。 列表中的其他地方的 `void` 类型的自变量产生错误。 例如：

```
C++
// OK same as GetTickCount()
long GetTickCount( void );
```

虽然指定`void`自变量是非法（此处所述的除外），但派生自类型`void`的类型（如指向`void`的指针和`void`的数组）可以出现在自变量声明列表中的任何位置。

默认自变量

函数签名中的最后一个或几个参数可能会分配有默认自变量，这意味着调用方可能会在调用函数时省略自变量（除非要指定某个其他值）。

C++

```
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)
{...}
```

有关详细信息，请参阅默认自变量。

函数返回类型

函数可能不会返回另一个函数或内置数组；但是，它可以返回指向这些类型的指针，或生成函数对象的 lambda。 除了这些情况，函数可以返回处于范围内的任何类型的值，或者它可以返回任何值（在这种情况下返回类型是 `void`）。

结尾返回类型

“普通”返回类型位于函数签名左侧。 *结尾返回类型*位于签名的最右侧，前面带有`->`运算符。 当返回值的类型取决于模板参数时，结尾返回类型在函数模板中尤其有用。

C++

```
template<typename Lhs, typename Rhs>
auto Add(const Lhs& lhs, const Rhs& rhs) -> decltype(lhs + rhs)
{
    return lhs + rhs;
}
```

当`auto`与结尾返回类型结合使用时，它对于 decltype 表达式生成的任何内容都只用作占位符，本身不执行类型推导。

函数局部变量

在函数主体内声明的变量称为局部变量。 非静态局部变量仅在函数体中可见，如果它们在堆栈上声明，会在函数退出时超出范围。 构造局部变量并通过值返回它时，编译器通常可以执行所谓的返回值优化以避免不必要的复制操作。 如果通过引用返回局部变量，则编译器会发出警告，因为调用方为使用该引用而进行的任何尝试会在局部变量已销毁之后进行。

在 C++ 中，局部变量可以声明为静态。 变量仅在函数体中可见，但是对于函数的所有实例，存在变量的单个副本。 局部静态对象将在 `atexit` 指定的终止期间销毁。 如果某个静态对象由于程序的控制流跳过了其声明而未构造，不会尝试销毁该对象。

返回类型中的类型推导 (C++14)

在 C++14 中，可以使用 **`auto`** 指示编译器从函数体推断返回类型，而不必提供结尾返回类型。 请注意， **`auto`** 始终推导为按值返回。 使用 `auto&&` 可指示编译器推导引用。

在此示例中， **`auto`** 会推导为 lhs 和 rhs 之和的非常量值副本。

C++

```
template<typename Lhs, typename Rhs>
auto Add2(const Lhs& lhs, const Rhs& rhs)
{
    return lhs + rhs; //returns a non-const object by value
}
```

请注意， **`auto`**不会保留它推导的类型的常量性。 对于返回值需要保留其自变量的常量性或引用性的转发函数，可以使用 **`decltype(auto)`** 关键字，该关键字使用 **`decltype`** 类型推断规则并保留所有类型信息。 **`decltype(auto)`**可以用作左侧的普通返回值，或结尾返回值。

下面的示例（基于来自 N3493 的代码）演示的 **`decltype(auto)`** 用于采用在模板实例化之前未知的返回类型实现函数自变量的完美转发。

C++

```
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());
}
```

从函数返回多个值

可通过多种方式从函数返回多个值：

1. 将值封装在命名类或结构对象中。 要求类或结构定义对调用方可见：

C++

```
#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. 返回 std::tuple 或 std::pair 对象：

C++

```
#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 版本 15.3 及更高版本**（在 /std:c++17 模式及更高版本中可用）： 使用结构化绑定。 结构化绑定的优点是，存储返回值的变量在声明的同时被初始化，在某些情况下，这可以显著提高效率。 在语句 `auto[x, y, z] = f();` 中，括号引入并初始化了整个函数块范围内的名称。

C++

```
#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. 除了使用返回值本身之外，还可以通过定义任意数量的参数来“返回”值以使用“按引用传递”，以便函数可以修改或初始化调用方提供的对象的值。 有关详细信息，请参阅 引用类型函数自变量。

函数指针

C++ 通过与 C 语言相同的方式支持函数指针。 但是更加类型安全的替代方法通常是使用函数对象。

建议使用`typedef`声明函数指针类型的别名（如果声明返回函数指针类型的函数）。 例如

C++

```
typedef int (*fp)(int);
fp myFunction(char* s); // function returning function pointer
```

如果不执行此操作，则函数声明的正确语法可以通过用函数名称和自变量列表替换标识符（上例中为`fp`）来从函数指针的声明符语法推导出，如下所示：

C++

```
int (*myFunction(char* s))(int);
```

前面的声明与使用前面的`typedef`的声明等效。

另请参阅

函数重载
包含变量参数列表的函数
显式默认设置的函数和已删除的函数
针对函数的依赖于自变量的名称 (Koenig) 查找
默认自变量
内联函数

Last updated on 2025/11/07

包含变量自变量列表的函数 (C++)

具有省略号（...）作为最后一个参数的函数声明采用可变数量的自变量。 C++仅为显式声明的参数提供类型检查。 当函数的参数数量和类型可能有所不同时，可以使用变量参数列表。 函数 `printf` 系列是具有变量参数列表的函数的示例。

包含变量自变量的函数

若要在声明参数后访问参数，请使用标准 include 文件 `<stdarg.h>` 中包含的宏，如本文所述。

Microsoft 专用

Microsoft C++允许将省略号指定为参数（如果省略号是最后一个参数），并且省略号位于省略号之前。 因此，声明 `int Func( int i, ... );` 是合法的，但 `int Func( int i ... );` 不是。

结束 Microsoft 专用

采用可变数量的参数的函数的声明至少需要一个占位符参数，即使它未使用。 如果未提供此占位符参数，则无法访问其余参数。

当类型的 **`char`** 参数作为变量参数传递时，它们将转换为类型 **`int`**。 同样，当类型的 **`float`** 参数作为变量参数传递时，它们将转换为类型 **`double`**。 其他类型的自变量受常见整型和浮点型提升的限制。 有关详细信息，请参阅 标准转换。

使用参数列表中的省略号 (...) 来声明需要变量列表的函数。 使用 `<stdarg.h>` include 文件中描述的类型和宏来访问通过变量列表传递的参数。 有关这些宏的详细信息，请参阅 va_arg、va_copy、va_end、va_start。

以下示例演示如何使用宏处理变量参数列表：

C++

```
// 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.
```

```c
//
// 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
```

上一个示例演示以下重要概念：

1. 在访问任何变量参数前，必须建立一个列表标记作为类型 `va_list` 的变量。 在前面的示例中，该标记称为 `vl`。
2. 使用 `va_arg` 宏访问各个参数。 必须告知 `va_arg` 宏要检索的参数的类型，以便它可以从堆栈中传输正确的字节数。 如果为 `va_arg` 指定的大小的类型与通过调用程序提供的类型不同，则结果是不可预知的。
3. 应将使用 `va_arg` 宏获取的结果显式强制转换为所需类型。
4. 必须调用 `va_end` 宏以终止变量参数处理。

Last updated on 2025/11/07

函数重载

C++ 允许在同一范围内指定多个同名函数。 这些函数称为重载函数或重载。 利用重载函数，你可以根据参数的类型和数量为函数提供不同的语义。

以采用 `print` 参数的 `std::string` 函数为例。 此函数执行的任务可能与采用 `double` 类型参数的函数大不相同。 通过重载，不必使用诸如 `print_string` 或 `print_double` 之类的名称。 在编译时，编译器会根据调用方传入的参数类型和数量选择要使用的重载。 如果你调用 `print(42.0)`，则会调用 `void print(double d)` 函数。 如果你调用 `print("hello world")`，则会调用 `void print(std::string)` 重载。

可以重载成员函数和自由函数。 下表显示了 C++ 使用函数声明的哪些部分来区分同一范围内具有相同名称的函数组。

重载注意事项

展开表

函数声明元素	是否用于重载?
函数返回类型	否
自变量的数量	是
自变量的类型	是
省略号存在或缺失	是
`typedef` 名称的使用	否
未指定的数组边界	否
`const` 或 `volatile`	是，应用于整个函数时
引用限定符（`&` 和 `&&`）	是

示例

以下示例演示了如何使用函数重载：

C++

```
// 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();
}
```

上述代码显示了文件范围内 `print` 函数的重载。

默认参数不被视为函数类型的一部分。 因此，它不用于选择重载函数。 仅在默认自变量上存在差异的两个函数被视为多个定义而不是重载函数。

不能为重载运算符提供默认参数。

自变量匹配

编译器根据当前范围内的函数声明与函数调用中提供的参数的最佳匹配，来选择要调用的重载函数。 如果找到合适的函数，则调用该函数。 此上下文中的“Suitable”具有下列含义之一：

- 找到完全匹配项。
- 已执行不重要的转换。
- 已执行整型提升。
- 已存在到所需自变量类型的标准转换。
- 已存在到所需参数类型的用户定义转换（转换运算符或构造函数）。
- 已找到省略号所表示的自变量。

编译器为每个自变量创建一组候选函数。 候选函数是这样一种函数，其中的实际自变量可以转换为形式自变量的类型。

为每个自变量生成一组“最佳匹配函数”，并且所选函数是所有集的交集。 如果交集包含多个函数，则重载是不明确的并会生成错误。 对于至少一个参数而言，最终选择的函数始终比组中的所有其他函数更匹配。 如果没有明显的优胜者，函数调用会生成编译器错误。

考虑下面的声明（针对下面的讨论中的标识，将函数标记为 `Variant 1`、`Variant 2` 和 `Variant 3`）：

```
C++
```

```
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;
```

请考虑下列语句：

C++

```
F1 = Add( F2, 23 );
```

前面的语句生成两个集：

展开表

集 1：其第一个参数的类型为 `Fraction` 的候选函数	集 2：其第二个参数可转换为类型 `int` 的候选函数
变体 1	Variant 1（可使用标准转换将 `int` 转换为 `long`）
变体 3	

集 2 中的函数具有从实参类型到形参类型的隐式转换。 其中一个函数将实参类型转换为相应形参类型的“成本”最小。

这两个集的交集为 Variant 1。 不明确的函数调用的示例为：

C++

```
F1 = Add( 3, 6 );
```

前面的函数调用生成以下集：

展开表

集 1：其第一个参数的类型为 `int` 的候选函数	集 2：其第二个参数的类型为 `int` 的候选函数
Variant 2（可使用标准转换将 `int` 转换为 `long`）	Variant 1（可使用标准转换将 `int` 转换为 `long`）

由于这两个集的交集为空，因此编译器会生成错误消息。

对于参数匹配，具有 n 个默认参数的函数被视为 n+1 个单独函数，并且每个函数均具有不同数量的参数。

省略号（`...`）用作通配符；它与任何实参匹配。 如果你未极其谨慎地设计重载函数集，它可能导致产生许多不明确的集。

> ⓘ **注意**
>
> 重载函数的多义性无法确定，直到遇到函数调用。 此时，将为函数调用中的每个自变量生成集，并且可以确定是否存在明确的重载。 这意味着，多义性可能会保留在代码中，直到它们由特定函数调用引发。

参数类型差异

重载函数区分使用不同的初始值设定项的自变量类型。 因此，对于重载而言，给定类型的自变量和对该类型的引用将视为相同。 由于它们采用相同的初始值设定项，因此它们被视为是相同的。 例如，`max( double, double )` 被视为与 `max( double &, double & )` 相同。 声明两个此类函数会导致错误。

出于同一原因，对于重载而言，由 **`const`** 或 **`volatile`** 修饰的类型的函数参数的处理方式与基类型没有什么不同。

但是，函数重载机制可以区分由 **`const`** 和 **`volatile`** 限定的引用和对基类型的引用。 它可以实现如下代码：

C++

```
// 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& ).
}
```

输出

Output

```
Over default constructor
Over&
Over default constructor
const Over&
Over default constructor
volatile Over&
```

对于重载而言，指向 `const` 和 `volatile` 对象的指针也被认为与指向基类型的指针不同。

自变量匹配和转换

当编译器尝试根据函数声明中的参数匹配实际参数时，如果未找到任何确切匹配项，它可以提供标准转换或用户定义的转换来获取正确类型。 转换的应用程序受这些规则的限制：

- 不考虑包含多个用户定义转换的转换序列。
- 不考虑可通过删除中间转换来缩短的转换序列。

最终的转换序列（如果有）称为最佳匹配序列。 可通过多种方式使用标准转换将 `int` 类型的对象转换为 `unsigned long` 类型的对象（如标准转换中所述）：

- 从 `int` 转换为 `long`，然后从 `long` 转换为 `unsigned long`。
- 从 `int` 转换为 `unsigned long`。

虽然第一个序列达到了预期的目标，但它不是最佳匹配序列，因为存在更短的序列。

下表显示了一组称为常用转换的转换。 常用转换对编译器选择哪个序列作为最佳匹配的影响有限。 表后说明了常用转换的效果。

常用转换

展开表

参数类型	转换后的类型
`type-name`	`type-name&`
`type-name&`	`type-name`
`type-name[]`	`type-name*`
`type-name(argument-list)`	`(*type-name)(argument-list)`

参数类型	转换后的类型
`type-name`	`const type-name`
`type-name`	`volatile type-name`
`type-name*`	`const type-name*`
`type-name*`	`volatile type-name*`

在其中尝试转换的序列如下：

1. 完全匹配。 用于调用函数的类型与函数原型中声明的类型之间的完全匹配始终是最佳匹配。 常用转换的序列将归类为完全匹配。 但是，不进行任何这些转换的序列被视为比进行转换的序列更佳：
    - 从指针，到指向 **`const`** 的指针（`type-name*` 到 `const type-name*`）。
    - 从指针，到指向 **`volatile`** 的指针（`type-name*` 到 `volatile type-name*`）。
    - 从引用，到对 **`const`** 的引用（`type-name&` 到 `const type-name&`）。
    - 从引用，到对 **`volatile`** 的引用（`type-name&` 到 `volatile type&`）。
2. 使用提升的匹配。 未归类为仅包含整型提升、从 **`float`** 到 **`double`** 的转换以及常用转换的完全匹配的任何序列都被归类为使用提升的匹配。 尽管比不上完全匹配，但使用提升的匹配仍优于使用标准转换的匹配。
3. 使用标准转换的匹配。 未归类为完全匹配或仅包含标准转换和常用转换的使用提升的匹配的序列将归类为使用标准转换的匹配。 在此类别中，以下规则将适用：
    - 从指向派生类的指针到指向直接或间接基类的指针的转换优于到 `void *` 或 `const void *` 的转换。
    - 从指向派生类的指针到指向基类的指针的转换会产生一个到直接基类的更好匹配。 假定类层次结构如下图所示：

显示首选转换的图形。

从 `D*` 类型到 `C*` 类型的转换优于从 `D*` 类型到 `B*` 类型的转换。 同样，从 `D*` 类型到 `B*` 类型的转换优于从 `D*` 类型到 `A*` 类型的转换。

此同一规则适用于引用转换。 从 `D&` 类型到 `C&` 类型的转换优于从 `D&` 类型到 `B&` 类型的转换等。

此同一规则适用于指向成员的指针转换。 从 `T D::*` 类型到 `T C::*` 类型的转换优于从 `T D::*` 类型到 `T B::*` 类型的转换等（其中，`T` 是该成员的类型）。

前面的规则仅沿派生的给定路径应用。 考虑下图中显示的关系图。

显示首选转换的多重继承图。

从 `C*` 类型到 `B*` 类型的转换优于从 `C*` 类型到 `A*` 类型的转换。 原因是它们位于同一个路径上，且 `B*` 更为接近。 但是，从 `C*` 类型到 `D*` 类型的转换不优于到 `A*` 类型的转换；没有首选项，因为这些转换遵循不同的路径。

1. 使用用户定义的转换的匹配。 此序列不能归类为完全匹配、使用提升的匹配或使用标准转换的匹配。 若要归类为使用用户定义转换的匹配，序列必须仅包含用户定义的转换、标准转换或常用转换。 使用用户定义转换的匹配被认为优于使用省略号（`...`）的匹配，但比不上使用标准转换的匹配。

2. 使用省略号的匹配。 与声明中的省略号匹配的任何序列将归类为使用省略号的匹配。 它被视为最弱匹配。

如果内置提升或转换不存在，则用户定义的转换将适用。 这些转换是根据要匹配的参数的类型来选择的。 考虑下列代码：

C++

```
// argument_matching1.cpp
class UDC
{
public:
   operator int()
   {
      return 0;
   }
   operator long();
};

void Print( int i )
{
```

```
}

UDC udc;

int main()
{
   Print( udc );
}
```

类 `UDC` 的可用用户定义转换来自 **`int`** 类型和 **`long`** 类型。 因此，编译器会考虑针对将匹配的对象类型的转换：`UDC`。 到 **`int`** 的转换已存在且已被选中。

在匹配参数的过程中，标准转换可应用于参数和用户定义的转换的结果。 因此，下面的代码将适用：

C++

```
void LogToFile( long l );
...
UDC udc;
LogToFile( udc );
```

在此示例中，编译器调用用户定义的转换 `operator long`，将 `udc` 转换为 **`long`** 类型。 如果未定义到 **`long`** 类型的用户定义转换，编译器会先使用用户定义的 `UDC` 转换将 **`int`** 类型转换为 `operator int` 类型。 然后应用从 **`int`** 类型到 **`long`** 类型的标准转换，以匹配声明中的参数。

如果需要任何用户定义的转换来匹配参数，则在计算最佳匹配时不会使用标准转换。 即使多个候选函数需要用户定义的转换，这些函数也被认为是相等的。 例如：

C++

```
// 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 );
}
```

`Func` 的两个版本都需要用户定义的转换，以将 `int` 类型转换为类类型参数。 可能的转换包括：

- 从 `int` 类型转换为 `UDC1` 类型（用户定义的转换）。
- 从 `int` 类型转换为 `long` 类型；然后转换为 `UDC2` 类型（两步转换）。

即使第二个转换需要标准转换和用户定义的转换，这两个转换仍被视为相等。

> **注意**
>
> 用户定义的转换被认为是构造转换或初始化转换。 编译器在确定最佳匹配时，会将这两种方法视为相等。

参数匹配和 `this` 指针

处理类成员函数的方式各不相同，具体取决于它们是否已被声明为 `static`。 `static` 函数没有提供 `this` 指针的隐式参数，因此它们被认为比常规成员函数少一个参数。 在其他方面，它们的声明相同。

不是 `static` 的成员函数要求隐含的 `this` 指针与调用该函数的对象类型相匹配。 或者，对于重载运算符，它们要求第一个参数与应用运算符的对象相匹配。 有关重载运算符的详细信息，请参阅重载运算符。

与重载函数中的其他参数不同，编译器在尝试匹配 `this` 指针参数时，不会引入临时对象，也不会尝试转换。

当 `->` 成员选择运算符用于访问类 `class_name` 的成员函数时， `this` 指针参数具有 `class_name * const` 的类型。 如果将成员声明为 `const` 或 `volatile`，则类型分别为 `const class_name * const` 和 `volatile class_name * const`。

`.` 成员选择运算符以相同的方式工作，只不过隐式 `&` (address-of) 运算符将成为对象名称的前缀。 下面的示例演示了此工作原理：

C++

```
// Expression encountered in code
obj.name
```

```
// How the compiler treats it
(&obj)->name
```

处理 `->*` 和 `.*` （指向成员的指针）运算符的左操作数的方式与处理与参数匹配相关的 `.` 和 `->` （成员选择）运算符的方式相同。

成员函数的引用限定符

引用限定符可以根据 **`this`** 指向的对象是 rvalue 还是 lvalue 来重载成员函数。 在选择不提供对数据的指针访问的情况下，使用此功能可避免不必要的复制操作。 例如，假设类 `C` 在其构造函数中初始化一些数据，并在成员函数 `get_data()` 中返回这些数据的副本。 如果 `C` 类型的对象是即将被销毁的 rvalue，编译器会选择 `get_data() &&` 重载，该重载将移动而不是复制数据。

C++

```
#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;
}
```

重载限制

多个限制管理可接受的重载函数集：

- 重载函数集内的任意两个函数必须具有不同的参数列表。
- 仅基于返回类型重载具有相同类型的参数列表的函数是错误的。

  Microsoft 专用

  可以根据返回类型重载 `operator new`，特别是根据指定的内存模型修饰符。

  结束 Microsoft 专用

- 只要一个成员函数是 `static`，而另一个不是 `static`，就无法重载。
- `typedef` 声明不定义新类型；它们引入现有类型的同义词。 它们不影响重载机制。 考虑下列代码：

  ```C++
  typedef char * PSTR;

  void Print( char *szToPrint );
  void Print( PSTR szToPrint );
  ```

  前面的两个函数具有相同的自变量列表。 `PSTR` 是 `char *` 类型的同义词。 在成员范围内，此代码生成错误。

- 枚举类型是不同的类型，并且可用于区分重载函数。
- 就区分重载函数而言，类型“array of”和“pointer to”是等效的，但仅适用于一维数组。 这些重载函数会发生冲突并生成错误消息：

  ```C++
  void Print( char *szToPrint );
  void Print( char szToPrint[] );
  ```

  对于更高维度的数组，第二个和后续维度被视为类型的一部分。 它们可用来区分重载函数：

  ```C++
  void Print( char szToPrint[] );
  void Print( char szToPrint[][7] );
  void Print( char szToPrint[][9][42] );
  ```

重载、重写和隐藏

同一范围内具有同一名称的任何两个函数声明都可以引用同一函数或两个不同的重载函数。 如果声明的自变量列表包含等效类型的自变量（如上一节所述），函数声明将引用同一函数。 否则，它们将引用使用重载选择的两个不同的函数。

需要严格遵守类范围。 在基类中声明的函数与在派生类中声明的函数不在同一范围内。 如果使用与基类中的 `virtual` 函数相同的名称声明派生类中的函数，则该派生类函数会重写基类函数。 有关详细信息，请参阅虚函数。

如果未将基类函数声明为 `virtual`，则称派生类函数隐藏它。 重写和隐藏与重载不同。

需要严格遵守块范围。 在文件范围中声明的函数与在本地声明的函数不在同一范围内。 如果在本地声明的函数与在文件范围中声明的函数具有相同名称，则在本地声明的函数将隐藏文件范围内的函数而不是导致重载。 例如：

C++

```
// 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" );
}
```

前面的代码显示函数 `func` 中的两个定义。 由于 `char *` 语句，采用 `main` 类型的参数的定义是 **`extern`** 的本地定义。 因此，采用 **`int`** 类型的参数的定义被隐藏，而对 `func` 的第一次调用出错。

对于重载的成员函数，不同版本的函数可能获得不同的访问权限。 它们仍被视为在封闭类的范围内，因此是重载函数。 请考虑下面的代码，其中的成员函数 `Deposit` 将重载；一个版本是公

共的，另一个版本是私有的。

此示例的目的是提供一个 `Account` 类，其中需要正确的密码来执行存款。 此操作通过重载来完成。

对 `Deposit` 中的 `Account::Deposit` 的调用会调用私有成员函数。 此调用是正确的，因为 `Account::Deposit` 是成员函数，可以访问类的私有成员。

C++

```
// 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;
}
```

另请参阅

函数 (C++)

Last updated on 2025/11/07

显式默认设置的函数和已删除的函数

在 C++11 中，默认函数和已删除函数使你可以显式控制是否自动生成特殊成员函数。 已删除的函数还可为你提供简单语言，以防止所有类型的函数（特殊成员函数和普通成员函数以及非成员函数）的自变量中出现有问题的类型提升，这会导致意外的函数调用。

显式默认设置的函数和已删除函数的好处

在 C++ 中，如果某个类型未声明它本身，则编译器将自动为该类型生成默认构造函数、复制构造函数、复制赋值运算符和析构函数。 这些函数称为*特殊成员函数*，它们使 C++ 中的简单用户定义类型的行为如同 C 中的结构。也就是说，无需额外的编码工作就可创建、复制和销毁它们。 C++11 会将移动语义引入语言中，并将移动构造函数和移动赋值运算符添加到编译器可自动生成的特殊成员函数的列表中。

这对于简单类型非常方便，但是复杂类型通常自己定义一个或多个特殊成员函数，这可以阻止自动生成其他特殊成员函数。 实践操作：

- 如果显式声明了任何构造函数，则不会自动生成默认构造函数。
- 如果显式声明了虚拟析构函数，则不会自动生成默认析构函数。
- 如果显式声明了移动构造函数或移动赋值运算符，则：
  - 不自动生成复制构造函数。
  - 不自动生成复制赋值运算符。
- 如果显式声明了复制构造函数、复制赋值运算符、移动构造函数、移动赋值运算符或析构函数，则：
  - 不自动生成移动构造函数。
  - 不自动生成移动赋值运算符。

> **注意**
>
> 此外，C++11 标准指定将以下附加规则：
>
> - 如果显式声明了复制构造函数或析构函数，则弃用复制赋值运算符的自动生成。
> - 如果显式声明了复制赋值运算符或析构函数，则弃用复制构造函数的自动生成。
>
> 在这两种情况下，Visual Studio 将继续隐式自动生成所需的函数且默认不发出警告。 自 Visual Studio 2022 版本 17.7 起，可以启用 C5267 来发出警告。

这些规则的结果也可能泄漏到对象层次结构中。 例如，如果基类出于任何原因无法拥有可从派生类调用的默认构造函数 - 也就是说，一个不采用任何参数的 `public` 或 `protected` 构造函数，那么从基类派生的类无法自动生成它自己的默认构造函数。

这些规则可能会使本应直接的内容、用户定义类型和常见 C++ 惯例的实现变得复杂 — 例如，通过以私有方式复制构造函数和复制赋值运算符，而不定义它们，使用户定义类型不可复制。

C++

```
struct noncopyable
{
  noncopyable() {}

private:
  noncopyable(const noncopyable&);
  noncopyable& operator=(const noncopyable&);
};
```

在 C++11 之前，此代码段是不可复制的类型的惯例形式。 但是，它具有几个问题：

- 复制构造函数必须以私有方式进行声明以隐藏它，但因为它进行了完全声明，所以会阻止自动生成默认构造函数。 如果你需要默认构造函数，则必须显式定义一个（即使它不执行任何操作）。
- 即使显式定义的默认构造函数不执行任何操作，编译器也会将它视为重要内容。 其效率低于自动生成的默认构造函数，并且会阻止 `noncopyable` 成为真正的 POD 类型。
- 尽管复制构造函数和复制赋值运算符在外部代码中是隐藏的，但成员函数和 `noncopyable` 的友元仍可以看见并调用它们。 如果它们进行了声明但是未定义，则调用它们会导致链接器错误。
- 虽然这是广为接受的惯例，但是除非你了解用于自动生成特殊成员函数的所有规则，否则意图不明确。

在 C++11 中，不可复制的习语可通过更直接的方法实现。

C++

```
struct noncopyable
{
  noncopyable() =default;
  noncopyable(const noncopyable&) =delete;
  noncopyable& operator=(const noncopyable&) =delete;
};
```

请注意如何解决与 C++11 之前的惯例有关的问题：

- 仍可通过声明复制构造函数来阻止生成默认构造函数，但可通过将其显式设置为默认值进行恢复。
- 显式设置的默认特殊成员函数仍被视为不重要的，因此性能不会下降，并且不会阻止 `noncopyable` 成为真正的 POD 类型。
- 复制构造函数和复制赋值运算符是公共的，但是已删除。 定义或调用已删除函数是编译时错误。
- 对于了解 `=default` 和 `=delete` 的人来说，意图是非常清楚的。 你不必了解用于自动生成特殊成员函数的规则。

对于创建不可移动、只能动态分配或无法动态分配的用户定义类型，存在类似惯例。 所有这些惯例都具有 C++11 之前的实现，这些实现会遭受类似问题，并且可在 C++11 中通过按照默认和已删除特殊成员函数实现它们，以类似方式进行解决。

显式默认设置的函数

可以默认设置任何特殊成员函数 — 以显式声明特殊成员函数使用默认实现、定义具有非公共访问限定符的特殊成员函数或恢复其他情况下被阻止其自动生成的特殊成员函数。

可通过如此示例所示进行声明来默认设置特殊成员函数：

C++

```
struct widget
{
  widget()=default;

  inline widget& operator=(const widget&);
};

inline widget& widget::operator=(const widget&) =default;
```

请注意，只要特殊成员函数可内联，便可以在类主体外部默认设置它。

由于普通特殊成员函数的性能优势，因此我们建议你在需要默认行为时首选自动生成的特殊成员函数而不是空函数体。 你可以通过显式默认设置特殊成员函数，或通过不声明它（也不声明其他会阻止它自动生成的特殊成员函数），来实现此目的。

已删除的函数

可以删除特殊成员函数和普通成员函数以及非成员函数，以阻止定义或调用它们。 通过删除特殊成员函数，可以更简洁地阻止编译器生成不需要的特殊成员函数。 必须在声明函数时将其删

除；不能在这之后通过声明一个函数然后不再使用的方式来将其删除。

C++

```
struct widget
{
  // deleted operator new prevents widget from being dynamically allocated.
  void* operator new(std::size_t) = delete;
};
```

删除普通成员函数或非成员函数可阻止有问题的类型提升导致调用意外函数。 这可发挥作用的原因是，已删除的函数仍参与重载决策，并提供比提升类型之后可能调用的函数更好的匹配。 函数调用将解析为更具体的但可删除的函数，并会导致编译器错误。

C++

```
// 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; }
```

请注意，在前面的示例中，使用 `call_with_true_double_only` 参数调用 **`float`** 将导致编译器错误，但使用 `call_with_true_double_only` 参数调用 **`int`** 不会导致编译器错误；在 **`int`** 的情况中，此参数将从 **`int`** 提升到 **`double`**，并成功调用函数的 **`double`** 版本，即使这可能并不是预期目的。 若要确保使用非双精度自变量对此函数进行的任何调用均会导致编译器错误，你可以声明已删除的函数的模板版本。

C++

```
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.
```

Last updated on 2025/11/07

针对函数的依赖于自变量的名称 (Koenig) 查找

编译器可以使用依赖于自变量的名称查找来查找非限定函数调用的定义。 依赖于自变量的名称查找也称为 Koenig 查找。 在命名空间、类、结构、联合或模板的层次结构中定义函数调用中每个自变量的类型。 当指定未限定的后缀函数调用时，编译器将在与每个自变量类型关联的层次结构中搜索函数定义。

示例

在此示例中，编译器注意到函数 `f()` 采用了自变量 `x`。 自变量 `x` 是命名空间 `A::X` 中定义的类型 `A`。 编译器搜索命名空间 `A` 并查找采用类型 `f()` 的参数的函数 `A::X` 的定义。

C++

```
// 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);
}
```

Last updated on 2025/11/07

默认自变量

在许多情况下，函数具有不常使用的自变量，因为使用默认值便已足够。 为了解决此问题，默认自变量工具允许为函数仅指定在给定调用中有意义的自变量。 为了阐释此概念，请考虑函数重载中所示的示例。

C++

```
// 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.
```

在许多应用程序中，可为 `prec` 提供合理的默认值，从而消除对两个函数的需求：

C++

```
// 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.
```

略微更改了 `print` 函数的实现以反映类型 **`double`** 仅存在一个此类函数这一事实：

C++

```
// 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();
}
```

若要调用新的 `print` 函数，请使用如下代码：

```C++
print( d );    // Precision of 2 supplied by default argument.
print( d, 0 ); // Override default argument to achieve other
//  results.
```

使用默认参数时，请注意以下几点：

- 默认参数仅在其中省略了尾随参数的函数调用中使用 - 它们必须是最后的参数。 因此，以下代码是非法的：

  ```C++
  int print( double dvalue = 0.0, int prec );
  ```

- 默认参数不能在以后的声明中重新定义，即使重新定义的参数与原始参数相同也是如此。 因此，以下代码将生成错误：

  ```C++
  // Prototype for print function.
  int print( double dvalue, int prec = 2 );

  ...

  // Definition for print function.
  int print( double dvalue, int prec = 2 )
  {
  ...
  }
  ```

  此代码的问题在于定义中的函数声明重新定义了 `prec` 的默认自变量。

- 以后的声明可添加额外的默认自变量。

- 可为指向函数的指针提供默认自变量。 例如：

  ```C++
  int (*pShowIntVal)( int i = 0 );
  ```

Last updated on 2025/11/07

内联函数 (C++)

关键字 `inline` 建议编译器使用函数定义中的代码替换对该函数的每次调用。

理论上，使用内联函数可以加速程序运行，因为它们消除了与函数调用关联的开销。 调用函数需要将返回地址推送到堆栈、将参数推送到堆栈、跳转到函数体，然后在函数完成时执行返回指令。 通过内联函数可以消除此过程。 相对于未内联扩展的函数，编译器还有其他机会来优化内联扩展的函数。 内联函数的一个缺点是程序的整体大小可能会增加。

内联代码替换操作由编译器自行决定。 例如，如果某个函数的地址已被占用或编译器判定函数过大，则编译器不会内联该函数。

类声明的主体中定义的函数是隐式内联函数。

示例

在以下类声明中，`Account` 构造函数是内联函数，因为它是在类声明的正文中定义的。 成员函数 `GetBalance`、`Deposit` 和 `Withdraw` 是在其定义中指定的 `inline`。 关键字 `inline` 在类声明中的函数声明中是可选的。

C++

```
// account.h
class Account
{
public:
    Account(double initial_balance)
    {
        balance = initial_balance;
    }

    double GetBalance() const;
    double Deposit(double amount);
    double Withdraw(double amount);

private:
    double balance;
};

inline double Account::GetBalance() const
{
    return balance;
}

inline double Account::Deposit(double amount)
{
    balance += amount;
    return balance;
}
```

```
inline double Account::Withdraw(double amount)
{
    balance -= amount;
    return balance;
}
```

> ⓘ **注意**
>
> 在类声明中，声明函数而无需 `inline` 关键字。 `inline` 关键字可以在类声明中指定；结果也相同。

给定的内联成员函数在每个编译单元中必须以相同的方式进行声明。 内联函数必须只有一个定义。

除非该函数的定义包含 `inline` 说明符，否则类成员函数默认为外部链接。 前面的示例显示，无需使用 `inline` 说明符显式声明这些函数。 在函数定义中使用 `inline` 建议编译器将其视为内联函数。 但是，无法在调用该函数后将函数重新声明为 `inline`。

`inline`、`__inline` 和 `__forceinline`

`inline` 和 `__inline` 说明符建议编译器将函数体的副本插入到调用函数的每个位置。

仅当编译器自己的成本收益分析显示有价值时，才会进行插入（称为*内联展开*或*内联*）。 内联展开以代码大小较大的潜在成本最大程度地减少函数调用开销。

`__forceinline` 关键字会重写成本收益分析，改为依赖于程序员的判断。 使用 `__forceinline` 时应小心谨慎。 不加选择地使用 `__forceinline` 可能会形成较大代码但是仅获得边际性能提升，或是在某些情况下，甚至会损失性能（例如，由于较大可执行文件的分页会增加）。

编译器将内联扩展选项和关键字视为建议。 不保证会对函数进行内联。 无法强制编译器对特定函数进行内联（即使使用 `__forceinline` 关键字）。 使用 `/clr` 进行编译时，如果对函数应用了安全特性，则编译器不会对函数进行内联。

为了与以前的版本兼容，除非指定了编译器选项 `_forceinline`，否则 `__inline` 和 `__forceinline` 分别是 `/Za` 和 的同义词。

`inline` 关键字告知编译器，内联展开是首选操作。 但编译器可以忽略它。 可能会出现这种行为的两种情况是：

- 递归函数。
- 在翻译单元中的其他位置通过指针引用的函数。

和编译器确定的*其他原因一样*，这些原因会干扰内联。 不要依赖于 `inline` 说明符来导致内联某个函数。

编译器可以在多个翻译单元中将头文件中定义的内联函数作为一个可调用函数来创建，而不是扩展它。 编译器为链接器标记生成的函数，防止出现单一定义规则 (ODR) 冲突。

与普通函数一样，内联函数中的自变量计算没有明确顺序。 事实上，这可能与使用普通函数调用协议传递时的自变量计算顺序不同。

/Ob 编译器优化选项来影响是否实际进行内联函数展开。
/LTCG 是否在源代码中请求跨模块内联。

示例 1

C++

```
// inline_keyword1.cpp
// compile with: /c
inline int max(int a, int b)
{
    return a < b ? b : a;
}
```

可以通过使用 `inline` 关键字或通过将函数定义置于类定义中，来内联声明类的成员函数。

示例 2

C++

```
// inline_keyword2.cpp
// compile with: /EHsc /c
#include <iostream>

class MyClass
{
public:
    void print() { std::cout << i; }   // Implicitly inline

private:
    int i;
};
```

Microsoft 专用

`__inline` 关键字等效于 `inline`。

如果出现以下情况，则即使 `__forceinline`，编译器也无法内联函数：

- 函数或其调用方是使用 `/Ob0`（调试生成的默认选项）编译的。
- 函数和调用方使用不同类型的异常处理（一个中使用 C++ 异常处理，另一个中使用结构化异常处理）。
- 函数具有变量自变量列表。
- 除非使用 `/Ox`、`/O1` 或 `/O2` 进行编译，否则函数使用内联程序集。
- 该函数是递归函数，不设置 `#pragma inline_recursion(on)`。 递归函数使用杂注内联为 16 个调用的默认深度。 若要减小内联深度，请使用 inline_depth pragma。
- 函数是虚函数，进行虚拟调用。 对虚函数的直接调用可以进行内联。
- 程序采用函数地址，调用通过指向函数的指针进行。 对采用其地址的函数的直接调用可以进行内联。
- 函数还使用 naked__declspec 修饰符进行标记。

如果编译器无法对使用 `__forceinline` 声明的函数进行内联，它会生成级别 1 警告，但以下情况除外：

- 该函数是使用 /Od 或 /Ob0 编译的。 在这些情况下不需要进行内联。
- 该函数是在外部定义的，位于包含的库或其他翻译单元中，或者是虚拟调用目标或间接调用目标。 编译器无法识别在当前翻译单元中找不到的非内联代码。

递归函数可以用内联代码替换为 inline_depth pragma 指定的深度，最多 16 个调用。 该深度之后，递归函数调用被视为对函数实例的调用。 内联启发式方法对递归函数检查到的深度不能超过 16。 inline_recursion pragma 控制当前进行展开的函数的内联展开。 若要了解相关信息，请参阅内联函数展开 (/Ob) 编译器选项。

结束 Microsoft 专用

有关使用 `inline` 说明符的详细信息，请参阅：

- 内联类成员函数
- 使用 dllexport 和 dllimport 定义内联 (C++) 函数

何时使用内联函数

内联函数最适合用于小型函数，例如提供数据成员访问权限的函数。 短函数对函数调用的开销很敏感。 较长的函数在调用和返回序列方面花费的时间可成比例地减少，而从内联的获益也会减少。

`Point` 类可以定义如下：

C++

```
// when_to_use_inline_functions.cpp
// compile with: /c
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;
}
```

假设坐标操作是此类客户端中相对常见的操作，则将两个访问器函数（前面示例中的 `x` 和 `y`）指定为 **`inline`** 通常将节省下列操作的开销：

- 函数调用（包括参数传递和在堆栈上放置对象地址）
- 保留调用者的堆栈帧
- 设置新的堆栈帧
- 返回值通信
- 还原旧堆栈帧
- 退回

内联函数与宏

宏与 `inline` 函数之间有一些共同之处。 但存在两个重要的差异。 请考虑以下示例：

C++

```
#include <iostream>

#define mult1(a, b) a * b
#define mult2(a, b) (a) * (b)
#define mult3(a, b) ((a) * (b))

inline int multiply(int a, int b)
{
    return a * b;
}
```

```
int main()
{
    std::cout << (48 / mult1(2 + 2, 3 + 3)) << std::endl; // outputs 33
    std::cout << (48 / mult2(2 + 2, 3 + 3)) << std::endl; // outputs 72
    std::cout << (48 / mult3(2 + 2, 3 + 3)) << std::endl; // outputs 2
    std::cout << (48 / multiply(2 + 2, 3 + 3)) << std::endl; // outputs 2

    std::cout << mult3(2, 2.2) << std::endl; // no warning
    std::cout << multiply(2, 2.2); // Warning C4244 'argument': conversion from
'double' to 'int', possible loss of data
}
```

Output

```
33
72
2
2
4.4
4
```

下面是宏与内联函数之间的一些差异：

- 宏始终是内联扩展的。 但内联函数仅当编译器确定内联函数是最佳操作时才会内联。
- 宏可能会导致意外行为，从而导致微小的 bug。 例如，表达式 `mult1(2 + 2, 3 + 3)` 展开为 `2 + 2 * 3 + 3`，其计算结果为 11，但预期结果为 24。 一种看似有效的修复是在函数宏的两个参数周围添加括号，得到 `#define mult2(a, b) (a) * (b)`，这将解决当前的问题，但在处理较大表达式的一部分时仍然可能导致意外的行为。 这种情况已在前面的示例中进行了演示，可以通过将宏定义为 `#define mult3(a, b) ((a) * (b))` 来解决该问题。
- 内联函数受编译器的语义处理约束，而预处理器会扩展宏，但没有任何好处。 宏不是类型安全的，而函数是。
- 计算一次作为内联函数的参数传递的表达式。 在某些情况下，作为宏的自变量传递的表达式可计算多次。 例如，请考虑如下事项：

C++

```
#include <iostream>

#define sqr(a) ((a) * (a))

int increment(int& number)
{
    return number++;
}

inline int square(int a)
{
    return a * a;
}
```

```
int main()
{
    int c = 5;
    std::cout << sqr(increment(c)) << std::endl; // outputs 30
    std::cout << c << std::endl; // outputs 7

    c = 5;
    std::cout << square(increment(c)) << std::endl; // outputs 25
    std::cout << c; // outputs 6
}
```

Output

```
30
7
25
6
```

在此示例中，在表达式 `increment` 展开为 `sqr(increment(c))` 时调用两次函数 `((increment(c)) * (increment(c)))`。 这导致第二次调用 `increment` 时返回 6，因此表达式的计算结果为 30。 任何包含副作用的表达式在宏中使用时都可能会影响结果，请检查完全展开的宏以检查该行为是否符合预期。 而当使用内联函数 `square` 时，将只调用一次 `increment` 函数，并得到正确结果 25。

另请参阅

noinline
auto_inline

Last updated on 2025/11/07

运算符重载

当应用于类的实例时， `operator` 关键字将声明一个用于指定 operator-symbol 含义的函数。 这将为运算符提供多个含义，或者将“重载”它。 编译器通过检查其操作数类型来区分运算符不同的含义。

语法

> type `operator` operator-symbol (*parameter-list*)

注解

你可以在全局或为各个类重新定义大多数内置运算符的函数。 重载运算符作为函数来实现。

重载运算符的名称是 `operator` x，其中 x 为下表中显示的运算符。 例如，若要重载加法运算符，需要定义一个名为“operator+”的函数。 同样地，若要重载加法/赋值运算符 +=，需要定义一个名为“operator+=”的函数。

可重定义的运算符

展开表

运算符	名称	类型
"	Comma	二元
"	逻辑非	一元
%>	不相等	二元
%	取模	二元
%=	取模赋值	二元
"	位与	二元
"	Address-of	一元
%>	逻辑与	二元
%>	按位“与”赋值	二元
[.]	函数调用	"
[.]	转换运算符	一元

运算符	名称	类型
`*`	乘法	二元
`*`	指针取消引用	一元
`*=`	乘法赋值	二元
+	附加内容	二元
+	一元加	一元
++	递增 [1]	一元
+=	加法赋值	二元
-	减	二元
-	一元求反	一元
--	递减 [1]	一元
-=	减法赋值	二元
">	成员选择	二元
`->*`	指向成员的指针选定内容	二元
/	分部	二元
/=	除法赋值	二元
<	小于	二元
<<	左移	二元
<<"	左移赋值	二元
<"	小于或等于	二元
=	赋值	二元
==	等式	二元
>	大于	二元
>"	大于或等于	二元
>>	右移	二元
>>"	右移赋值	二元
[.]	数组下标	"

运算符	名称	类型
^	异或	二元
^=	异或赋值	二元
|	位或	二元
|=	按位“与或”赋值	二元
||	逻辑或	二元
~	二元反码	一元
`delete`	删除	"
`new`	新建	"
转换运算符	转换运算符	一元

[1] 存在两个一元递增和递减运算符版本：前置递增和后置递增。

有关详细信息，请参阅运算符重载的一般规则。 以下主题对各种类别的重载运算符的约束进行了介绍：

- 一元运算符
- 二元运算符
- 转让
- 函数调用
- 下标
- 类成员访问
- 递增和递减。
- 用户定义的类型转换

无法重载下表中显示的运算符。 该表包括预处理器符号 # 和 ##。

不可重定义的运算符

展开表

运算符	名称
"	成员选择
`.*`	指向成员的指针选定内容
:"	范围解析
? "	条件逻辑
#	预处理器转换为字符串
##	预处理器串联

尽管通常是在代码中遇到重载运算符时由编译器对其进行隐式调用，但也可以按照与调用任何成员或非成员函数相同的方式来显式调用重载运算符：

C++

```
Point pt;
pt.operator+( 3 );  // Call addition operator to add 3 to pt.
```

示例

以下示例重载 + 运算符，将两个复数相加并返回结果。

C++

```
// 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();
}
```

Output

```
6.8, 11.2
```

本节内容

- 运算符重载的一般规则
- 重载一元运算符
- 二元运算符
- 转让
- 函数调用
- 下标
- 成员访问

另请参阅

C++ 内置运算符、优先级和关联性
关键字

Last updated on 2025/11/07

运算符重载的一般规则

以下规则约束如何实现重载运算符。 但是，这些规则不适用于 new 和 delete 运算符，这两个运算符将单独介绍。

- 不能定义新运算符，如“.”。
- 将运算符应用于内置数据类型时，不能重新定义其含义。
- 重载运算符必须是非静态类成员函数或全局函数。 需要访问私有或受保护的类成员的全局函数必须声明为该类的友元。 全局函数必须至少采用一个类类型或枚举类型的自变量，或者作为对类类型或枚举类型的引用的自变量。 例如：

  C++

  ```
  // 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()
  {
  }
  ```

  前面的代码示例将小于运算符声明为成员函数；但是，加法运算符被声明为具有友元访问的全局函数。 请注意，可以为给定运算符提供多个实现。 对于前面的加法运算符，将提供两个实现以便于交换。 同样可能会实现用于将 `Point` 与 `Point` 相加、将 `int` 与 `Point` 相加等此类运算符。

- 运算符遵循它们通常用于内置类型时指定的操作数的优先级、分组和数量。 因此，没有方法来表达概念“将 2 和 3 与 `Point` 类型的对象相加”（应该让 2 与 x 坐标相加，3 与 y 坐标相加）。
- 声明为成员函数的一元运算符将不采用自变量；如果声明为全局函数，它们将采用一个自变量。
- 声明为成员函数的二元运算符将采用一个自变量；如果声明为全局函数，它们将采用两个自变量。
- 如果运算符可用作一元或二元运算符（、、*、+ 和 -），则可以单独重载每种用法。

- 重载运算符不能具有默认自变量。
- 除赋值 (operator=) 之外的所有重载运算符均由派生类继承。
- 成员函数重载运算符的第一个自变量始终属于针对其调用运算符的对象的类类型（从中声明运算符的类或派生自该类的类）。 没有为第一个参数提供转换。

请注意，任何运算符的含义都可以完全更改。 这包括 address-of (&)、赋值 (=) 和函数调用运算符的含义。 此外，还可以使用运算符重载来更改可用来确定内置类型的标识。 例如，以下四个语句在完全计算时通常是等效的：

C++

```
var = var + 1;
var += 1;
var++;
++var;
```

无法依靠此标识来确定重载运算符的类类型。 此外，在使用这些运算符时针对基本类型的某些隐含要求对重载运算符放宽了。 例如，相加/赋值运算符 += 要求在应用于基本类型时左操作数是 lvalue；在重载该运算符时，不存在这样的要求。

> **注意**
>
> 为保持一致性，定义重载运算符时通常最好遵循内置类型的模型。 如果某个重载运算符的语义与它在其他上下文中的含义差别很大，则它造成的混淆会盖过它的用处。

另请参阅

运算符重载

Last updated on 2025/11/07

重载一元运算符

一元运算符从单个作数生成结果。 可以定义标准一元运算符集的重载，以处理用户定义的类型。

可重载的一元运算符

可以在用户定义的类型上重载以下一元运算符：

- `!` （逻辑 NOT）
- `&` （地址）
- `~` （补充）
- `*` （指针取消引用）
- `+` （一元加）
- `-` （一元否定）
- `++` （前缀增量）或（后缀增量）
- `--` （前缀递减）或（后缀递减）
- 转换运算符

一元运算符重载声明

可以将重载的一元运算符声明为非静态成员函数或非成员函数。 重载的一元成员函数不采用任何参数，因为它们隐式作 `this`。 使用一个参数声明非member 函数。 声明这两个窗体时，编译器遵循重载解析规则来确定要使用的函数（如果有）。

以下规则适用于所有前缀一元运算符。 若要将一元运算符函数声明为非静态成员函数，请使用以下声明形式：

> `return-type` `operator` `op` `();`

在此窗体中， `return-type` 返回类型是返回类型， `op` 是上表中所列的运算符之一。

若要将一元运算符函数声明为非成员函数，请使用以下声明形式：

> `return-type` `operator` `op` `(` `class-type` `);`

在此窗体中， `return-type` 返回类型 `op` 是上表中所列的运算符之一，也是 `class-type` 要对其作的参数的类类型。

后缀形式 `++` ，并 `--` 采用额外的 `int` 参数来区分它们与前缀形式。 有关前缀和后缀形式`++` `--` 的详细信息，请参阅递增和递减运算符重载。

> ⓘ **注意**
>
> 对一元运算符的返回类型没有限制。 例如，逻辑 NOT （`!`） 返回 `bool` 值是有意义的，但不会强制实施此行为。

另请参阅

运算符重载

Last updated on 2025/11/07

递增和递减运算符重载 (C++)

由于递增和递减运算符各有两个变量，因此它们属于一个特殊类别：

- 前置递增和后置递增
- 前置递减和后置递减

编写重载的运算符函数时，为这些运算符的前缀和后缀版本实现单独的版本很有用。 若要区分这两者，请遵循以下规则：运算符的前缀形式与声明任何其他一元运算符的方式完全相同；后缀形式接受 `int` 类型的额外参数。

> **注意**
>
> 当为递增或递减运算符的后缀形式指定重载运算符时，其他参数的类型必须是 `int`；指定任何其他类型都将产生错误。

以下示例显示如何为 `Point` 类定义前缀和后缀递增和递减运算符：

C++

```
// 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()
{
}
```

可使用以下函数原型在文件范围中（全局）定义同一运算符：

C++

```
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
```

表示递增或递减运算符的后缀形式的 `int` 类型的参数不常用于传递参数。 它通常包含值 0。 但是，可按以下方式使用它：

C++

```
// increment_and_decrement2.cpp
class Int
{
public:
    Int operator++( int n ); // Postfix increment operator
private:
    int _i;
};
```

```
Int Int::operator++( int n )
{
    Int result = *this;
    if( n != 0 )    // Handle case where an argument is passed.
        _i += n;
    else
        _i++;       // Handle case where no argument is passed.
    return result;
}

int main()
{
   Int i;
   i.operator++( 25 ); // Increment by 25.
}
```

除显式调用之外，没有针对使用递增或递减运算符来传递这些值的语法，如前面的代码所示。实现此功能的更直接的方法是重载加法/赋值运算符 (+=)。

另请参阅

运算符重载

Last updated on 2025/11/07

二元运算符

下表显示可重载的运算符的列表。

可重新定义的二进制运算符

展开表

运算符	名称
"	Comma
%>	不相等
%	取模
%=	取模/赋值
"	位与
%>	逻辑与
%>	按位“与”/赋值
`*`	乘法
`*=`	乘法/赋值
+	附加内容
+=	加法/赋值
-	减
-=	减法/赋值
">	成员选择
`->*`	指向成员的指针选定内容
/	分部
/=	除法/赋值
<	小于
<<	左移
<<"	左移/赋值
<"	小于或等于

运算符	名称
=	分配
==	等式
>	大于
>"	大于或等于
>>	右移
>>"	右移/赋值
^	异或
^=	异或/赋值
|	位或
|=	按位“与或”/赋值
||	逻辑或

若要将二元运算符函数声明为非静态成员，您必须用以下形式声明它：

> *ret-type* `operator` *op* （*arg*）

其中，ret-type 是返回类型，op 是上表中列出的运算符之一，而 arg 是任何类型的自变量。

若要将二元运算符函数声明为全局函数，您必须用以下形式声明它：

> *ret-type* `operator` *op* （*arg1*，*arg2*）

其中，ret-type 和 op 是成员运算符函数，而 arg1 和 arg2 是自变量。 至少有一个参数必须是类类型。

> ⓘ **注意**
>
> 对二元运算符的返回类型没有限制；但是，大多数用户定义的二元运算符将返回类类型或对类类型的引用。

另请参阅

运算符重载

Last updated on 2025/11/07

任务

严格地说，赋值运算符 (=) 是二元运算符。 其声明与任何其他二元运算符的相同，但有以下例外：

- 它必须是非静态成员函数。 没有“operator=”可声明为非成员函数。
- 它不由派生类继承。
- 默认“operator=”函数可由类类型的编译器生成（如果该函数不存在）。

以下示例阐释如何声明赋值运算符：

C++

```
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;
}
```

所提供的自变量是表达式的右侧。 此运算符返回对象以保留赋值运算符的行为，赋值运算符在赋值完成后返回左侧的值。 这样能形成赋值链，例如：

C++

```
pt1 = pt2 = pt3;
```

复制赋值运算符不会与复制构造函数混淆。 后者是在从现有对象构造新对象的过程中调用的：

C++

```
// 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.
```

> **注意**
>
> 建议遵循**三法则 (Rule of Three)**，即定义复制赋值运算符的类还应显式定义复制构造函数和析构函数，以及移动构造函数和移动赋值运算符（从 C++11 开始）。

另请参阅

- 运算符重载
- 复制构造函数和复制赋值运算符 (C++)

Last updated on 2025/11/07

函数调用 (C++)

使用括号调用的函数调用运算符是二元运算符。

语法

```
primary-expression ( expression-list )
```

备注

在此上下文中，`primary-expression` 为第一个操作数，并且 `expression-list`（可能为自变量的空列表）为第二个操作数。 函数调用运算符用于需要大量参数的操作。 这之所以有效，是因为 `expression-list` 是列表而非单一操作数。 函数调用运算符必须是非静态成员函数。

函数调用运算符在重载时不会修改函数的调用方式；相反，它会在运算符应用于给定类的类型的对象时修改解释该运算符的方式。 例如，以下代码通常没有意义：

C++

```
Point pt;
pt( 3, 2 );
```

但是，如果存在一个适当的重载函数调用运算符，则此语法可用于将 `x` 坐标偏移 3 个单位并将 `y` 坐标偏移 2 个单位。 下面的代码显示了这样的定义：

C++

```
// 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 );
}
```

请注意，函数调用运算符适用于对象的名称，而不是函数的名称。

也可以使用指向函数的指针（而非该函数本身）重载函数调用运算符。

C++

```
typedef void(*ptf)();
void func()
{
}
struct S
{
   operator ptf()
   {
      return func;
   }
};

int main()
{
   S s;
   s();//operates as s.operator ptf()()
}
```

另请参阅

运算符重载

Last updated on 2025/11/07

下标

下标运算符 ([ ]) （如函数调用运算符）被视为二元运算符。 下标运算符必须是采用单个参数的非静态成员函数。 此自变量可以是任何类型，并指定所需的数组下标。

示例

以下示例演示如何创建用于实现边界检查的 `int` 类型的矢量：

C++

```
// 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;
}
```

Output

```
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
```

注释

当 `i` 在前一个程序中达到 10 时，operator[] 将检测是否在使用超出边界的下标并发出错误消息。

请注意，函数 operator[] 将返回引用类型。 这会使它成为左值，从而使您可以在赋值运算符的任何一侧使用下标表达式。

另请参阅

运算符重载

Last updated on 2025/11/07

成员访问

类成员访问可通过重载成员访问运算符 (>-) 来控制。 此运算符被视为此用法中的一元运算符，而重载运算符函数必须是类成员函数。 因此，此类函数的声明是：

语法

```
class-type *operator->()
```

备注

其中，class-type 是此运算符所属的类的名称。 成员访问运算符函数必须是非静态成员函数。

此运算符（通常与指针取消引用运算符一起使用）用于实现在取消引用用法或对用法计数前验证指针的“智能指针”。

无法重载 . 成员访问运算符。

另请参阅

运算符重载

Last updated on 2025/11/07

类和结构 (C++)

此部分介绍 C++ 类和结构。 这两个构造在 C++ 中是相同的，只不过在结构中，默认可访问性是公共的，而在类中，默认值是私有的。

类和结构是用于定义你自己的类型的构造。 类和结构都可以包含数据成员和成员函数，使你可以描述类型的状态和行为。

本文包含以下主题：

- class
- struct
- 类成员概述
- 成员访问控制
- 继承
- 静态成员
- 用户定义的类型转换
- 可变数据成员（可变说明符）
- 嵌套类声明
- 匿名类类型
- 指向成员的指针
- this 指针
- C++ 位域

三种类类型是结构、类和联合。 它们使用 struct、class 和 union 关键字进行声明。 下表显示三种类类型之间的差异。

有关联合的详细信息，请参阅联合。 有关 C++/CLI 和 C++/CX 中类和结构的信息，请参阅类和结构。

结构、类和联合的访问控制和约束

展开表

结构	类	Unions
类键为 `struct`	类键为 `class`	类键为 `union`
默认访问是公共的	默认访问是私有的	默认访问是公共的
没有使用约束	没有使用约束	一次只使用一个成员

另请参阅

C++ 语言参考

Last updated on 2025/11/07

class （C++）

关键字 `class` 声明类类型或定义类类型的对象。

语法

```
[template-spec]
class [ms-decl-spec] [tag [: base-list ]]
{
   member-list
} [declarators];
[ class ] tag declarators;
```

参数

template-spec
可选模板规范。 有关详细信息，请参阅模板。

class
`class` 关键字。

ms-decl-spec
可选存储类规范。 有关详细信息，请参阅 __declspec 关键字。

tag
为类提供的类型名称。 标记将变成类范围内的保留字。 标记是可选项。 如果省略，则定义匿名类。 有关详细信息，请参阅匿名类类型。

base-list
此类将从中派生其成员的类或结构的可选列表。 有关详细信息，请参阅基类。 每个基类或结构名称的前面可具有访问说明符（public、private、protected）和 virtual 关键字。 有关详细信息，请参阅控制对类成员的访问中的成员访问表。

member-list
类成员列表。 有关详细信息，请参阅类成员概述。

declarators
声明符列表，指定类类型的一个或多个实例的名称。 如果类的所有数据成员是 `public`，则声明符可以包含初始值设定项列表。 与类相比，这在结构（其数据成员默认为 `public`）中更为常见。 有关详细信息，请参阅声明符概述。

备注

有关一般类的详细信息，请参阅以下主题之一：

- struct
- union
- __multiple_inheritance
- __single_inheritance
- __virtual_inheritance

有关 C++/CLI 和 C++/CX 中托管类和结构的信息，请参阅类和结构

示例

C++

```
// class.cpp
// compile with: /EHsc
// Example of the class keyword
// Exhibits polymorphism/virtual functions.

#include <iostream>
#include <string>
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;
}
```

另请参阅

关键字
类和结构

Last updated on 2025/11/07

结构（C++）

`struct` 关键字定义结构类型和/或结构类型的变量。

语法

```
[template-spec] struct [ms-decl-spec] [tag [: base-list ]]
{
   member-list
} [declarators];
[struct] tag declarators;
```

参数

template-spec
可选模板规范。 有关详细信息，请参阅模板规范。

struct
`struct` 关键字。

ms-decl-spec
可选存储类规范。 有关详细信息，请参阅 __declspec 关键字。

标签
为结构提供的类型名称。 标记将变成结构范围内的保留字。 标记是可选项。 如果省略，则定义匿名结构。 有关详细信息，请参阅匿名类类型。

base-list
此结构将从中派生其成员的类或结构的可选列表。 有关详细信息，请参阅基类。 每个基类或结构名称的前面可具有访问说明符（public、private、protected）和 virtual 关键字。 有关详细信息，请参阅控制对类成员的访问中的成员访问表。

member-list
结构成员列表。 有关详细信息，请参阅类成员概述。 这里的唯一区别是，使用 `struct` 代替了 `class`。

declarators
指定结构名称的声明符列表。 声明符列表声明了一个或多个结构类型实例。 如果结构的所有数据成员是 `public`，则声明符可包含初始值设定项列表。 初始值设定项列表在结构中很常见，因为数据成员默认为 `public`。 有关详细信息，请参阅声明符概述。

注解

结构类型是用户定义的复合类型。 它由可具有不同类型的字段或成员构成。

在 C++ 中，结构与类相同，只不过其成员默认为 `public`。

有关 C++/CLI 中托管类和结构的信息，请参阅类和结构。

使用结构

在 C 中，必须显式使用 `struct` 关键字来声明结构。 在 C++ 中，不需要在定义该类型之后使用 `struct` 关键字。

可以选择在定义结构类型时，通过在右大括号和分号之间放置一个或多个逗号分隔的变量名称来声明变量。

可以初始化结构变量。 每个变量的初始化必须括在大括号中。

相关信息，请参阅 class、union 和 enum。

示例

C++

```
#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
```

Last updated on 2025/11/07

类成员概述

`class` 或 `struct` 由其成员组成。 类的工作由其成员函数执行。 它所维持的状态存储在其数据成员中。 成员的初始化由构造函数完成，释放内存和释放资源等清理工作由析构函数完成。 在 C++ 11 和更高版本中，数据成员可以（并且通常应该）在声明时初始化。

类成员的种类

成员类别的完整列表如下：

- 特殊成员函数。
- 成员函数概述。
- 可变和静态数据成员包括内置类型和其他用户定义的类型。
- 运算符
- 嵌套类声明和。）
- Unions
- 枚举。
- 位域。
- 有元。
- 别名和 typedef。

> ⓘ **注意**
>
> 前面的列表中包括友元，因为它们包含在类声明中。 但是，它们不是真正的类成员，因为它们不在类范围内。

类声明示例

下面的示例显示了一个简单的类声明：

C++

```
// 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 };
```

成员可访问性

在成员列表中声明类的成员。 可使用称为访问说明符的关键字将类的成员列表分为任意数量的 `private`、`protected` 和 `public` 部分。 冒号 `:` 必须跟在访问说明符后面。 这些部分不需要是连续的，也就是说，这些关键字中的任何一个都可能在成员列表中多次出现。 关键字指定所有成员直到下一个访问说明符或右大括号的访问。 有关详细信息，请参阅成员访问控制 (C++)。

静态成员

可将数据成员声明为静态，这表示类的所有对象都有权访问它的同一副本。 可将成员函数声明为静态，在这种情况下它只能访问类的静态数据成员（且不具有 `this` 指针）。 有关详细信息，请参阅静态数据成员。

特殊成员函数

如果未在源代码中指定特殊成员函数，那么编译器自动提供的函数则为特殊成员函数。

- 默认构造函数
- 复制构造函数
- (C++11) 移动构造函数
- 复制赋值运算符
- (C++11) 移动赋值运算符
- 析构函数

有关详细信息，请参阅特殊成员函数。

按成员初始化

在 C++ 11 和更高版本中，非静态成员声明符可以包含初始值设定项。

C++

```
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()
{
}
```

如果在构造函数中对成员赋值，则该值将覆盖声明时指定的值。

对于给定类类型的所有对象，只有一个静态数据成员的共享副本。 必须在文件范围内定义静态数据成员并可在此范围内将其初始化。 有关静态数据成员的详细信息，请参阅静态数据成员。 以下示例演示如何使用静态数据成员：

C++

```cpp
// 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;
```

注意

类名 `CanInit2` 的前面必须有 `i` 以指定所定义的 `i` 是类 `CanInit2` 的成员。

另请参阅

类和结构

Last updated on 2025/11/07

成员访问控制 (C++)

通过访问控制，可以将类的 public 接口与 private 实现详细信息和仅供派生类使用的 protected 成员分离开来。 访问说明符应用于在它之后声明的所有成员，直到遇到下一个访问说明符。

C++

```
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();
};
```

默认访问是类中的 `private`，以及结构或联合中的 `public`。 类中的访问说明符可按任何顺序使用任意次数。 类类型的对象的存储分配是依赖于实现的。 但是，编译器必须保证在访问说明符之间将成员分配到更高的连续内存地址。

成员访问控制

展开表

访问类型	含义
private	声明为 `private` 的类成员只能由类的成员函数和友元（类或函数）使用。
protected	声明为 `protected` 的类成员可由类的成员函数和友元（类或函数）使用。 此外，它们还可由派生自该类的类使用。
public	声明为 `public` 的类成员可由任意函数使用。

访问控制有助于阻止通过不适当的方式使用对象。 在执行显式类型转换（强制转换）时，此保护将丢失。

注意

访问控制同样适用于所有名称：成员函数、成员数据、嵌套类和枚举数。

派生类中的访问控制

两个因素控制基类的哪些成员可在派生类中访问；这些相同的因素控制对派生类中的继承成员的访问：

- 派生类是否使用 `public` 访问说明符声明基类。
- 基类中对成员的访问权限如何。

下表显示了这些因素之间的交互以及如何确定基类成员访问。

基类中的成员访问

展开表

`private`	`protected`	`public`
始终无法通过任何派生访问进行访问	如果使用 `private` 派生，则在派生类中为 `private`	如果使用 `private` 派生，则在派生类中为 `private`
	如果使用 `protected` 派生，则在派生类中为 `protected`	如果使用 `protected` 派生，则在派生类中为 `protected`
	如果使用 `protected` 派生，则在派生类中为 `public`	如果使用 `public` 派生，则在派生类中为 `public`

以下示例演示了访问派生：

C++

```
// 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
}
```

在 `DerivedClass1` 中，成员函数 `PublicFunc` 是 **`public`** 成员， `ProtectedFunc` 是 **`protected`** 成员，因为 `BaseClass` 是 **`public`** 基类。 `PrivateFunc` 对于 **`private`** 是 `BaseClass` 的，因此任何派生类都无法访问它。

在 `DerivedClass2` 中，函数 `PublicFunc` 和 `ProtectedFunc` 被视为 **`private`** 成员，因为 `BaseClass` 是 **`private`** 基类。 同样， `PrivateFunc` 对于 **`private`** 是 `BaseClass` 的，因此任何派生类都无法访问它。

您可以声明派生类而不使用基类访问说明符。 在这种情况下，如果派生类声明使用 **`private`** 关键字，则将派生视为 **`class`**。 如果派生类声明使用 **`public`** 关键字，则将派生视为 **`struct`**。 例如，以下代码：

C++

```
class Derived : Base
...
```

等效于：

C++

```
class Derived : private Base
...
```

同样，以下代码：

```C++
struct Derived : Base
...
```

等效于：

```C++
struct Derived : public Base
...
```

声明为拥有 `private` 访问权限的成员对函数或派生类是不可访问的，除非使用基类中的 `friend` 声明来声明这些函数或类。

`union` 类型不能有基类。

> ⓘ **注意**
>
> 当指定 private 基类时，建议显式使用 `private` 关键字，以便让派生类的用户了解成员访问。

访问控制和静态成员

在将基类指定为 `private` 时，它只影响非静态成员。 在派生类中仍可访问公共静态成员。 但是，使用指针、引用或对象访问基类的成员需要转换，此时将再次应用访问控制。 请考虑以下示例：

```C++
// 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: 'Base::CountOf'
                               // not accessible because
                               // 'Derived1' uses 'private'
                               // to inherit from 'Base'
    return cCount;
}
```

在前面的代码中，访问控制禁止从指向 `Derived2` 的指针转换为指向 `Base` 的指针。 **`this`** 指针是 `Derived2 *` 类型的隐式表示形式。 若要选择 `CountOf` 函数，必须将 **`this`** 转换为 `Base *` 类型。 不允许执行此类转换，因为 `Base` 是 private 的 `Derived2` 间接基类。 到 private 基类类型的转换只适用于指向立即派生类的指针。 这就是为什么可以将 `Derived1 *` 类型的指针转换为 `Base *` 类型。

显式调用 `CountOf` 函数，而不使用指针、引用或对象来选择它，意味着没有转换。 这就是允许调用的原因。

派生类 `T` 的成员和友元可以将指向 `T` 的指针转换为指向 private 的 `T` 直接基类的指针。

对虚函数的访问

适用于 virtual 函数的访问控制是由用于进行函数调用的类型决定的。 重写函数的声明不会影响给定类型的访问控制。 例如：

C++

```
// 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;
}
```

在前面的示例中，使用指向 `GetState` 类型的指针调用虚函数 `VFuncBase` 将调用 `VFuncDerived::GetState`，并且会将 `GetState` 视为 **`public`**。 但是，使用指向 `GetState` 类型的指针调用 `VFuncDerived` 是一种访问控制冲突，因为 `GetState` 在 **`private`** 类中被声明为 `VFuncDerived`。

> **⊗ 注意**
>
> 可以使用指向基类 `GetState` 的指针调用虚函数 `VFuncBase`。 这并不意味着调用的函数是该函数的基类版本。

具有多重继承的访问控制

在涉及虚拟基类的多重继承方格中，可通过多个路径到达给定的名称。 由于可沿着这些不同的路径应用不同的访问控制，因此该编译器选择允许大多数访问的路径。 请参见下图：

沿继承图的路径访问

在该图中，通过类 `VBase` 始终到达类 `RightPath` 中声明的名称。 右路径更易于访问，因为 `RightPath` 将 `VBase` 声明为 **`public`** 基类，而 `LeftPath` 将 `VBase` 声明为 **`private`**。

另请参阅

C++ 语言参考

Last updated on 2025/11/07

`friend` (C++)

在某些情况下，类授予成员级对不是类成员的函数或单独类中的所有成员的访问权限非常有用。 这些免费函数和类称为 *好友*，由 `friend` 关键字标记。 只有类实现者可以声明其好友是谁。 函数或类不能将自身声明为任何类的友元。 在类定义中 `friend` ，使用非member 函数或其他类的关键字和名称授予它对类的私有和受保护成员的访问权限。 在模板定义中，类型参数可以声明为 .`friend`

语法

`friend-declaration`:

- `friend` `function-declaration`
- `friend` `function-definition`
- `friend` `elaborated-type-specifier` `;`;
- `friend` `simple-type-specifier` `;`
- `friend` `typename-specifier` `;`

`friend` 声明

如果声明 `friend` 了以前未声明的函数，该函数将导出到封闭的非类范围。

在 `friend` 声明中声明的函数被视为使用关键字声明 `extern` 的函数。 有关详细信息，请参阅 extern。

尽管具有全局范围的函数可以在原型之前声明为 `friend` 函数，但在其完整类声明的外观之前，成员函数不能声明为 `friend` 函数。 以下代码演示了此类声明如何失败：

C++

```
class ForwardDeclared;   // Class name is known.
class HasFriends
{
    friend int ForwardDeclared::IsAFriend();   // C2039 error expected
};
```

前面的示例将类名 `ForwardDeclared` 输入到作用域中，但完整的声明（具体而言，声明函数 `IsAFriend`的部分）未知。 `friend`类`HasFriends`中的声明生成错误。

在 C++11 中，类有两种形式的友元声明：

C++

```
friend class F;
friend F;
```

第一个窗体在最内部的命名空间中未找到该名称的现有类，则引入新的类 F。 C++11：第二种形式不引入新类;当类已声明时，可以使用它，并且必须在声明模板类型参数或 `typedef` 作为参数 `friend` 时使用它。

在尚未声明引用的类型时使用 `friend class F` ：

C++

```
namespace NS
{
    class M
    {
        friend class F;  // Introduces F but doesn't define it
    };
}
```

如果 `friend` 与尚未声明的类类型一起使用，则会发生错误：

C++

```
namespace NS
{
    class M
    {
        friend F; // error C2433: 'NS::F': 'friend' not permitted on data
declarations
    };
}
```

在以下示例中， `friend F` 指 `F` 在 NS 范围之外声明的类。

C++

```
class F {};
namespace NS
{
    class M
    {
        friend F;  // OK
    };
}
```

用于 `friend F` 将模板参数声明为友元：

C++

```
template <typename T>
class my_class
{
    friend T;
    //...
};
```

用于 `friend F` 将 typedef 声明为友元：

C++

```
class Foo {};
typedef Foo F;

class G
{
    friend F; // OK
    friend class F // Error C2371 -- redefinition
};
```

若要声明两个类是彼此的朋友，必须将整个第二个类指定为第一个类的友元。 此限制的原因是编译器有足够的信息来仅在声明第二个类时声明单个友元函数。

> ⓘ **注意**
>
> 虽然整个第二类必须是第一类的朋友，但你可以选择第一类中的哪些函数将是第二类的朋友。

friend 函数

`friend`函数是一个函数，该函数不是类的成员，但有权访问该类的私有成员和受保护的成员。 友元函数不被视为类成员;它们是提供特殊访问权限的正常外部函数。 好友不在类范围内，除非它们是另一个类的成员，否则不会使用成员选择运算符 （**和** `->`） 调用它们。 函数 `friend` 由授予访问权限的类声明。 可以在 `friend` 类声明中的任意位置放置声明。 它不受访问控制关键字的影响。

下面的示例显示了一个 `Point` 类和一个友元函数 `ChangePrivate`。 该 `friend` 函数有权访问作为参数接收的对象 `Point` 的私有数据成员。

C++

```
// 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
}
```

将成员类成员作为好友

类成员函数可以声明为其他类中的好友。 请看下面的示例：

C++

```
// 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
```

在前面的示例中，仅向函数 `A::Func1( B& )` 授予 `friend` 对类 `B` 的访问权限。 因此，对私有成员 `_b` 的访问在类 `A` 中 `Func1` 是正确的，但不在类中 `Func2`。

类 **`friend`** 是一个类，其成员函数都是 `friend` 类的函数，即其成员函数有权访问其他类的私有成员和受保护的成员。 **`friend`** 假设类 `B` 中的声明是：

C++

```
friend class A;
```

在这种情况下，类 `A` 中的所有成员函数都将被授予 `friend` 对类 `B` 的访问权限。 以下代码是 类的示例 `friend` ：

C++

```
// 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();
}
```

除非明确指定为友谊，否则友谊不是相互的。 在上面的示例中，无法访问其私有成员的成员 `YourOtherClass` 函数 `YourClass`。

托管类型（在 C++/CLI 中）不能有任何 `friend` 函数、 `friend` 类或 `friend` 接口。

友谊不是继承的，这意味着派生自 `YourOtherClass` 的类无法访问 `YourClass`'s private members。 友谊不是可传递的 `YourOtherClass` ，所以那些是不能访问 `YourClass` 私人成员的朋友的班级。

下图显示了四个类声明： `Base`、 `Derived`、 `aFriend` 和 `anotherFriend`。 只有类 `aFriend` 可以直接访问 （以及可能继承的任何成员） 的 `Base` 私有成员 `Base` 。

内联 `friend` 定义

可以在类声明内定义（给定函数体）的友元函数。 这些函数是内联函数。 与成员内联函数一样，它们的行为就像在看到所有类成员之后立即定义，但在类范围关闭之前（在类声明的末尾）。 类声明内定义的友元函数位于封闭类的范围内。

另请参阅

关键字

Last updated on 2025/11/07

private （C++）

语法

```
private:
   [member-list]
private base-class
```

备注

当位于类成员列表之前时，`private` 关键字指定这些成员仅可从成员函数和该类的友元中进行访问。 这适用于声明到下一个访问指示符或类的末尾的所有成员。

当位于基类的名称之前时，`private` 关键字指定基类的公共成员和受保护成员为派生类的私有成员。

类中成员的默认访问是私有的。 结构或联合中成员的默认访问是公共的。

基类的默认访问对于类是私有的，而对于结构是公共的。 联合不能具有基类。

有关相关信息，请参阅控制对类成员的访问中的友元、公共、受保护的和成员访问表。

/clr 专用

在 CLR 类型中，C++ 访问说明符关键字（`public`、`private` 和 `protected`）可能影响与程序集相关的类型和方法的可见性。 有关详细信息，请参阅成员访问控制。

> ⓘ **注意**
>
> 使用 /LN 编译的文件不受此行为的影响。 在这种情况下，所有托管类（公共或私有）都将可见。

END /clr 专用

示例

C++

```cpp
// 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
}
```

另请参阅

控制对类成员的访问
关键字

Last updated on 2025/11/07

protected （C++）

语法

```
protected:
   [member-list]
protected base-class
```

备注

`protected` 关键字指定对 member-list 中的类成员直到下一个访问说明符（`public` 或 `private`）或类定义的末尾的访问。 只能通过以下项使用声明为 `protected` 的类成员：

- 最初声明这些成员的类的成员函数。
- 最初声明这些成员的类的友元。
- 使用公共或受保护访问（派生自最初声明这些成员的类）派生的类。
- 也对受保护成员具有专用访问权限的以私有方式派生的直接类。

当位于基类的名称之前时，`protected` 关键字指定基类的公共成员和受保护成员是其派生类的受保护成员。

受保护成员不像 `private` 成员那样是专用的（仅对从中声明它们的类的成员可访问），但受保护成员也不像 `public` 成员那样是公开的（在任何函数中均可访问）。

同样被声明为 `static` 的受保护成员对派生类的任何友元或成员函数均是可访问的。 未被声明为 `static` 的受保护成员对派生类中的友元或成员函数是可访问的，但只能通过指向派生类的指针、对派生类的引用或派生类的对象来访问。

若要获得相关信息，请参阅控制对类成员的访问中的 friend、public、private 和成员访问表。

/clr 专用

在 CLR 类型中，C++ 访问说明符关键字（`public`、`private` 和 `protected`）可能影响与程序集相关的类型和方法的可见性。 有关详细信息，请参阅成员访问控制。

> ⓘ **注意**

使用 /LN 编译的文件不受此行为的影响。 在这种情况下，所有托管类（公共或私有）都将可见。

END /clr 专用

示例

C++

```
// 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 the derived class
}
```

另请参阅

控制对类成员的访问
关键字

Last updated on 2025/11/07

public （C++）

语法

```
public:
   [member-list]
public base-class
```

备注

当位于类成员列表前面时，`public` 关键字指定这些成员可从任何函数访问。 这适用于声明到下一个访问指示符或类的末尾的所有成员。

当位于基类名称前面时，`public` 关键字指定基类的公共和受保护成员分别是派生类的公共成员和受保护成员。

类中成员的默认访问是私有的。 结构或联合中成员的默认访问是公共的。

基类的默认访问对于类是私有的，而对于结构是公共的。 联合不能具有基类。

有关详细信息，请参阅 private、protected、friend 以及控制对类成员的访问中的成员访问表。

/clr 专用

在 CLR 类型中，C++ 访问说明符关键字（`public`、`private` 和 `protected`）可能影响与程序集相关的类型和方法的可见性。 有关详细信息，请参阅成员访问控制。

> **注意**
>
> 使用 /LN 编译的文件不受此行为的影响。 在这种情况下，所有托管类（公共或私有）都将可见。

END /clr 专用

示例

C++

```cpp
// 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
}
```

另请参阅

控制对类成员的访问
关键字

Last updated on 2025/11/07

大括号初始化

并不总是需要为 `class` 定义构造函数，特别是相对比较简单的类。 用户可以使用统一初始化来初始化 `class` 或 `struct` 的对象，如下面的示例所示：

C++

```
// 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 };

    // When there's no constructor, an empty brace initializer does
    // value initialization = {0,0,0,0,0}
    TempData td_emptyInit{};

    // 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;
}
```

如果 `class` 或 `struct` 没有构造函数，请按 `class` 中声明的成员的顺序提供列表元素。 如果 `class` 具有构造函数，请按参数顺序提供元素。 如果类型具有隐式或显式声明的默认构造函数，你可以使用大括号初始化（具有空大括号）对其进行调用。 例如，可通过使用空大括号和非空大括号初始化来初始化以下 `class`：

C++

```
#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);
}
```

如果类具有非默认构造函数，则类成员在大括号初始值设定项中的显示顺序是对应参数在构造函数中的显示顺序，而不是成员的声明顺序（如上一示例中的 `class_a` 一样）。 否则，如果类型没有声明的构造函数，则成员初始值设定项必须按声明的顺序显示在大括号初始值设定项中。 在这种情况下，可以根据需要初始化尽可能多的公共成员，但不能跳过任何成员。 以下示例演示了在无声明的构造函数时在大括号初始化中使用的顺序：

C++

```
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
}
```

如果默认构造函数已显式声明，但标记为“已删除”，则无法使用空大括号初始化：

C++

```
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
}
```

可以在通常进行初始化的任何位置使用大括号初始化 - 例如，初始化为函数参数或返回值，或使用 `new` 关键字初始化：

C++

```
class_d* cf = new class_d{4.5};
kr->add_d({ 4.5 });
return { 4.5 };
```

在 `/std:c++17` 模式和更高版本中，空大括号初始化的规则略有限制。 请参阅派生构造函数和扩展聚合初始化。

initializer_list 构造函数

initializer_list 类表示可以在构造函数和其他上下文中使用的指定类型的对象的列表。 您可通过使用大括号初始化构造 initializer_list：

C++

```
initializer_list<int> int_list{5, 6, 7};
```

> ⓘ **重要**
>
> 若要使用此类，必须包括 <initializer_list> 标头。

可以复制 `initializer_list`。 在这种情况下，新列表的成员是对原始列表成员的引用：

C++

```
initializer_list<int> ilist1{ 5, 6, 7 };
initializer_list<int> ilist2( ilist1 );
if (ilist1.begin() == ilist2.begin())
    cout << "yes" << endl; // expect "yes"
```

标准库容器类以及 `string`、`wstring` 和 `regex` 具有 `initializer_list` 构造函数。 以下示例演示如何使用这些构造函数执行大括号初始化：

C++

```
vector<int> v1{ 9, 10, 11 };
map<int, string> m1{ {1, "a"}, {2, "b"} };
string s{ 'a', 'b', 'c' };
regex rgx{ 'x', 'y', 'z' };
```

另请参阅

类和结构
构造函数

Last updated on 2025/11/07

对象生存期和资源管理 (RAII)

与托管语言不同，C++ 没有自动回收垃圾，这是在程序运行时释放堆内存和其他资源的一个内部进程。 C++ 程序负责将所有已获取的资源返回到操作系统。 未能释放未使用的资源称为“泄漏”。 在进程退出之前，泄漏的资源无法用于其他程序。 特别是内存泄漏是 C 样式编程中 bug 的常见原因。

新式 C++ 通过声明堆栈上的对象，尽可能避免使用堆内存。 当某个资源对于堆栈来说太大时，则它应由对象拥有。 当该对象初始化时，它会获取它拥有的资源。 然后，该对象负责在其析构函数中释放资源。 在堆栈上声明拥有资源的对象本身。 对象拥有资源的原则也称为“资源获取即初始化”(RAII)。

当拥有资源的堆栈对象超出范围时，会自动调用其析构函数。 这样，C++ 中的垃圾回收与对象生存期密切相关，是确定性的。 资源始终在程序中的已知点发布，你可以控制该点。 仅类似 C++ 中的确定析构函数可公平处理内存和非内存资源。

下面的示例显示了简单查询 `w`。 它在函数范围内的堆栈上声明，并在函数块的末尾销毁。 对象 `w` 没有资源（例如堆分配的内存）。 它的唯一成员 `g` 本身在堆栈上声明，并且与 `w` 一起超出范围。 `widget` 析构函数中不需要特殊代码。

C++

```
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(); }"
```

在以下示例中，`w` 拥有内存资源，因此必须在其析构函数中具有代码才能删除内存。

C++

```
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
```

自 C++11 以来，有一种更好的方法可以编写前面的示例：使用标准库中的智能指针。 智能指针可处理对其拥有的内存的分配和删除。 使用智能指针将不需要在 `widget` 类中显式析构函数。

C++

```
#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
```

使用智能指针进行内存分配，可以消除内存泄漏的可能性。 此模型适用于其他资源，例如文件句柄或套接字。 可以在你的类中以类似的方式管理自己的资源。 有关详细信息，请参阅智能指针。

C++ 的设计可确保对象在超出范围时被销毁。 也就是说，它们在块被退出时以与构造相反的顺序被摧毁。 销毁对象时，将按特定顺序销毁其基项和成员。 在全局范围内在任何块之外声明的对象可能会导致问题。 如果全局对象的构造函数引发异常，将很难调试。

另请参阅

欢迎回到 C++
C++ 语言参考
C++ 标准库

Last updated on 2025/11/07

用于编译时封装的 Pimpl（现代 C++）

pimpl idiom 是一种新式 C++ 技术，用于隐藏实现、最小化耦合和分离接口。 Pimpl 是“指向实现的指针”的缩写。你可能已经熟悉这个概念，但知道它的其他名称，如柴郡猫或编译器防火墙成语。

为何使用 pimpl?

下面是 pimpl idiom 如何改善软件开发生命周期：

- 编译依赖项的最小化。
- 接口和实现的分离。
- 可移植性。

Pimpl 标头

C++

```
// my_class.h
class my_class {
   //  ... all public and protected stuff goes here ...
private:
   class impl; unique_ptr<impl> pimpl; // opaque type here
};
```

pimpl idiom 避免重新生成级联和脆对象布局。 它非常适合（可传递）常用类型。

Pimpl 实现

在 .cpp 文件中定义 `impl` 类。

C++

```
// 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 ...
}
```

最佳实践

请考虑是否添加对非引发交换专用化的支持。

另请参阅

欢迎回到 C++
C++ 语言参考
C++ 标准库

Last updated on 2025/11/07

ABI 边界处的可移植性

在二进制接口边界使用可充分移植的类型和约定。 “可移植类型”是 C 内置类型或只包含 C 内置类型的结构。 类类型只能在调用方和被调用方接受布局、调用约定等时使用。这仅在两者是使用相同的编译器和编译器设置编译时可能。

如何为了 C 可移植性平展类

如果调用方可能是使用另一编译器/语言编译的，则使用特定调用约定“平展”为“外部 C”API：

C++

```
// 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"
}
```

另请参阅

欢迎回到 C++
C++ 语言参考
C++ 标准库

Last updated on 2025/11/07

构造函数 (C++)

若要自定义类初始化其成员的方式，或是如要在创建类的对象时调用函数，请定义构造函数。 构造函数具有与类相同的名称，没有返回值。 可以定义所需数量的重载构造函数，以各种方式自定义初始化。 通常，构造函数具有公共可访问性，以便类定义或继承层次结构外部的代码可以创建类的对象。 但也可以将构造函数声明为 `protected` 或 `private`。

构造函数可以选择采用成员初始化表达式列表。 与在构造函数主体中赋值相比，初始化类成员是更高效的方式。 以下示例演示具有三个重载构造函数的类 `Box`。 最后两个构造函数使用成员初始化列表：

C++

```
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 };
};
```

声明类的实例时，编译器会基于重载决策选择要调用的构造函数：

C++

```
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)
}
```

- 构造函数可以声明为 `inline`、explicit、friend 或 constexpr。
- 构造函数可以初始化一个已声明为 `const`、`volatile` 或 `const volatile` 的对象。 该对象在构造函数完成之后成为 `const`。
- 若要在实现文件中定义构造函数，请为它提供限定名称，如同任何其他成员函数一样：`Box::Box(){...}`。

成员初始化表达式列表

构造函数可以选择具有成员初始化表达式列表，该列表会在构造函数主体运行之前初始化类成员。 （成员初始化表达式列表与类型为 的初始化表达式列表不同。std::initializer_list<T>）

首选成员初始化表达式列表，而不是在构造函数主体中赋值。 成员初始化表达式列表直接初始化成员。 以下示例演示了成员初始化表达式列表，该列表由冒号后的所有 `identifier(argument)` 表达式组成：

C++

```
    Box(int width, int length, int height)
        : m_width(width), m_length(length), m_height(height)
    {}
```

标识符必须引用类成员；它使用参数的值进行初始化。 参数可以是构造函数参数之一、函数调用或 std::initializer_list<T>。

`const` 成员和引用类型的成员必须在成员初始化表达式列表中进行初始化。

若要确保在派生构造函数运行之前完全初始化基类，请调用初始化表达式列表中的任何参数化基类构造函数。

默认构造函数

默认构造函数通常没有参数，但它们可以具有带默认值的参数。

C++

```
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){}
...
}
```

默认构造函数是特殊成员函数之一。 如果类中未声明构造函数，则编译器提供隐式 `inline` 默认构造函数。

C++

```
#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
}
```

如果你依赖于隐式默认构造函数，请确保在类定义中初始化成员，如前面的示例所示。 如果没有这些初始化表达式，成员会处于未初始化状态，Volume() 调用会生成垃圾值。 一般而言，即使不依赖于隐式默认构造函数，也最好以这种方式初始化成员。

可以通过将隐式默认构造函数定义为已删除来阻止编译器生成它：

C++

```
    // Default constructor
    Box() = delete;
```

如果有任何类成员不是默认可构造，则编译器生成的默认构造函数会定义为已删除。 例如，类类型的所有成员及其类类型成员都必须具有可访问的默认构造函数和析构函数。 引用类型的所有数据成员和所有 `const` 成员都必须具有默认成员初始化表达式。

调用编译器生成的默认构造函数并尝试使用括号时，系统会发出警告：

C++

```
class myclass{};
int main(){
myclass mc();     // warning C4930: prototyped function not called (was a variable
```

```
definition intended?)
}
```

此语句是“最棘手的分析”问题的示例。 可以将 `myclass md();` 解释为函数声明或是对默认构造函数的调用。 因为 C++ 分析程序更偏向于声明，因此表达式会被视为函数声明。 有关详细信息，请参阅最棘手的分析。

如果声明了任何非默认构造函数，编译器不会提供默认构造函数：

C++

```
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
}
```

如果类没有默认构造函数，则无法通过单独使用方括号语法来构造该类的对象数组。 例如，在前面提到的代码块中，Box 数组无法进行如下声明：

C++

```
Box boxes[3]; // C2512: no appropriate default constructor available
```

但是，你可以使用一组初始化表达式列表来初始化 Box 对象数组：

C++

```
Box boxes[3]{ { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };
```

有关详细信息，请参阅初始化表达式。

复制构造函数

复制构造函数通过从相同类型的对象复制成员值来初始化对象。 如果类成员都是简单类型（如标量值），则编译器生成的复制构造函数已够用，你无需定义自己的函数。 如果类需要更复杂的初始化，则需要实现自定义复制构造函数。 例如，如果类成员是指针，则需要定义复制构造函数以分配新内存，并从其他指针指向的对象复制值。 编译器生成的复制构造函数只是复制指针，以便新指针仍指向其他指针的内存位置。

复制构造函数可能具有以下签名之一：

C++

```
    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");
```

定义复制构造函数时，还应定义复制赋值运算符 (=)。 有关详细信息，请参阅赋值以及复制构造函数和复制赋值运算符。

可以通过将复制构造函数定义为已删除来阻止复制对象：

C++

```
    Box (const Box& other) = delete;
```

尝试复制对象会产生错误“C2280: 尝试引用已删除的函数”。

移动构造函数

移动构造函数是特殊成员函数，它将现有对象数据的所有权移交给新变量，而不复制原始数据。 它采用 rvalue 引用作为其第一个参数，以后的任何参数都必须具有默认值。 移动构造函数在传递大型对象时可以显著提高程序的效率。

C++

```
Box(Box&& other);
```

当对象由相同类型的另一个对象初始化时，如果另一对象即将被毁且不再需要其资源，则编译器会选择移动构造函数。 以下示例演示了一种由重载决策选择移动构造函数的情况。 在调用 `get_Box()` 的构造函数中，返回值是 xvalue（过期值）。 它未分配给任何变量，因此即将超出范围。 为了为此示例提供动力，我们为 Box 提供表示其内容的大型字符串向量。 移动构造函数

不会复制该向量及其字符串，而是从过期值“box”中“窃取”它，以便该向量现在属于新对象。 只需调用 `std::move` 即可，因为 `vector` 和 `string` 类都实现自己的移动构造函数。

C++

```
#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 Print_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.Print_Contents(); // Prove that we have all the values

    char ch;
    cin >> ch; // keep window open
    return 0;
}
```

如果类未定义移动构造函数，则在没有用户声明的复制构造函数、复制赋值运算符、移动赋值运算符或析构函数时，编译器会生成隐式构造函数。 如果未定义显式或隐式移动构造函数，则原本使用移动构造函数的操作会改用复制构造函数。 如果类声明了移动构造函数或移动赋值运算符，则隐式声明的复制构造函数会定义为已删除。

如果作为类类型的任何成员缺少析构函数或是如果编译器无法确定要用于移动操作的构造函数，则隐式声明的移动构造函数会定义为已删除。

有关如何编写不常用的移动构造函数的详细信息，请参阅移动构造函数和移动赋值运算符 (C++)。

显式默认构造函数和已删除构造函数

你可以显式设置默认复制构造函数、设置默认构造函数、移动构造函数、复制赋值运算符、移动赋值运算符和析构函数。 你可以显式删除所有特殊成员函数。

C++

```
class Box2
{
public:
    Box2() = delete;
    Box2(const Box2& other) = default;
    Box2& operator=(const Box2& other) = default;
    Box2(Box2&& other) = default;
    Box2& operator=(Box2&& other) = default;
    //...
};
```

有关详细信息，请参阅显式默认设置的函数和已删除的函数。

constexpr 构造函数

构造函数在以下情况下可能会声明为 constexpr

- 声明为默认，或是在一般情况下满足 constexpr 函数 的所有条件；
- 类没有虚拟基类；
- 每个参数都是文本类型；
- 主体不是函数 try 块；
- 所有非静态数据成员和基类子对象都会进行初始化；
- 如果类是 (a) 具有变体成员的联合，或是 (b) 具有匿名联合，则只有一个联合成员会进行初始化；
- 类类型的每个非静态数据成员以及所有基类子对象都具有 constexpr 构造函数

初始化表达式列表构造函数

如果某个构造函数采用 std::initializer_list<T> 作为其参数，并且任何其他参数都具有默认自变量，则当类通过直接初始化来实例化时，会在重载决策中选择该构造函数。 可以使用 initializer_list 初始化可接受它的任何成员。 例如，假设前面演示的 Box 类具有 `std::vector<string>` 成员 `m_contents`。 可以提供如下所示的构造函数：

C++

```
    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)
{}
```

随后创建如下所示的 Box 对象：

C++

```
    Box b{ "apples", "oranges", "pears" }; // or ...
    Box b2(initializer_list<string> { "bread", "cheese", "wine" }, 2, 4, 6);
```

显式构造函数

如果类具有带一个参数的构造函数，或是如果除了一个参数之外的所有参数都具有默认值，则参数类型可以隐式转换为类类型。 例如，如果 `Box` 类具有一个类似于下面这样的构造函数：

C++

```
Box(int size): m_width(size), m_length(size), m_height(size){}
```

可以初始化 Box，如下所示：

```C++
Box b = 42;
```

或将一个 int 传递给采用 Box 的函数：

```C++
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);
```

这类转换可能在某些情况下很有用，但更常见的是，它们可能会导致代码中发生细微但严重的错误。 作为一般规则，应对构造函数（和用户定义的运算符）使用 `explicit` 关键字以防止出现这种隐式类型转换：

```C++
explicit Box(int size): m_width(size), m_length(size), m_height(size){}
```

构造函数是显式函数时，此行会导致编译器错误：`ShippingOrder so(42, 10.8);`。 有关详细信息，请参阅用户定义的类型转换。

构造函数顺序

构造函数按此顺序执行工作：

1. 按声明顺序调用基类和成员构造函数。
2. 如果类派生自虚拟基类，则会将对象的虚拟基指针初始化。
3. 如果类具有或继承了虚函数，则会将对象的虚函数指针初始化。 虚函数指针指向类中的虚函数表，确保虚函数正确地调用绑定代码。
4. 它执行自己函数体中的所有代码。

下面的示例显示，在派生类的构造函数中，基类和成员构造函数的调用顺序。 首先，调用基构造函数。 然后，按照在类声明中出现的顺序初始化基类成员。 最后，调用派生构造函数。

C++

```cpp
#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;
}
```

输出如下：

Output

```
Contained1 ctor
Contained2 ctor
BaseContainer ctor
Contained3 ctor
DerivedContainer ctor
```

派生类构造函数始终调用基类构造函数，因此，在完成任何额外任务之前，它可以依赖于完全构造的基类。 基类构造函数按派生顺序进行调用 — 例如，如果 `ClassA` 派生自 `ClassB`，而后者派

生自 `ClassC`，那么首先调用 `ClassC` 构造函数，然后调用 `ClassB` 构造函数，最后调用 `ClassA` 构造函数。

如果基类没有默认构造函数，则必须在派生类构造函数中提供基类构造函数参数：

C++

```
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);
}
```

如果构造函数引发异常，析构的顺序与构造的顺序相反：

1. 构造函数主体中的代码将展开。
2. 基类和成员对象将被销毁，顺序与声明顺序相反。
3. 如果是非委托构造函数，所有完全构造的基类对象和成员均会销毁。 但是，对象本身不是完全构造的，因此析构函数不会运行。

派生构造函数和扩展聚合初始化

如果基类的构造函数是非公共的，但可由派生类进行访问，那么在 `/std:c++17` 模式及更高版本模式下的 Visual Studio 2017 及更高版本中，无法使用空括号来初始化派生类型的对象。

以下示例演示 C++14 一致行为：

C++

```
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.
```

在 C++17， `Derived` 现被视作聚合类型。 这意味着 `Base` 通过私有默认构造函数进行的初始化将作为扩展的聚合初始化规则的一部分而直接发生。 以前， `Base` 私有构造函数通过 `Derived` 构造函数调用，它之所以能够成功是因为 `friend` 声明。

以下示例展示了在 `/std:c++17` 模式下的 Visual Studio 2017 及更高版本中的 C++17 行为：

C++

```
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': can't access
               // private member declared in class 'Base'
```

具有多重继承的类的构造函数

如果类从多个基类派生，那么将按照派生类声明中列出的顺序调用基类构造函数：

C++

```
#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;
}
```

你应看到以下输出：

Output

```
BaseClass1 ctor
BaseClass2 ctor
BaseClass3 ctor
DerivedClass ctor
```

委托构造函数

委托构造函数调用同一类中的其他构造函数，以完成部分初始化工作。 在具有多个全都必须执行类似工作的构造函数时，此功能非常有用。 可以在一个构造函数中编写主逻辑，并从其他构造函数调用它。 在以下简单示例中，Box(int) 将其工作委托给 Box(int,int,int)：

C++

```
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
};
```

所有构造函数完成后，完全初始化的构造函数将立即创建对象。 有关详细信息，请参阅委托构造函数。

继承构造函数 (C++11)

派生类可以使用 `using` 声明从直接基类继承构造函数，如下面的示例所示：

C++

```
#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()*/
```

Visual Studio 2017 及更高版本：`using` 模式及更高版本模式下的 `/std:c++17` 语句可将来自基类的所有构造函数引入范围（除了签名与派生类中的构造函数相同的构造函数）。 一般而言，当派生类未声明新数据成员或构造函数时，最好使用继承构造函数。

如果类型指定基类，则类模板可以从类型参数继承所有构造函数：

C++

```
template< typename T >
class Derived : T {
    using T::T;   // declare the constructors from T
    // ...
};
```

如果基类的构造函数具有相同签名，则派生类无法从多个基类继承。

构造函数和复合类

包含类类型成员的类称为“复合类”。 创建复合类的类类型成员时，调用类自己的构造函数之前，先调用构造函数。 当包含的类没有默认构造函数是，必须使用复合类构造函数中的初始化列表。 在之前的 `StorageBox` 示例中，如果将 `m_label` 成员变量的类型更改为新的 `Label` 类，则必须调用基类构造函数，并且将 `m_label` 变量（位于 `StorageBox` 构造函数中）初始化：

C++

```
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"});
}
```

本部分内容

- 复制构造函数和复制赋值运算符
- 移动构造函数和移动赋值运算符
- 委托构造函数

另请参阅

类和结构

Last updated on 2025/11/07

复制构造函数和复制赋值运算符 (C++)

> ⓘ **注意**
>
> 从 C++11 中开始，该语言支持两类赋值：*复制赋值*和*移动赋值*。 在本文中，“赋值”意味着复制赋值，除非有其他显式声明。 有关移动赋值的信息，请参阅**移动构造函数和移动赋值运算符 (C++)**。
>
> 赋值操作和初始化操作都会导致对象被复制。

- **赋值**：将一个对象的值分配给另一个对象时，第一个对象将复制到第二个对象。 因此，此代码将 `b` 的值复制到 `a`：

  C++
  ```
  Point a, b;
  ...
  a = b;
  ```

- **初始化**：在声明新对象、按值传递函数参数或从函数返回值时，将发生初始化。

您可以为类类型的对象定义“复制”的语义。 例如，假设有以下代码：

C++
```
TextFile a, b;
a.Open( "FILE1.DAT" );
b.Open( "FILE2.DAT" );
b = a;
```

前面的代码可能表示“将 FILE1.DAT 的内容复制到 FILE2.DAT”，也可能表示“忽略 FILE2.DAT 并使 `b` 成为 FILE1.DAT 的另一个句柄”。必须将合适的复制语义附加到每个类，如下所示：

- 使用返回对类类型的引用的赋值运算符 **`operator=`**，并采用 **`const`** 引用传递的一个参数，例如 `ClassName& operator=(const ClassName& x);`。

- 使用复制构造函数。

如果不声明复制构造函数，编译器将为你生成成员的复制构造函数。 同样，如果不声明复制赋值运算符，编译器将为你生成成员的复制赋值运算符。 声明复制构造函数不会取消编译器生成的复制赋值运算符，反之亦然。 如果实现其中任一方法，我们建议也实现另一个。 实现这两者时，代码的含义是明确的。

复制构造函数采用 `ClassName&` 类型的参数，其中 *`ClassName`* 是类的名称。 例如：

C++

```cpp
// spec1_copying_class_objects.cpp
class Window
{
public:
    Window( const Window& );            // Declare copy constructor.
    Window& operator=(const Window& x); // Declare copy assignment.
    // ...
};

int main()
{
}
```

> **注意**
>
> 尽可能创建复制构造函数的参数 `const ClassName&` 的类型。 这可防止复制构造函数意外更改复制的对象。 它还允许从 `const` 对象复制。

编译器生成的构造函数

编译器生成的复制构造函数（如用户定义的复制构造函数）具有单个自变量类型“对 class-name 的引用”。当所有基类和成员类都具有声明为采用 `const` *class-name*& 类型的单个自变量的复制构造函数时，将引发异常。 在这种情况下，编译器生成的复制构造函数的自变量也是 `const`。

当复制构造函数的自变量类型不是 `const` 时，复制 `const` 对象进行初始化将产生错误。 反之则不然：如果自变量是 `const`，可以复制不是 `const` 的对象进行初始化。

编译器生成的赋值运算符遵循 `const` 的相同模式。 除非所有基类和成员类中的赋值运算符都采用 `ClassName&` 类型的自变量，否则它们将采用 `const ClassName&` 类型的单个自变量。 在这种情况下，类的生成的赋值运算符采用 `const` 自变量。

> **注意**
>
> 当虚拟基类由复制构造函数（编译器生成或用户定义的）初始化时，将只初始化一次：在构造它们时。

含义类似于复制构造函数的含义。 当自变量类型不是 `const` 时，从 `const` 对象赋值将产生错误。 反之则不然：如果将 `const` 值赋给不是 `const` 的值，则赋值能成功。

有关重载赋值运算符的详细信息，请参阅赋值。

Last updated on 2025/11/07

移动构造函数和移动赋值运算符 (C++)

本主题介绍如何为 C++ 类编写*移动构造函数*和移动赋值运算符。 移动构造函数使右值对象拥有的资源无需复制即可移动到左值中。 有关移动语义的详细信息，请参阅 Rvalue 引用声明符：&&。

此主题基于用于管理内存缓冲区的 C++ 类 `MemoryBlock`。

C++

```
// 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.
};
```

以下过程介绍如何为示例 C++ 类编写移动构造函数和移动赋值运算符。

为 C++ 创建移动构造函数

1. 定义一个空的构造函数方法，该方法采用一个对类类型的右值引用作为参数，如以下示例所示：

   C++

   ```
   MemoryBlock(MemoryBlock&& other)
       : _data(nullptr)
       , _length(0)
   {
   }
   ```

2. 在移动构造函数中，将源对象中的类数据成员添加到要构造的对象：

   C++

```
_data = other._data;
_length = other._length;
```

3. 将源对象的数据成员分配给默认值。 这可以防止析构函数多次释放资源（如内存）：

```C++
other._data = nullptr;
other._length = 0;
```

为 C++ 类创建移动赋值运算符

1. 定义一个空的赋值运算符，该运算符采用一个对类类型的右值引用作为参数并返回一个对类类型的引用，如以下示例所示：

```C++
MemoryBlock& operator=(MemoryBlock&& other)
{
}
```

2. 在移动赋值运算符中，如果尝试将对象赋给自身，则添加不执行运算的条件语句。

```C++
if (this != &other)
{
}
```

3. 在条件语句中，从要将其赋值的对象中释放所有资源（如内存）。

   以下示例从要将其赋值的对象中释放 `_data` 成员：

```C++
// Free the existing resource.
delete[] _data;
```

   执行第一个过程中的步骤 2 和步骤 3 以将数据成员从源对象转移到要构造的对象：

```C++
// 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. 返回对当前对象的引用，如以下示例所示：

```C++
return *this;
```

示例：完成移动构造函数和赋值运算符

以下示例显示了 `MemoryBlock` 类的完整移动构造函数和移动赋值运算符：

```C++
// Move constructor.
MemoryBlock(MemoryBlock&& other) noexcept
   : _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) noexcept
{
   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;
}
```

示例 使用移动语义来提高性能

以下示例演示移动语义如何能提高应用程序的性能。 此示例将两个元素添加到一个矢量对象，然后在两个现有元素之间插入一个新元素。 `vector` 类使用移动语义，通过移动矢量元素（而非复制它们）来高效地执行插入操作。

C++

```
// 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));
}
```

该示例产生下面的输出：

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 = 75. Moving resource.
In MemoryBlock(MemoryBlock&&). length = 25. Moving resource.
In ~MemoryBlock(). length = 0.
In ~MemoryBlock(). length = 0.
In MemoryBlock(size_t). length = 50.
In MemoryBlock(MemoryBlock&&). length = 50. Moving resource.
```

```
In MemoryBlock(MemoryBlock&&). length = 25. Moving resource.
In MemoryBlock(MemoryBlock&&). length = 75. Moving resource.
In ~MemoryBlock(). length = 0.
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.
```

在 Visual Studio 2010 之前，此示例生成了以下输出：

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.
```

使用移动语义的此示例版本比不使用移动语义的版本更高效，因为前者执行的复制、内存分配和内存释放操作更少。

可靠编程

若要防止资源泄漏，请始终释放移动赋值运算符中的资源（如内存、文件句柄和套接字）。

若要防止不可恢复的资源损坏，请正确处理移动赋值运算符中的自我赋值。

如果为你的类同时提供了移动构造函数和移动赋值运算符，则可以编写移动构造函数来调用移动赋值运算符，从而消除冗余代码。 以下示例显示了调用移动赋值运算符的移动构造函数的修改后的版本：

C++

```
// Move constructor.
MemoryBlock(MemoryBlock&& other) noexcept
```

```
    : _data(nullptr)
    , _length(0)
{
    *this = std::move(other);
}
```

std::move 函数将左值 `other` 转换为右值。

另请参阅

引用声明符：&&
std::move

Last updated on 2025/11/07

委托构造函数

许多类具有执行类似操作（例如，验证参数）的多个构造函数：

C++

```
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;
    }
};
```

你可以通过添加一个执行所有验证的函数来减少重复的代码，但是如果一个构造函数可以将部分工作委托给其他构造函数，则 `class_c` 的代码更易于了解和维护。 若要添加委托构造函数，请使用 `constructor (. . .) : constructor (. . .)` 语法：

C++

```
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 };
}
```

当您单步调试上一示例时，请注意，构造函数 `class_c(int, int, int)` 首先调用构造函数 `class_c(int, int)`，该构造函数反过来调用 `class_c(int)`。 每个构造函数将仅执行其他构造函数不会执行的工作。

调用的第一个构造函数将初始化对象，以便此时初始化其所有成员。 不能在委托给其他构造函数的构造函数中执行成员初始化，如下所示：

C++

```
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;
};
```

下一示例演示非静态数据成员初始值设定项的使用。 请注意，如果构造函数还将初始化给定数据成员，则将重写成员初始值设定项：

C++

```
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;
}
```

构造函数委托语法不会阻止意外创建构造函数递归 - Constructor1 将调用 Constructor2（其调用 Constructor1），在出现堆栈溢出之前不会出错。 您应当避免循环。

C++

```cpp
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() { }
};
```

Last updated on 2025/11/07

析构函数 (C++)

析构函数是一个成员函数，在对象超出范围或通过调用 `delete` 或 `delete[]` 显式销毁对象时，会自动调用析构函数。 析构函数与类同名，前面带有波形符 (`~`)。 例如，声明 `String` 类的析构函数： `~String()`。

如果你未定义析构函数，编译器会提供一个默认的析构函数；对于某些类来说，这就足够了。 当类维护必须显式释放的资源（例如系统资源的句柄，或指向在类的实例被销毁时应释放的内存的指针）时，你需要定义一个自定义的析构函数。

请考虑 `String` 类的以下声明：

C++

```
// spec1_destructors.cpp
#include <string> // strlen()

class String
{
    public:
        String(const char* ch);  // Declare the constructor
        ~String();               // Declare the destructor
    private:
        char* _text{nullptr};
};

// Define the constructor
String::String(const char* ch)
{
    size_t sizeOfText = strlen(ch) + 1; // +1 to account for trailing NULL

    // 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 the string.
    delete[] _text;
}

int main()
{
```

```
    String str("We love C++");
}
```

在前面的示例中，析构函数 `String::~String` 使用 `delete[]` 运算符来动态释放为文本存储分配的空间。

声明析构函数

析构函数是具有与类相同的名称但前面是波形符 (`~`) 的函数

多个规则管理析构函数的声明。 析构函数：

- 不接受自变量。
- 不返回值（或`void`）。
- 无法声明为`const`、`volatile`或`static`。 但是，可以调用它们来进行声明为 `const`、`volatile` 或 `static` 的对象的析构。
- 可以声明为 `virtual`。 通过使用虚拟析构函数，无需知道对象的类型即可销毁对象—使用虚函数机制调用该对象的正确析构函数。 析构函数也可以声明为抽象类的纯虚函数。

使用构造函数

当下列事件之一发生时，将调用析构函数：

- 具有块范围的本地（自动）对象超出范围。
- 使用 `delete` 来解除分配使用 `new` 分配的对象。 使用 `delete[]` 会导致未定义的行为。
- 使用 `delete[]` 来解除分配使用 `new[]` 分配的对象。 使用 `delete` 会导致未定义的行为。
- 临时对象的生存期结束。
- 程序结束，并且存在全局或静态对象。
- 使用析构函数的完全限定名显式调用了析构函数。

析构函数可以随意调用类成员函数和访问类成员数据。

析构函数的使用有两个限制：

- 你不能拿走它的地址。

- 派生类不继承其基类的析构函数。

破坏的顺序

当对象超出范围或被删除时，其完整析构中的事件序列如下所示：

1. 将调用该类的析构函数，并且会执行该析构函数的主体。
2. 非静态成员对象的析构函数在调用时，其顺序与它们在类声明中的出现顺序相反。 用于这些成员的构造的可选成员初始化列表不影响构造或析构的顺序。
3. 非虚拟基类的析构函数以声明的相反顺序被调用。
4. 虚拟基类的析构函数以声明的相反顺序被调用。

C++

```cpp
// 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;
}
```

输出

```
A3 dtor
A2 dtor
A1 dtor

B1 dtor

B3 dtor
B2 dtor
B1 dtor
```

虚拟基类

按照与虚拟基类在定向非循环图形中显示的顺序的相反顺序调用这些虚拟基类的析构函数（深度优先、从左到右、后序遍历）。 下图描述了继承关系图。

下面列出了图中显示的类的类定义：

```C++
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 {};
```

为了确定 `E` 类型的对象的虚拟基类的析构顺序，编译器将通过应用以下算法来生成列表：

1. 向左遍历关系图，并从关系图中的最深点开始（在此示例中，为 `E`）。
2. 执行左移遍历，直到访问了所有节点。 记下当前节点的名称。
3. 重新访问上一个节点（向下并向右）以查明要记住的节点是否为虚拟基类。
4. 如果记住的节点是虚拟基类，请浏览列表以查看是否已将其输入。 如果它不是虚拟基类，则将其忽略。
5. 如果记住的节点尚未包含在列表中，请将其添加到列表的底部。
6. 向上遍历关系图并沿下一个路径向右遍历。
7. 转到步骤 2。
8. 在用完最后一个向上路径时，请记下当前节点的名称。
9. 转到步骤 3。
10. 继续执行此过程，直到底部节点再次成为当前节点。

因此，对于 `E` 类，析构顺序为：

1. 非虚拟基类 `E`。
2. 非虚拟基类 `D`。
3. 非虚拟基类 `C`。
4. 虚拟基类 `B`。
5. 虚拟基类 `A`。

此过程将生成唯一条目的有序列表。 任何类名均不会出现两次。 在构造列表后，将以相反的顺序遍历该列表，并且将调用列表中每个类（从最后一个到第一个）的析构函数。

如果某个类中的构造函数或析构函数依赖于另一个组件必须先被创建或保持更长时间的存在—例如，上图中所示，如果`A`的析构函数在执行时依赖`B`仍然存在，或者反之亦然—那么构造或析构的顺序就显得尤为重要。

继承关系图中各个类之间的这种相互依赖项本质上是危险的，因为稍后派生类可以更改最左边的路径，从而更改构造和析构的顺序。

非虚拟基类

非虚拟基类的析构函数将按照声明中基类名称的相反顺序被调用。 考虑下列类声明：

```
C++
class MultInherit : public Base1, public Base2
...
```

在前面的示例中， `Base2` 的析构函数先于 `Base1` 的析构函数被调用。

显式析构函数调用

很少需要显式调用析构函数。 但是，对置于绝对地址的对象进行清理会很有用。 这些对象通常使用采用位置参数的用户定义的 `new` 运算符进行分配。 `delete`运算符不能释放该内存，因为它不是从自由存储区分配的（有关详细信息，请参阅new 和 delete 运算符）。 但是，对析构函数的调用可以执行相应的清理。 若要显式调用 `s` 类的对象 `String` 的析构函数，请使用下列语句之一：

```
C++
s.String::~String();     // non-virtual call
ps->String::~String();   // non-virtual call

s.~String();       // Virtual call
ps->~String();     // Virtual call
```

可以使用如上所示的析构函数的显式调用语法，无论类型是否定义了析构函数。 这允许您进行此类显式调用，而无需了解是否为此类型定义了析构函数。 显式调用析构函数，其中未定义的析构函数无效。

可靠编程

如果类获得了资源，就需要析构函数，为了安全地管理资源，它可能必须实现复制构造函数和复制赋值。

如果用户未定义这些特殊函数，编译器会对其进行隐式定义。 隐式生成的构造函数和赋值运算符执行浅层的、成员式的复制，如果一个对象管理一个资源，这基本上就可以确定是错的。

在下一个示例中，隐式生成的复制构造函数将使指针 `str1.text` 和 `str2.text` 引用同一内存，当我们从 `copy_strings()` 返回时，该内存将被删除两次，这是未定义的行为：

C++

```
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
```

显式定义析构函数、复制构造函数或复制赋值运算符可防止对移动构造函数和移动赋值运算符进行隐式定义。 在这种情况下，如果复制成本高昂，未能提供移动操作通常意味着错过了优化的机会。

另请参阅

复制构造函数和复制赋值运算符
移动构造函数和移动赋值运算符

Last updated on 2025/11/07

成员函数概述

成员函数是静态或非静态的。 静态成员函数的行为与其他成员函数的行为不同，因为静态成员函数不具有隐式 `this` 参数。 非静态成员函数具有 `this` 指针。 可以在类声明的内部或外部定义成员函数（无论是静态的还是非静态的）。

如果在类声明的内部定义一个成员函数，则该函数会被视为内联函数，并且不需要用其类名来限定函数名称。 尽管已将类声明中定义的函数视为内联函数，但你可以使用 `inline` 关键字来记录代码。

在类声明中声明函数的示例如下所示：

C++

```
// 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()
{
}
```

如果成员函数的定义在类声明的外部，则仅在将该函数显式声明为 `inline` 时才将其视为内联函数。 此外，必须通过范围解析运算符（`::`）用类名称限定定义中的函数名称。

以下示例与类 `Account` 的以前的声明等效，只不过 `Deposit` 函数是在类声明的外部定义的：

C++

```
// 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()
{
}
```

> **注意**
>
> 虽然成员函数既可在类声明的内部进行定义也可单独进行定义，但在定义类后，不能将任何成员函数添加到类中。

包含成员函数的类可具有多个声明，但成员函数本身只能在程序中有一个定义。 多个定义会导致在链接时出现错误消息。 如果类包含内联函数定义，则这些函数定义必须与遵守此“一个定义”规则相同。

Last updated on 2025/11/07

虚拟说明符

虚拟关键字只能应用于非静态类成员函数。 它表示函数的调用绑定将推迟到运行时。 有关详细信息，请参阅虚函数。

Last updated on 2025/11/07

override 说明符

可以使用 override 关键字来指定在基类中重写虚函数的成员函数。

语法

```
function-declaration override;
```

备注

override 仅在成员函数声明之后使用时才是区分上下文的且具有特殊含义；否则，它不是保留的关键字。

示例

使用 override 有助于防止你的代码中出现意外的继承行为。 以下示例演示在未使用 override 的情况下，可能不打算使用派生类的成员函数行为。 编译器不会发出此代码的任何错误。

C++

```
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
};
```

当使用 override 时，编译器会生成错误，而不会在不提示的情况下创建新的成员函数。

C++

```cpp
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()
};
```

若要指定不能重写函数且不能继承类，请使用 final 关键字。

另请参阅

final 说明符
关键字

Last updated on 2025/11/07

final 说明符

可以使用 final 关键字指定无法在派生类中重写的虚函数。 您还可以使用它指定无法继承的类。

语法

```
function-declaration final;
class class-name final base-classes
```

备注

final 只有在函数声明或类名称后使用时才是区分上下文的且具有特殊含义；否则，它不是保留的关键字。

在类声明中使用 final 时，`base-classes` 是声明的可选部分。

示例

下面的示例使用 final 关键字指定无法重写虚函数。

C++

```
class BaseClass
{
    virtual void func() final;
};

class DerivedClass: public BaseClass
{
    virtual void func(); // compiler error: attempting to
                         // override a final function
};
```

有关如何指定可以重写成员函数的信息，请参阅 override 说明符。

下一个示例使用 final 关键字指定无法继承类。

C++

```
class BaseClass final
{
};
```

```
class DerivedClass: public BaseClass // compiler error: BaseClass is
                                     // marked as non-inheritable
{
};
```

另请参阅

关键字
override 说明符

Last updated on 2025/11/07

继承 (C++)

本节解释如何使用派生类生成可扩展的程序。

概述

可使用名为"继承"的机制从现有类派生新类（请参阅单一继承中开头的信息）。 用于派生的类称为特定派生类的"基类"。 使用以下语法声明派生类：

C++

```
class DerivedSingleBase : [virtual] [access-specifier] Base
{
    // member list
};

class DerivedMultipleBases : [virtual] [access-specifier] Base1,
    [virtual] [access-specifier] Base2, ...
{
    // member list
};
```

在类的标记（名称）后面是一个冒号，后面跟着基本规范列表。 以这种方式命名的基类必须已提前声明。 基本规范可包含访问说明符，它是关键字 `public`、`protected` 或 `private` 之一。 这些访问说明符显示在基类名称的前面并且仅适用于该基类。 这些说明符控制派生类使用基类成员的权限。 有关对基类成员的访问的信息，请参阅成员访问控制。 如果访问说明符被省略，则对该基类的访问被视为 `private`。 基本规范可能包含关键字 `virtual` 以指示虚拟继承。 此关键字可能出现在访问说明符前面或后面（如果有）。 如果使用虚拟继承，则基类称为虚拟基类。

可指定多个基类，并用逗号分隔。 如果指定了单个基类，则继承模型为单一继承。 如果指定了多个基类，则继承模型称为多重继承。

本文包含以下主题：

- 单一继承
- 多个基类
- 虚函数
- 显式替代
- 抽象类
- 范围规则摘要

Microsoft 专用

本节记录了 __super 和 __interface 关键字。

结束 Microsoft 专用

另请参阅

C++ 语言参考

Last updated on 2025/11/07

虚函数

虚函数是应在派生类中重新定义的成员函数。 当使用指针或对基类的引用来引用派生的类对象时，可以为该对象调用虚函数并执行该函数的派生类版本。

虚函数确保为该对象调用正确的函数，这与用于进行函数调用的表达式无关。

假定基类包含声明为 virtual 的函数，并且派生类定义了相同的函数。 为派生类的对象调用派生类中的函数，即使它是使用指针或对基类的引用来调用的。 以下示例显示了一个基类，它提供了 `PrintBalance` 函数和两个派生类的实现

C++

```
// 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();
}
```

在前面的代码中，对 `PrintBalance` 的调用是相同的， `pAccount` 所指向的对象除外。 由于 `PrintBalance` 是虚拟的，因此将调用为每个对象定义的函数版本。 派生类 `PrintBalance` 和 `CheckingAccount` 中的 `SavingsAccount` 函数“重写”基类 `Account` 中的函数。

如果声明的类不提供 `PrintBalance` 函数的重写实现，则使用基类 `Account` 中的默认实现。

派生类中的函数仅在基类中的虚函数的类型相同时重写这些虚函数。 派生类中的函数不能只是与其返回类型中的基类的虚函数不同；参数列表也必须不同。

当使用指针或引用调用函数时，以下规则将适用：

- 根据为其调用的对象的基本类型来解析对虚函数的调用。
- 根据指针或引用的类型来解析对非虚函数的调用。

以下示例说明在通过指针调用时虚函数和非虚函数的行为：

C++

```cpp
// 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.
}
```

Output

```
Derived::NameOf
Invoked by Base
Derived::NameOf
Invoked by Derived
```

请注意，无论 `NameOf` 函数是通过指向 `Base` 的指针还是通过指向 `Derived` 的指针进行调用，它都会调用 `Derived` 的函数。 它调用 `Derived` 的函数，因为 `NameOf` 是虚函数，并且 `pBase` 和 `pDerived` 都指向类型 `Derived` 的对象。

由于仅为类类型的对象调用虚函数，因此不能将全局函数或静态函数声明为 **`virtual`**。

在派生类中声明重写函数时可使用 **`virtual`** 关键字，但它不是必需的；虚函数的重写始终是虚拟的。

必须定义基类中的虚函数，除非使用 pure-specifier 声明它们。 有关纯虚拟函数的详细信息，请参阅 抽象类。

可通过使用范围解析运算符 (`::`) 显式限定函数名称来禁用虚函数调用机制。 考虑先前涉及 `Account` 类的示例。 若要调用基类中的 `PrintBalance`，请使用如下所示的代码：

C++

```
CheckingAccount *pChecking = new CheckingAccount( 100.00 );
```

```
pChecking->Account::PrintBalance();  //  Explicit qualification.

Account *pAccount = pChecking;  // Call Account::PrintBalance

pAccount->Account::PrintBalance();   //  Explicit qualification.
```

在前面的示例中，对 `PrintBalance` 的调用将禁用虚函数调用机制。

Last updated on 2025/11/07

单一继承

在“单继承”（继承的常见形式）中，类仅具有一个基类。 考虑下图中阐释的关系。

简单的单继承关系图

注意该图中从常规到特定的进度。 在大多数类层次结构的设计中发现的另一个常见特性是，派生类与基类具有“某种”关系。 在该图中，`Book` 是一种 `PrintedDocument`，而 `PaperbackBook` 是一种 `book`。

该图中的另一个要注意的是：`Book` 既是派生类（来自 `PrintedDocument`），又是基类（`PaperbackBook` 派生自 `Book`）。 此类类层次结构的框架声明如下面的示例所示：

C++

```
// 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` 被视为 `Book` 的“直接基”类；它是 `PaperbackBook` 的“间接基”类。 差异在于，直接基类出现在类声明的基础列表中，而间接基类不是这样的。

在声明派生的类之前声明从中派生每个类的基类。 为基类提供前向引用声明是不够的；它必须是一个完整声明。

在前面的示例中，使用了访问说明符 **`public`**。 成员访问控制中介绍了公共的、受保护的和私有的继承的含义。

类可用作多个特定类的基类，如下图所示。

有向非循环图示例

在上面显示的名为“有向非循环图”（或“DAG”）的关系图中，一些类是多个派生类的基类。 但反过来却行不通；任何给定的派生类只有一个直接基类。 该图中的关系图描述“单继承”结构。

> ⓘ **注意**
>
> 有向非循环图对于单继承不是唯一的。 它们还用于表示多重继承关系图。

在继承中，派生类包含基类的成员以及您添加的所有新成员。 因此，派生类可以引用基类的成员（除非在派生类中重新定义这些成员）。 当在派生类中重新定义了直接或间接基类的成员时，范围解析运算符 (`::`) 可用于引用这些成员。 请看以下示例：

C++

```
// 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;
}
```

请注意，`Book` 的构造函数（`Book::Book`）具有对数据成员 `Name` 的访问权。 在程序中，可以创建和使用类型为 `Book` 的对象，如下所示：

C++

```
//  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();
```

如前面的示例所示，以相同的方式使用类成员和继承的数据和函数。 如果类 `Book` 的实现调用 `PrintNameOf` 函数的重新实现，则只能通过使用范围解析（`Document`）运算符来调用属于 `::` 类的函数：

C++

```
// 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();
}
```

如果存在可访问的明确基类，则可以隐式将派生类的指针和引用转换为其基类的指针和引用。 下面的代码使用指针演示了此概念（相同的原则适用于引用）：

C++

```
// 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;
}
```

在前面的示例中，创建了不同的类型。 但是，由于这些类型都派生自 `Document` 类，因此存在对 `Document *` 的隐式转换。 因此， `DocLib` 是“异类列表”（其中包含的所有对象并非属于同一类型），该列表包含不同类型的对象。

由于 `Document` 类具有一个 `PrintNameOf` 函数，因此它可以打印库中每本书的名称，但它可能会忽略某些特定于文档类型的信息（`Book` 的页计数、 `HelpFile` 的字节数等）。

> ⓘ **注意**
>
> 强制使用基类来实现函数（如 `PrintNameOf`）通常不是最佳设计。 **虚函数**提供其他设计替代方法。

Last updated on 2025/11/07

基本类

继承过程将创建一个新的派生类，它由基类的成员加上派生类添加的任何新成员组成。 在多重继承中，可以构建一个继承关系图，其中相同的基类是多个派生类的一部分。 下图显示了此类关系图。

单个基类的多个实例

在该图中，显示了 `CollectibleString` 和 `CollectibleSortable` 的组件的图形化表示形式。 但是，基类 `Collectible` 位于通过 `CollectibleSortableString` 路径和 `CollectibleString` 路径的 `CollectibleSortable` 中。 若要消除此冗余，可以在继承此类类时将其声明为虚拟基类。

Last updated on 2025/11/07

多个基类

一个类可以派生自多个基类。 在多重继承模型（其中类派生自多个基类）中，使用 *base-list* 语法元素指定基类。 例如，可以指定派生自 `CollectionOfBook` 和 `Collection` 的 `Book` 的类声明：

C++

```
// deriv_MultipleBaseClasses.cpp
// compile with: /LD
class Collection {};
class Book {};
class CollectionOfBook : public Book, public Collection {
    // New members
};
```

指定基类的顺序并不重要，只不过在某些情况下，将调用构造函数和析构函数。 在这些情况下，指定基类的顺序将影响：

- 调用构造函数的顺序。 如果你的代码依赖要在 `Book` 部分之前初始化的 `CollectionOfBook` 的 `Collection` 部分，则规范的顺序很重要。 初始化按照在 base-list 中指定类的顺序进行。
- 调用析构函数以进行清理的顺序。 同样，如果在销毁另一部分时必须呈现类的特定"部分"，则顺序非常重要。 析构函数的调用顺序与在 base-list 中指定类的顺序相反。

> **注意**
>
> 基类的规范顺序会影响类的内存布局。 不要基于内存中基成员的顺序做出任何编程决策。

在指定 *base-list* 时，不能多次指定同一类名。 但是，可以将类多次作为派生类的间接基。

虚拟基类

由于一个类可能多次成为派生类的间接基类，因此 C++ 提供了一种优化这种基类的工作方式的方法。 虚拟基类提供了一种节省空间和避免使用多重继承的类层次结构中出现多义性的方法。

每个非虚拟对象包含在基类中定义的数据成员的一个副本。 这种重复浪费了空间，并要求您在每次访问基类成员时都必须指定所需的基类成员的副本。

当将某个基类指定为虚拟基时，该基类可以多次作为间接基而无需复制其数据成员。 基类的数据成员的单个副本由将其用作虚拟基的所有基类共享。

声明虚拟基类时，`virtual` 关键字出现在派生类的基列表中。

请考虑下图中的类层次结构，它演示了模拟的午餐排队：

模拟的午餐排队图

在该图中，`Queue` 是 `CashierQueue` 和 `LunchQueue` 的基类。 但是，当将这两个类组合成 `LunchCashierQueue` 时，会出现以下问题：新类包含类型 `Queue` 的两个子对象，一个来自 `CashierQueue`，另一个来自 `LunchQueue`。 下图显示了概念上的内存布局（实际物理内存布局可能会进行优化）：

模拟的午餐排队对象

`Queue` 对象中有两个 `LunchCashierQueue` 子对象。 以下代码将 `Queue` 声明为虚拟基类：

C++

```
// deriv_VirtualBaseClasses.cpp
// compile with: /LD
class Queue {};
class CashierQueue : virtual public Queue {};
class LunchQueue : virtual public Queue {};
class LunchCashierQueue : public LunchQueue, public CashierQueue {};
```

`virtual` 关键字可确保仅包含子对象 `Queue` 的一个副本（请参阅下图）。

具有虚拟基类的模拟的午餐排队对象

一个类可以同时具有一个给定类型的虚拟组件和非虚拟组件。 下图演示了这种情况：

同一个类的虚拟组件与非虚拟组件

在图中，`CashierQueue` 和 `LunchQueue` 将 `Queue` 用作虚拟基类。 但是，`TakeoutQueue` 将 `Queue` 指定为基类而不是虚拟基类。 因此，`LunchTakeoutCashierQueue` 具有类型 `Queue` 的两个子对象：一个来自包含 `LunchCashierQueue` 的继承路径，另一个来自包含 `TakeoutQueue` 的路径。 下图对此进行了演示。

带虚拟和非虚拟继承的对象布局

> **注意**
>
> 与非虚拟继承相比较，虚拟继承提供了显著的大小优势。 但是，它可能会引入额外的处理开销。

如果派生类重写它从虚拟基类继承的虚函数，并且派生基类的构造函数或析构函数使用指向虚拟基类的指针调用该虚函数，则编译器可能会将其他隐藏的“vtordisp”字段引入到具有虚拟基的类中。 `/vd0` 编译器选项禁止添加隐藏的 vtordisp 构造函数/析构函数置换成员。 `/vd1` 编译器选项（默认）可在必要时启用它们。 仅当确定所有类构造函数和析构函数以虚拟方式调用虚函数时才关闭 vtordisps。

`/vd` 编译器选项会影响整个编译模块。 使用 `vtordisp` pragma 可以逐个类地禁用 `vtordisp` 字段，然后重新启用它们：

C++

```
#pragma vtordisp( off )
class GetReal : virtual public { ... };
#pragma vtordisp( on )
```

名称多义性

多重继承使得沿多个路径继承名称成为可能。 沿这些路径的类成员名称不一定是唯一的。 这些名称冲突称为“多义性”。

任何引用类成员的表达式必须采用明确的引用。 以下示例说明如何产生多义性：

C++

```
// 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();  // 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 {};
```

对于前面的类声明，如下所示的代码是不明确的，因为 `b` 所指的 `b` 是在 `A` 中还是在 `B` 中并不清楚：

C++

```
C *pc = new C;

pc->b();
```

请看前面的示例。 由于名称 `a` 是类 `A` 和类 `B` 的成员，因此编译器无法辨明哪个 `a` 指定将调用函数。 如果成员可以引用多个函数、对象、类型或枚举数，则对该成员的访问是不明确的。

编译器通过按此顺序执行测试来检测多义性：

1. 如果对名称的访问是不明确的（如上所述），则会生成错误消息。
2. 如果重载函数是明确的，则将解析它们。
3. 如果对名称的访问违背了成员访问权限，则会生成错误消息。 有关详细信息，请参阅 Member-Access 控件。

在表达式通过继承产生多义性时，您可以通过限定考虑中的名称及其类名来手动消除该多义性。 若要适当编译上面的示例而不产生多义性，请使用如下代码：

C++

```
C *pc = new C;

pc->B::a();
```

> ⓘ **注意**
>
> 在声明 `C` 时，如果在 `B` 的范围内引用 `C`，则可能会导致出现错误。 但不会发出任何错误，直到在 `B` 的范围内实际创建对 `C` 的非限定引用。

主导

可以通过继承图访问多个名称（函数、对象或枚举器）。 这种情况被视为与非虚拟基类一起使用时目的不明确。 这些名称与虚拟基类一起使用时目的不明确，除非其中一个名称“决定”其他名称。

如果某个名称在两个类中定义并且一个类派生自另一个类，则该名称可控制另一个名称。 基准名称是派生类中的名称；此名称在本应出现多义性时使用，如以下示例所示：

C++

```
// 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.
};
```

不明确的转换

从指向类类型的指针或对类类型的引用的显式或隐式转换可能会导致多义性。 下图（指向基类的指针的不明确转换）显示如下内容：

- `D` 类型的对象的声明。
- 将地址运算符 （`&`） 应用于该对象的效果。 address-of 运算符总是提供该对象的基址。
- 将使用 address-of 运算符获取的指针显式转换为基类类型 `A` 的效果。 将该对象的地址强制转换为 `A*` 类型并不总是为编译器提供足够的信息，以供 `A` 类型的子对象进行选择；在这种情况下，将存在两个子对象。

指针到基类的不明确转换

到类型 `A*` （指向 `A` 的指针）的转换是不明确的，因为无法辩明 `A` 类型的哪个子对象是正确的。 可以通过显式指定要使用的子对象来避免多义性，如下所示：

```
C++
(A *)(B *)&d        // Use B subobject.
(A *)(C *)&d        // Use C subobject.
```

多义性和虚拟基类

如果使用虚拟基类，则函数、对象、类型和枚举数可通过多重继承路径到达。 因为仅有一个基类实例，因此在访问这些名称时不存在二义性。

下图显示如何使用虚拟和非虚拟继承构成对象。

虚拟和非虚拟派生

在该图中，通过非虚拟基类访问类 `A` 的任何成员都将导致二义性；编译器没有解释是使用与 `B` 关联的子对象还是与 `C` 关联的子对象的信息。 但是，将 `A` 指定为虚拟基类时，访问哪一个子对象都不成问题。

另请参阅

继承

Last updated on 2025/11/07

显式重写 (C++)

Microsoft 专用

如果在两个或更多个接口中声明了同一虚函数，并且某个类派生自这些接口，则可以显式重写每个虚函数。

有关使用 C++/CLI 在托管代码中进行显式重写的信息，请参阅显式重写。

结束 Microsoft 专用

示例

以下代码示例演示了如何使用显式重载：

C++

```
// 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;
}
```

Output

```
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)
```

另请参阅

继承

Last updated on 2025/11/07

抽象类 (C++)

抽象类作为可从中派生更具体的类的一般概念的表达。 无法创建抽象类类型的对象。 但可以使用指向抽象类类型的指针和引用。

可以通过声明至少一个纯虚拟成员函数来创建抽象类。 这是使用 pure 说明符 () *语法声明的虚函数* `= 0`。 派生自抽象类的类必须实现纯虚函数或者它们必须也是抽象类。

请考虑虚函数中所述的示例。 类 `Account` 的用途是提供通用功能，但 `Account` 类型的对象太通用，因此没什么用。 这表示 `Account` 是抽象类的很合适的候选项：

C++

```
// 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;
};
```

此声明与上一个声明的唯一区别是， `PrintBalance` 是用 pure 说明符 (`= 0`) 声明的。

抽象类限制

抽象类不能用于：

- 变量或成员数据
- 自变量类型
- 函数返回类型
- 显式转换的类型

如果抽象类的构造函数调用一个纯虚函数，无论是以直接还是间接方式，结果都是不确定的。 但是，抽象类的构造函数和析构函数都可以调用其他成员函数。

定义的纯虚函数

抽象类中的纯虚函数可以定义或具有实现。 只能使用完全限定的语法调用此类函数：

abstract-class-name::*function-name*()

在设计基类包含纯虚析构函数的类层次结构时，定义的纯虚函数非常有用。 这是因为对象销毁期间会始终调用基类析构函数。 请考虑以下示例：

C++

```
// deriv_RestrictionsOnUsingAbstractClasses.cpp
// Declare an abstract base class with a pure virtual destructor.
// It's the simplest possible abstract class.
class base
{
public:
    base() {}
    // To define the virtual destructor outside the class:
    virtual ~base() = 0;
    // Microsoft-specific extension to define it inline:
//  virtual ~base() = 0 {};
};

base::~base() {} // required if not using Microsoft extension

class derived : public base
{
public:
    derived() {}
    ~derived() {}
};

int main()
{
    derived aDerived; // destructor called when it goes out of scope
}
```

该示例演示了 Microsoft 编译器扩展如何支持向纯虚 `~base()` 添加内联定义。 还可以使用 `base::~base() {}`.在类外定义它。

当对象 `aDerived` 超出范围时，将调用类 `derived` 的析构函数。 编译器生成代码以在 `base` 析构函数之后隐式调用类 `derived` 的析构函数。 纯虚函数 `~base` 的空实现确保至少函数的某个实现存在。 如果没有，链接器将为隐式调用生成未解析的外部符号错误。

> **注意**
>
> 在前面的示例中，纯虚函数 `base::~base` 是从 `derived::~derived` 隐式调用的。 还可使用完全限定的成员函数名称显式调用纯虚函数。 此类函数必须具有实现，否则调用会在链接时导致错误。

另请参阅

继承

Last updated on 2025/11/07

范围规则摘要

名称的使用在其范围内必须是明确的（直至确定重载的点）。 如果名称表示一个函数，则该函数的参数的数目和类型必须明确。 如果名称保持明确，则应用 member-access 规则。

构造函数初始值设定项

构造函数初始值设定项在为其指定它们的构造函数的最外层块的作用域中计算。 因此，它们可使用构造函数的参数名。

全局名称

在以下情况下，对象、函数或枚举器的名称为全局名称：在任何函数或类的外部引用该名称或将全局一元范围运算符 (`::`) 作为其前缀，以及未将该名称与上述任何二元运算符结合使用：

- 范围解析 (`::`)
- 对象和引用的成员选择 (.)
- 指针的成员选择 (->)

限定名称

与二进制范围解析运算符 (`::`) 一起使用的名称称为“限定名称”。在二进制范围解析运算符之后指定的名称必须是该运算符左侧指定的类的成员或其基类的成员。

在成员选择运算符（. 或 -）后指定的名称必须是在该运算符左侧指定的对象的类类型成员或其基类成员>。 在成员选择运算符 (->) 右侧指定的名称也可以是其他类类型的对象，前提是 - 左侧是一个类对象并且该类定义了计算结果为指向某个其他类类型的指针的重载成员选择运算符 (->)>。 （类成员访问中更详细地讨论了此规定。）

编译器将按以下顺序搜索名称，并在找到名称时停止搜索：

1. 如果名称在函数内部使用，则搜索当前块范围；否则搜索全局范围。
2. 由里向外的每个封闭块范围，包括最外面的函数范围（包括函数参数）。
3. 如果名称在成员函数内使用，则在类的范围中搜索名称。
4. 在类的基类中搜索名称。
5. 搜索封闭的嵌套类范围（如果有）及其基项。 搜索继续，直到搜索最外面的封闭类范围。

6. 在全局范围中搜索。

但是，您可对此搜索顺序做如下修改：

1. 在名称前面放置 `::` 可强制从全局范围处开始搜索。
2. 名称前面带有 `class`、`struct` 和 `union` 关键字，可强制编译器仅搜索 `class`、`struct` 或 `union` 名称。
3. 范围解析运算符 (`::`) 左侧的名称只能是 `class`、`struct`、`namespace` 或 `union` 名称。

如果名称引用非静态成员，但用于静态成员函数，则将生成错误消息。 同样，如果名称引用封闭类中的任何非静态成员，则将生成错误消息，因为封闭类中没有此封闭类 `this` 指针。

函数参数名

函数定义中的函数参数名被视为位于函数的最外层块的范围内。 因此，它们是本地名称并且将在函数退出时超出范围。

函数声明（原型）中的函数参数名位于声明的局部范围内，并且将在声明结尾处超出范围。

默认参数位于它们作为默认值的参数的范围内，如前面两段中所述。 但是，它们无法访问局部变量或非静态类成员。 默认参数的计算时间是函数调用时，但计算位置是在函数声明的原始范围内。 因此，成员函数的默认参数始终在类范围中计算。

另请参阅

继承

Last updated on 2025/11/07

继承关键字

Microsoft 专用

> `class` *`class-name`*
>
> `class __single_inheritance` *`class-name`*
>
> `class __multiple_inheritance` *`class-name`*
>
> `class __virtual_inheritance` *`class-name`*

地点：

`class-name`
要声明的类的名称。

C++允许在类定义之前声明指向类成员的指针。 例如：

C++

```
class S;
int S::*p;
```

在上面的代码中， `p` 声明为指向类 S 的整数成员的指针。但是， `class S` 尚未在此代码中定义；它只是声明的。 当编译器遇到此类指针时，它必须使指针的通用表示形式。 表示形式的大小取决于指定的继承模型。 可通过三种方法向编译器指定继承模型：

- 在命令行中使用 /vmg 开关
- pointers_to_members使用杂注
- 使用继承关键字`__single_inheritance`和`__multiple_inheritance` `__virtual_inheritance`。 此方法基于每个类控制继承模型。

> **注意**
>
> 如果在定义类后始终声明指向类成员的指针，则无需使用这些选项。

如果在定义类之前声明指向类成员的指针，则它可能会对生成的可执行文件的大小和速度产生负面影响。 类使用的继承越复杂，表示指向类成员的指针所需的字节数越大。 而且，解释指针所需的代码越大。 单个（或否）继承最复杂，虚拟继承最复杂。 在定义类之前声明的成员的指针始终使用最大、最复杂的表示形式。

如果上面的示例更改为：

```C++
class __single_inheritance S;
int S::*p;
```

然后，无论指定的命令行选项或杂注，指向成员的 `class S` 指针都将使用尽可能小的表示形式。

> **注意**
>
> 类指针到成员表示形式的同一向前声明应在声明指向该类成员的指针的每个转换单元中发生，声明应在声明指向成员的指针之前发生。

为了与以前的版本、`_single_inheritance`和`_virtual_inheritance` `_multiple_inheritance`同义词兼容`__single_inheritance` `__multiple_inheritance`， `__virtual_inheritance`除非指定编译器选项/Za（禁用语言扩展）。

结束 Microsoft 专用

另请参阅

关键字

Last updated on 2025/11/07

virtual （C++）

`virtual` 关键字声明一个虚拟函数或一个虚拟基类。

语法

```
virtual [type-specifiers] member-function-declarator
virtual [access-specifier] base-class-name
```

参数

type-specifiers
指定此虚拟成员函数的返回类型。

member-function-declarator
声明一个成员函数。

access-specifier
定义对基类的访问级别（`public`、`protected` 或 `private`）。 可以在 `virtual` 关键字之前或之后显示。

base-class-name
标识一个以前声明的类类型。

备注

有关详细信息，请参阅虚拟函数。

另请参阅以下关键字：class、private、public 和 protected。

另请参阅

关键字

Last updated on 2025/11/07

`__super`

Microsoft 专用

允许您显式说明要为正在重写的函数调用基类实现。

语法

```
__super::member_function();
```

备注

在重载决策阶段将考虑所有可访问的基类方法，可提供最佳匹配项的函数就是调用的函数。

`__super` 只能在成员函数体内显示。

`__super` 不能与声明一起使用。 有关详细信息，请参阅使用声明。

在引入可注入代码的特性后，你的代码可能包含一个或多个基类，你不知道该基类的名称，但其包含你希望调用的方法。

示例

C++

```
// 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)
   }
};
```

结束 Microsoft 专用

另请参阅

关键字

Last updated on 2025/11/07

__interface

Microsoft 专用

可定义 Microsoft C++ 接口，如下所示：

- 可从零个或多个基接口继承。
- 不能从基类继承。
- 只能包含公共的纯虚方法。
- 不能包含构造函数、析构函数或运算符。
- 不能包含静态方法。
- 不能包含数据成员；允许使用属性。

语法

```
modifier __interface interface-name {interface-definition};
```

备注

C++ 类或结构可通过这些规则实现，但 `__interface` 会强制实施它们。

例如，以下是示例接口定义：

```C++
__interface IMyInterface {
   HRESULT CommitX();
   HRESULT get_X(BSTR* pbstrName);
};
```

有关托管接口的信息，请参阅接口类。

请注意，您无需显式指示 `CommitX` 和 `get_X` 函数是纯虚函数。 第一个函数的等效声明为：

```C++
virtual HRESULT CommitX() = 0;
```

`__interface` 表示 novtable `__declspec` 修饰符。

示例

以下示例演示如何使用接口中声明的属性。

C++

```
// 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);
}
```

Output

```
p->int_data = 100
bstr_data = Testing
```

结束 Microsoft 专用

另请参阅

关键字
接口特性

Last updated on 2025/11/07

特殊成员函数

特殊成员函数是类（或结构）成员函数，在某些情况下，编译器会自动为你生成。 包括默认构造函数、析构函数、复制构造函数和复制赋值运算符，以及移动构造函数和移动赋值运算符。 如果类未定义一个或多个特殊成员函数，则编译器可以隐式声明和定义所使用的函数。 编译器生成的实现称为默认特殊成员函数。 编译器不会生成不需要的函数。

可以使用 = default 关键字显式声明默认的特殊成员函数。 这使得编译器仅在需要时才定义函数，就像根本没有声明函数一样。

在某些情况下，编译器可能会生成已删除的特殊成员函数，这些函数未定义，因此不可调用。 如果考虑到类的其他属性，对类的特定特殊成员函数的调用没有意义，则可能会发生这种情况。 若要显式防止自动生成特殊成员函数，可以使用 = delete 关键字将其声明为已删除。

编译器会生成一个默认构造函数，只有当你没有声明任何其他构造函数时，该构造函数才不采用任何参数。 如果只声明了一个采用参数的构造函数，则尝试调用默认构造函数的代码会导致编译器生成错误消息。 编译器生成的默认构造函数对对象执行简单的逐个成员默认初始化。 默认初始化使所有成员变量处于不确定状态。

默认析构函数对对象执行逐个成员的析构。 仅当基类析构函数为虚拟函数时，它才是虚拟的。

默认的复制和移动构造和赋值操作执行非静态数据成员的逐个成员位模式复制或移动。 仅当未声明析构函数或移动或复制操作时，才会生成移动操作。 仅当未声明任何复制构造函数时，才会生成默认复制构造函数。 如果声明了移动操作，它将被隐式删除。 仅当未显式声明复制赋值运算符时才会生成默认的复制赋值运算符。 如果声明了移动操作，它将被隐式删除。

另请参阅

C++ 语言参考

Last updated on 2025/11/07

静态成员 (C++)

类可以包含静态成员数据和成员函数。 当数据成员被声明为“`static`”时，只会为类的所有对象保留一个数据副本。

静态数据成员不是给定的类类型的对象的一部分。 因此，静态数据成员的声明不被视为一个定义。 在类范围中声明数据成员，但在文件范围内执行定义。 这些静态类成员具有外部链接。 以下示例对此进行了说明：

C++

```
// 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()
{
}
```

在前面的代码中，该成员 `bytecount` 在类 `BufferedOutput` 中声明，但它必须在类声明的外部定义。

在不引用类类型的对象的情况下，可以引用静态数据成员。 可以获取使用 `BufferedOutput` 对象编写的字节数，如下所示：

C++

```
long nBytes = BufferedOutput::bytecount;
```

对于存在的静态成员，类类型的所有对象的存在则没有必要。 还可以使用成员选择（. 和 ->）运算符访问静态成员。 例如：

```
C++
BufferedOutput Console;

long nBytes = Console.bytecount;
```

在前面的示例中，不会评估对对象(`Console`)的引用；返回的值是静态对象 `bytecount` 的值。

静态数据成员遵循类成员访问规则，因此只允许类成员函数和友元拥有对静态数据成员的私有访问权限。 这些规则在成员访问控制中介绍。 例外情况是，无论静态数据成员的访问限制如何，都必须在文件范围内进行定义。 如果进行显式初始化数据成员，则必须使用定义提供初始值设定项。

静态成员的类型不是由其类名称限定的。 因此，`BufferedOutput::bytecount` 的类型为 **`long`**。

另请参阅

类和结构

Last updated on 2025/11/07

用户定义的类型转换 (C++)

*转换*会从其他类型的值生成某个类型的新值。 *标准转换*内置于 C++ 语言并支持其内置类型，可以创建*用户定义的转换*，以转换到用户定义的类型、从这些类型转换或者在这些类型之间执行转换。

标准转换执行内置类型之间的转换、通过继承相关联的类型的指针或引用之间的转换、与 void 指针的双向转换以及到 null 指针的转换。 有关详细信息，请参阅标准转换。 用户定义的转换执行用户定义的类型之间的转换，或者执行用户定义的类型和内置类型之间的转换。 可以将它们实现为转换构造函数或转换函数。

转换可以是显式（当程序员通过调用从一个类型转换为另一个类型时，例如强制转换或直接初始化的情况），也可以是隐式（当语言或程序调用其他类型而非程序员给定的类型时）。

在以下情况下尝试隐式转换：

- 提供给函数的自变量与匹配参数的类型不同。
- 从函数返回的值与函数返回类型的类型不同。
- 初始值表达式与其初始化的对象的类型不同。
- 用于控制条件语句、循环构造或切换的表达式不具有对其进行控制时所需的结果类型。
- 提供给运算符的操作数与匹配的操作数参数的类型不同。 对于内置运算符，这两个操作数的类型必须相同，并且要转换为可表示它们的常规类型。 有关详细信息，请参阅标准转换。 对于用户定义的运算符，每个操作数都必须与匹配的操作数参数的类型相同。

当一个标准转换无法完成隐式转换时，编译器可以使用用户定义的转换（可选择随后使用其他标准转换）来完成此操作。

当转换站点提供两个或多个用户定义的用于执行相同转换的转换时，该转换将被视为不明确。 这种不明确性是一个错误，因为编译器无法确定应选择哪一个可用转换。 但若只是定义执行相同转换的多种方式，则它不是一个错误，因为可用的转换集在源代码中的不同位置可能不同，例如，可取决于源文件中所包含的头文件。 只要转换站点只提供一个转换，就不会出现不明确性。 多种方式都会导致出现不明确转换，但最常见的方式如下：

- 多重继承。 该转换在多个基类中定义。
- 不明确的函数调用。 该转换定义为目标类型的转换构造函数以及源类型的转换函数。 有关详细信息，请参阅转换函数。

通常只需通过更全面地限定涉及类型的名称或执行显式强制转换来阐明你的意图，即可解决不明确性。

转换构造函数和转换函数都遵循成员访问控制规则，但仅当可以确定明确的转换时才考虑这些转换的可访问性。 这意味着，即使竞争的转换的访问级别会阻止使用该转换，该转换也可能具有不明确性。 有关成员可访问性的详细信息，请参阅成员访问控制。

explicit 关键字和有关隐式转换的问题

默认情况下，当你创建用户定义的转换时，编译器可使用它来执行隐式转换。 有时这是你需要进行的操作，但另一些时候用于指导编译器进行隐式转换的简单规则会使其接受你不希望接受的代码。

会导致问题的隐式转换的一个已知示例为：向 `bool` 的转换。 你可能出于很多原因要创建可用于布尔上下文的类的类型（例如，为了使其可用于控制 `if` 语句或循环），但在编译器执行向内置类型的用户定义的转换时，该编译器在之后可以应用其他标准转换。 此附加标准转换的目的是允许执行类似于从 `short` 提升到 `int` 的操作，但它也允许进行不太明显的转换（例如，从 `bool` 到 `int` 的转换），这使类的类型可用于你预期之外的整数上下文。 此特定问题称为*安全 Bool 问题*。 `explicit` 关键字有助于解决这种问题。

`explicit` 关键字会通知编译器指定的转换不能用于执行隐式转换。 如果要在引入 `explicit` 关键字之前享受隐式转换在语法方面带来的便利，则必须接受隐式转换有时候造成的意外后果，或使用较不为方便的已命名转换函数作为解决方法。 现在使用 `explicit` 关键字，你可以创建简便的转换，它们只能用于执行显式强制转换或直接初始化，并且不会导致“安全 Bool 问题”中举例说明的问题类型。

`explicit` 关键字可应用于自 C++98 后的转换构造函数和自 C++11 后的转换函数。 以下部分包含有关如何使用 `explicit` 关键字的详细信息。

转换构造函数

转换构造函数定义了从用户定义的类型或内置类型到用户定义的类型的转换。 以下示例演示了从内置类型 `double` 转换为用户定义的类型 `Money` 的转换构造函数。

C++

```
#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;
}
```

请注意，对函数 `display_balance` 的首次调用（它将采用类型 `Money` 的自变量）不需要转换，因为它的自变量类型正确。 但是，在第二次调用 `display_balance` 时需要转换，因为参数类型（带有值 **`double`** 的 `49.95`）并非该函数所期望的参数类型。 该函数不能直接使用此值，但是由于存在从参数类型（**`double`**）到匹配的参数类型（`Money`）的转换，因此类型 `Money` 的临时值将通过该参数进行构造并将用于完成该函数调用。 在对 `display_balance` 的第三次调用中，请注意，参数不是 **`double`** 而是带有值 **`float`** 的 `9.99`，但仍可以完成此次函数调用，因为编译器可以执行标准转换（在此案例中，执行从 **`float`** 到 **`double`** 的标准转换），然后执行从 **`double`** 到 `Money` 的用户定义的转换，以完成必要的转换。

声明转换构造函数

以下规则适用于声明转换构造函数：

- 转换的目标类型是要构造的用户定义的类型。
- 转换构造函数通常仅采用一个参数，它为源类型。 但是，当每个附加参数都具有默认值时，转换构造函数可指定这些附加参数。 源类型保留了第一个参数的类型。
- 与所有构造函数一样，转换构造函数不指定返回类型。 在声明中指定返回类型是一个错误。
- 转换构造函数可以为显式。

显式转换构造函数

通过将转换构造函数声明为 **`explicit`**，它只能用于执行对象的直接初始化或执行显式强制转换。 这将阻止接受类类型的自变量的函数同样隐式接受转换构造函数的源类型的自变量，并且

将阻止从该源类型的值复制初始化类的类型。 以下示例演示了如何定义显式转换构造函数，以及它对哪个代码格式正确所产生的影响。

C++

```
#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;
}
```

在此示例中，请注意，你可以继续使用显示转换构造函数来执行 `payable` 的直接初始化。 如果你要改为复制初始化 `Money payable = 79.99;`，这将是一个错误操作。 对 `display_balance` 的首次调用不受影响，因为自变量的类型正确。 对 `display_balance` 的第二次调用是一个错误，因为转换构造函数无法用于执行隐式转换。 由于到 `display_balance` 的显式强制转换，因此对 `Money` 的第三次调用合法，但请注意，编译器仍然已通过插入从 **`float`** 到 **`double`** 的隐式强制转换来帮助完成该强制转换。

尽管允许隐式转换带来的便利很具吸引力，但执行此操作会引入难以发现的 Bug。 根据经验，应该让所有转换构造函数变为显式，除非你确定希望隐式执行特定转换。

转换函数

转换函数定义了从用户定义的类型到其他类型的转换。 这些函数有时称为“强制转换运算符”，因为在值强制转换为其他类型时将随转换构造函数一起调用它们。 以下示例演示了从用户定义

的类型 `Money` 转换为内置类型 **`double`** 的转换函数：

```C++
#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;
}
```

请注意，成员变量 `amount` 设置为 private，并且到类型 **`double`** 的公共转换函数的引入目的只是为了返回 `amount` 的值。 在函数 `display_balance` 中，当 `balance` 的值通过使用流插入运算符 `<<` 流入标准输出时，将执行隐式转换。 由于没有为用户定义的类型 `Money` 定义任何流插入运算符，但存在一个适用于内置类型 **`double`** 的流插入运算符，因此编译器可以使用从 `Money` 到 **`double`** 的转换函数来满足此流插入运算符。

转换函数通过派生类继承。 派生类中的转换函数仅在其转换为完全相同的类型时覆盖继承的转换函数。 例如，派生类的用户定义的转换函数“运算符 int”不会重写（甚至不会影响）基类的用户定义的转换函数“运算符 short”，即使标准转换定义了 **`int`** 和 **`short`** 之间的转换关系也是如此。

声明转换函数

以下规则适用于声明转换函数：

- 必须在声明转换函数之前先声明转换的目标类型。 无法在转换函数的声明中声明类、结构、枚举和 typedef。

  ```C++
  operator struct String { char string_storage; }() // illegal
  ```

- 转换函数不采用自变量。 在声明中指定任何参数是一个错误操作。

- 转换函数具有由转换函数名称（它也是转换的目标类型的名称）指定的返回类型。 在声明中指定返回类型是一个错误。
- 转换函数可以是虚函数。
- 转换函数可以是显式函数。

显式转换函数

当转换函数声明为显式函数时，它只能用于执行显式强制转换。 这还将阻止接受转换函数目标类型的自变量的函数同样隐式接受类类型的自变量，并且将阻止从该类类型的值复制初始化目标类型的实例。 以下示例演示了如何定义显式转换函数，以及它对哪个代码格式正确所产生的影响。

C++

```
#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;
}
```

此处，转换函数“运算符 double”已声明为显式，并且已在函数 `double` 中引入对类型 `display_balance` 的显式强制转换以执行转换。 若已省略此强制转换，则编译器将无法定位适用于类型 `<<` 的相应流插入运算符 `Money`，并且将发生错误。

Last updated on 2025/11/07

可变数据成员 (C++)

此关键字只能应用于类的非静态、非常量和非参考数据成员。 如果某个数据成员被声明为 `mutable`，则从 `const` 成员函数为此数据成员赋值是合法的。

语法

```
mutable member-variable-declaration;
```

备注

例如，以下代码在编译时不会出错，因为 `m_accessCount` 已声明为 `mutable`，因此可以由 `GetFlag` 修改，即使 `GetFlag` 是常量成员函数也是如此。

C++

```cpp
// mutable.cpp
class X
{
public:
    bool GetFlag() const
    {
        m_accessCount++;
        return m_flag;
    }
private:
    bool m_flag;
    mutable int m_accessCount;
};
```

另请参阅

关键字

Last updated on 2025/11/07

嵌套类声明

可以在一个类的范围内声明另一个类。 这种类称为“嵌套类”。嵌套类被认为在封闭类的范围内且可在该范围内使用。 若要从嵌套类的即时封闭范围之外的某个范围引用该类，则必须使用完全限定名。

下面的示例演示如何声明嵌套类：

C++

```
// 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` 和 `BufferedIO::BufferedOutput` 在 `BufferedIO` 内声明。 这些类名称在类 `BufferedIO` 的范围外不可见。 但是， `BufferedIO` 类型的对象不包含 `BufferedInput` 或 `BufferedOutput` 类型的任何对象。

嵌套类只能从封闭类中直接使用名称、类型名称，静态成员的名称和枚举数。 若要使用其他类成员的名称，您必须使用指针、引用或对象名。

在前面的 `BufferedIO` 示例中，枚举 `IOError` 可由嵌套类中的成员函数、`BufferedIO::BufferedInput` 或 `BufferedIO::BufferedOutput` 直接访问，如函数 `good` 中所示。

> ⓘ **注意**
>
> 嵌套类仅在类范围内声明类型。 它们不会导致创建嵌套类的包含对象。 前面的示例声明两个嵌套类，但未声明这些类类型的任何对象。

在将类型名称与前向声明一起声明时，会引发嵌套类声明的范围可见性的异常。 在这种情况下，由前向声明声明的类名在封闭类的外部可见，其范围定义为最小的封闭非类范围。 例如：

C++

```
// 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;
}
```

嵌套类中的访问权限

将一个类嵌入另一个类中不会为嵌入类的成员函数提供特殊访问权限。 同样，封闭类的成员函数不具有对嵌套类的成员的特殊访问权限。

嵌套类中的成员函数

在嵌套类中声明的成员函数可在文件范围中定义。 前面的示例可能已编写：

C++

```
// 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()
{
}
```

在前面的示例中，qualified-type-name 语法用于声明函数名称。 声明：

C++

```
BufferedIO::BufferedInput::read()
```

表示“作为 `read` 类范围内的 `BufferedInput` 类的成员的 `BufferedIO` 函数”。由于此声明使用 qualified-type-name 语法，因此以下形式的构造是可能的：

C++

```
typedef BufferedIO::BufferedInput BIO_INPUT;

int BIO_INPUT::read()
```

上述声明与前一个声明等效，但它使用了 **`typedef`** 名称来代替类名称。

嵌套类中的友元函数

嵌套类中声明的友元函数被认为是在嵌套类而不是封闭类的范围内。 因此，友元函数未获得对封闭类的成员或成员函数的特定访问权限。 如果需要使用在友元函数中的嵌套类中声明的名称，并且友元函数是在文件范围内定义的，请使用限定的类型名称，如下所示：

C++

```
// 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()
{
}
```

以下代码演示声明为友元函数的函数 `GetExtendedErrorStatus`。 在文件范围内定义的函数中，将消息从静态数组复制到类成员中。 请注意，`GetExtendedErrorStatus` 的更佳实现是将其声明为：

```
C++
int GetExtendedErrorStatus( char *message )
```

利用前面的接口，许多类可以通过传递要复制错误消息的内存位置来使用此函数的服务。

另请参阅

类和结构

Last updated on 2025/11/07

匿名类类型

类可以是匿名的 - 也就是说，可以在没有 identifier 的情况下声明类。 在将类名称替换为 `typedef` 名称时，这会很有用，如下所示：

C++

```
typedef struct
{
    unsigned x;
    unsigned y;
} POINT;
```

> **注意**
>
> 上面示例中显示的匿名类的用法对于保留与现有 C 代码的兼容性很有用。 在某些 C 代码中，将 `typedef` 与匿名结构结合使用是很普遍的。

如果您希望对类成员的引用就像它未包含在独立类中的情况一样出现，则匿名类也很有用，如下所示：

C++

```
struct PTValue
{
    POINT ptLoc;
    union
    {
        int  iValue;
        long lValue;
    };
};

PTValue ptv;
```

在上面的代码中，可以使用对象成员选定内容运算符 (.) 访问 `iValue`，如下所示：

C++

```
int i = ptv.iValue;
```

匿名类受某些限制的约束。 有关匿名联合的详细信息，请参阅 联合。 匿名类：

- 不能具有构造函数或析构函数。
- 不能作为函数的参数传递（除非使用省略号使类型检查无效）。

- 无法作为函数中的返回值返回。

匿名结构

Microsoft 专用

利用 Microsoft C 扩展，您可以在另一个结构中声明结构变量，而无需为其指定名称。 这些嵌套结构称为匿名结构。 C++ 不允许匿名结构。

您可以像访问包含结构中的成员一样访问匿名结构的成员。

C++

```
// 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
```

结束 Microsoft 专用

Last updated on 2025/11/07

指向成员的指针

指向成员的指针的声明是指针声明的特例。 使用以下序列来声明它们：

> storage-class-specifiers$_{opt}$ cv-qualifiers$_{opt}$ type-specifier ms-modifier$_{opt}$ qualified-name`::*` cv-qualifiers$_{opt}$ identifier pm-initializer$_{opt}$`;`

1. 声明说明符：
    - 可选存储类说明符。
    - 可选 `const` 说明符和 `volatile` 说明符。
    - 类型说明符：类型的名称。 这是要指向的成员的类型，而不是类。
2. 声明符：
    - 可选的 Microsoft 专用修饰符。 有关详细信息，请参阅 Microsoft 专用修饰符。
    - 包含要指向的成员的类的限定名。
    - `::` 运算符。
    - `*` 运算符。
    - 可选 `const` 说明符和 `volatile` 说明符。
    - 命名指向成员的指针的标识符。
3. 指向成员的可选指针初始值设定项：
    - `=` 运算符。
    - `&` 运算符。
    - 类的限定名。
    - `::` 运算符。
    - 适当类型的类的非静态成员的名称。

像往常一样，允许在单个声明中使用多个声明符（以及任何关联的初始值设定项）。 指向成员的指针不能指向类的静态成员、引用类型的成员或 `void`。

指向类的成员的指针与普通指针不同，因为它有该成员的类型的类型信息和该成员所属的类的类型信息。 常规指针只标识内存中的一个对象或只具有其地址。 指向类的某个成员的指针标识类

的所有实例中的该成员。 以下示例声明类、`Window` 和一些指向成员数据的指针。

C++

```
// 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()
{
}
```

在前面的示例中， `pwCaption` 是一个指针，它指向类型 `Window` 的类 **`char*`** 的任何成员。 `pwCaption` 的类型是 `char * Window::*`。 下一个代码片段将指针声明为 `SetCaption` 和 `GetCaption` 成员函数。

C++

```
const char * (Window::* pfnwGC)() = &Window::GetCaption;
bool (Window::* pfnwSC)( const char * ) = &Window::SetCaption;
```

指针 `pfnwGC` 和 `pfnwSC` 分别指向 `GetCaption` 类的 `SetCaption` 和 `Window`。 以下代码直接使用指向成员 `pwCaption` 的指针将信息复制到窗口标题：

C++

```
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';
```

`.*` 和 `->*` 运算符（指向成员运算符的指针）的区别在于 `.*` 运算符选择给定对象或对象引用的成员，而 `->*` 运算符通过指针选择成员。 有关这些运算符的更多信息，请参阅使用指向成员的指针运算符的表达式。

指向成员的指针运算符的结果是成员的类型。 在本例中，该值为 `char *`。

以下代码片段使用指向成员的指针调用成员函数 `GetCaption` 和 `SetCaption`：

C++

```
// 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 );
```

针对指向成员的指针的限制

静态成员的地址不是指向成员的指针。 它是指向静态成员的一个实例的常规指针。 对于给定类的所有对象，只存在一个静态成员的实例。 这意味着可以使用普通的 address-of (&) 和取消引用 (*) 运算符。

指向成员和虚函数的指针

通过指向成员函数的指针调用虚函数就如同直接调用函数一样。 在 v 表中查找正确的函数并调用。

一直以来，虚函数工作的关键是通过指向基类的指针来调用它们。 有关虚拟函数的详细信息，请参阅 虚拟函数。

以下代码演示如何通过指向成员函数的指针调用虚函数：

C++

```
// 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" << endl;
}

class Derived : public Base
{
public:
    void Print();  // Print is still a virtual function.
};

void Derived::Print()
{
    cout << "Print function for class Derived" << endl;
}

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
```

Last updated on 2025/11/07

`this` 指针

`this` 指针是只能在 `class`、`struct` 或 `union` 类型的非静态成员函数中访问的指针。 它指向为其调用成员函数的对象。 静态成员函数不具有 `this` 指针。

语法

```C++
this
this->member-identifier
```

备注

对象的 `this` 指针不是对象本身的一部分。 它不是对象上 `sizeof` 语句的结果的一部分。 当对某个对象调用非静态成员函数时，编译器会将该对象的地址作为隐藏的参数传递给函数。 例如，以下函数调用：

```C++
myDate.setMonth( 3 );
```

可以解释为：

```C++
setMonth( &myDate, 3 );
```

对象的地址可从成员函数的内部作为 `this` 指针提供。 `this` 指针的大多数使用都是隐式的。 在引用 `this` 的成员时使用显式 class 是合法的，但没有必要。 例如：

```C++
void Date::setMonth( int mn )
{
   month = mn;             // These three statements
   this->month = mn;       // are equivalent
   (*this).month = mn;
}
```

表达式 `*this` 通常用于从成员函数返回当前对象：

```C++
```

```
return *this;
```

`this` 指针还用于防止自引用：

C++

```
if (&Object != this) {
// do not execute in cases of self-reference
```

> ⓘ **注意**
>
> 由于 `this` 指针不可修改，因此不允许对 `this` 赋值。 C++ 的早期实现允许对 `this` 赋值。

有时，`this` 指针可直接使用，例如，当操作自引用数据结构，而其中需要当前对象的地址时。

示例

C++

```
// 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();
}
```

Output

```
my buffer
your buffer
```

`this` 指针的类型

`this` 指针的类型根据函数声明是否包括 `const` 和/或 `volatile` 关键字而更改。 以下语法描述了成员函数中 `this` 的类型：

.- .

成员函数的声明符会确定 *`cv-qualifier-list`*。 它可以是 `const` 或 `volatile` （或两者）。 *`class-type`* 是 class 的名称。

无法重新分配 `this` 指针。 成员函数声明中使用的 `const` 或 `volatile` 限定符适用于由该函数范围中的 class 指针指向的 `this` 实例，如下标所示：

展开表

成员函数声明	`this` 指针的类型为 class 命名为 `myClass`
`void Func()`	`myClass *`
`void Func() const`	`const myClass *`
`void Func() volatile`	`volatile myClass *`
`void Func() const volatile`	`const volatile myClass *`

下表介绍了有关 `const` 和 `volatile` 的详细信息。

`this` 修饰符的语义

展开表

修饰符	含义
`const`	无法更改成员数据；无法调用不是 `const` 的成员函数。
`volatile`	每当访问成员数据时，都会从内存中加载该数据；禁用某些优化。

将 `const` 对象传递给不是 `const` 的成员函数是错误的。

同样，将 `volatile` 对象传递给不是 `volatile` 的成员函数也是错误的。

声明为 `const` 的成员函数无法更改成员数据。 在 `const` 函数中， `this` 指针是指向 `const` 对象的指针。

> **注意**
>
> 构造函数和析构函数不能声明为 `const` 或 `volatile`。 但是，可以在 `const` 或 `volatile` 对象上调用它们。

另请参阅

关键字

Last updated on 2025/11/07

C++ 位域

类和结构可包含比整型类型占用更少存储空间的成员。 这些成员被指定为位域。 位域成员声明符规范的语法如下：

语法

declarator:*constant-expression*

注解

（可选）declarator 是在程序中访问成员的名称。 它必须是整型类型（包括枚举类型）。 constant-expression 指定结构中成员所占据的位数。 匿名位域—即不带标识符的位域成员—可用于填充。

> ⓘ **注意**
>
> 宽度为 0 的未命名位域强制将下一个位域与下一个类型边界对齐，其中类型是成员的类型。

下面的示例声明包含位域的结构：

C++

```
// 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)
};
```

`Date`类型的对象的概念上的内存布局如下图所示：

`nYear`的长度为 8 位，这会溢出声明类型`unsigned short`的字边界。 因此，它始于新`unsigned short`的开头。 并不必使所有位域均适合基础类型的对象；根据声明中请求的位数来分配新的存

储单元。

Microsoft 专用

声明为位域的数据从低位到高位进行排序，如上图所示。

结束 Microsoft 专用

如果结构的声明包含长度为 0 的未命名字段（如以下示例所示）：

C++

```
// 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)
};
```

则内存布局如下图中所示：

位域的基础类型必须是整数类型，如内置类型中所述。

如果对类型`const T&`的引用的初始值设定项是引用类型为`T`的位域的 lvalue，引用不会直接绑定到位域。 相反，引用会绑定到一个临时初始值设定项以保留位域的值。

位域的限制

以下列表详述了位域的错误操作：

- 采用位域的地址。
- 使用位域初始化非 `const` 引用。

另请参阅

类和结构

Last updated on 2025/11/07

C++ 中的 Lambda 表达式

项目 • 2023/04/03

在 C++ 11 和更高版本中，Lambda 表达式（通常称为 Lambda）是一种在被调用的位置或作为参数传递给函数的位置定义匿名函数对象（闭包）的简便方法。 Lambda 通常用于封装传递给算法或异步函数的少量代码行。 本文将提供 Lambda 的定义、将它与其他编程技术进行比较。 其中介绍了各自的优点，并提供了一些基本示例。

相关文章

- Lambda 表达式与函数对象
- 使用 Lambda 表达式
- constexpr Lambda 表达式

Lambda 表达式的各个部分

下面是作为第三个参数 `std::sort()` 传递给函数的简单 lambda：

C++

```
#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
    );
}
```

下图显示了 lambda 语法的各个部分：

1. capture 子句（在 C++ 规范中也称为 Lambda 引导。）
2. 参数列表（可选）。 （也称为 Lambda 声明符）
3. mutable 规范（可选）。
4. exception-specification（可选）。
5. trailing-return-type（可选）。
6. Lambda 体。

capture 子句

Lambda 可在其主体中引入新的变量（用 C++14），它还可以访问（或"捕获"）周边范围内的变量。 Lambda 以 capture 子句开头。 它指定捕获哪些变量，以及捕获是通过值还是通过引用进行的。 有与号 (`&`) 前缀的变量通过引用进行访问，没有该前缀的变量通过值进行访问。

空 capture 子句 `[ ]` 指示 lambda 表达式的主体不访问封闭范围中的变量。

可以使用默认捕获模式来指示如何捕获 Lambda 体中引用的任何外部变量：`[&]` 表示通过引用捕获引用的所有变量，而 `[=]` 表示通过值捕获它们。 可以使用默认捕获模式，然后为特定变量显式指定相反的模式。 例如，如果 lambda 体通过引用访问外部变量 `total` 并通过值访问外部变量 `factor`，则以下 capture 子句等效：

```
C++

[&total, factor]
[factor, &total]
[&, factor]
[=, &total]
```

使用默认捕获时，只有 Lambda 体中提及的变量才会被捕获。

如果 capture 子句包含默认捕获 `&`，则该 capture 子句的捕获中没有任何标识符可采用 `&identifier` 形式。 同样，如果 capture 子句包含默认捕获 `=`，则该 capture 子句没有任何捕获可采用 `=identifier` 形式。 标识符或 **`this`** 在 capture 子句中出现的次数不能超过一次。 以下代码片段给出了一些示例：

```
C++

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
}
```

捕获后跟省略号是一个包扩展，如以下可变参数模板示例中所示：

```
C++

template<class... Args>
void f(Args... args) {
    auto x = [args...] { return g(args...); };
    x();
}
```

要在类成员函数体中使用 Lambda 表达式，请将 `this` 指针传递给 capture 子句，以提供对封闭类的成员函数和数据成员的访问权限。

Visual Studio 2017 版本 15.3 及更高版本（在 /std:c++17 模式及更高版本中可用）： 可以通过在 capture 子句中指定 `*this` 通过值捕获 `this` 指针。 通过值捕获会将整个闭包复制到调用 Lambda 的每个调用站点。 （闭包是封装 Lambda 表达式的匿名函数对象）。当 Lambda 在并行或异步操作中执行时，通过值捕获非常有用。 它在某些硬件体系结构（如 NUMA）上特别有用。

有关展示如何将 Lambda 表达式与类成员函数一起使用的示例，请参阅 Lambda 表达式示例中的“示例：在方法中使用 Lambda 表达式”。

在使用 capture 子句时，建议你记住以下几点（尤其是使用采取多线程的 Lambda 时）：

- 引用捕获可用于修改外部变量，而值捕获却不能实现此操作。 （`mutable` 允许修改副本，而不能修改原始项。）
- 引用捕获会反映外部变量的更新，而值捕获不会。
- 引用捕获引入生存期依赖项，而值捕获却没有生存期依赖项。 当 Lambda 以异步方式运行时，这一点尤其重要。 如果在异步 Lambda 中通过引用捕获局部变量，该局部变量将很容易在 Lambda 运行时消失。 代码可能会导致在运行时发生访问冲突。

通用捕获 (C++14)

在 C++14 中，可以在 capture 子句中引入和初始化新变量，而无需将这些变量存在于 lambda 函数的封闭范围内。 初始化可以任何任意表达式表示；且将从该表达式生成的类

型推导新变量的类型。 借助此功能，你可以从周边范围捕获只移动的变量（例如 `std::unique_ptr`）并在 Lambda 中使用它们。

```
C++

pNums = make_unique<vector<int>>(nums);
//...
      auto a = [ptr = move(pNums)]()
        {
           // use ptr
        };
```

参数列表

Lambda 既可以捕获变量，也可以接受输入参数。 参数列表（在标准语法中称为 Lambda 声明符）是可选的，它在大多数方面类似于函数的参数列表。

```
C++

auto y = [] (int first, int second)
{
    return first + second;
};
```

在 C++14 中，如果参数类型是泛型，则可以使用 `auto` 关键字作为类型说明符。 此关键字将告知编译器将函数调用运算符创建为模板。 参数列表中的每个 `auto` 实例等效于一个不同的类型参数。

```
C++

auto y = [] (auto first, auto second)
{
    return first + second;
};
```

Lambda 表达式可以将另一个 Lambda 表达式作为其自变量。 有关详细信息，请参阅 Lambda 表达式示例一文中的“高阶 Lambda 表达式”。

由于参数列表是可选的，因此在不将自变量传递到 Lambda 表达式，并且其 Lambda 声明符不包含 exception-specification、trailing-return-type 或 `mutable` 的情况下，可以省略空括号。

mutable 规范

通常，Lambda 的函数调用运算符是 const-by-value，但对 `mutable` 关键字的使用可将其取消。它不产生 mutable 数据成员。 利用 `mutable` 规范，Lambda 表达式的主体可以修改通过值捕获的变量。 本文后面的一些示例将展示如何使用 `mutable`。

异常规范

你可以使用 `noexcept` 异常规范来指示 Lambda 表达式不会引发任何异常。 与普通函数一样，如果 Lambda 表达式声明 `noexcept` 异常规范且 Lambda 体引发异常，Microsoft C++ 编译器将生成警告 C4297，如下所示：

C++

```
// throw_lambda_expression.cpp
// compile with: /W4 /EHsc
int main() // C4297 expected
{
   []() noexcept { throw 5; }();
}
```

有关详细信息，请参阅异常规范 (throw)。

返回类型

将自动推导 Lambda 表达式的返回类型。 无需使用 auto 关键字，除非指定了 trailing-return-type。 trailing-return-type 类似于普通函数或成员函数的 return-type 部分。 但是，返回类型必须跟在参数列表的后面，你必须在返回类型前面包含 trailing-return-type 关键字 `->`。

如果 Lambda 体仅包含一个返回语句，则可以省略 Lambda 表达式的 return-type 部分。 或者，在表达式未返回值的情况下。 如果 lambda 体包含单个返回语句，编译器将从返回表达式的类型推导返回类型。 否则，编译器会将返回类型推导为 `void`。 下面的示例代码片段说明了这一原则：

C++

```
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 isn't
valid
```

lambda 表达式可以生成另一个 lambda 表达式作为其返回值。 有关详细信息，请参阅 Lambda 表达式示例中的“高阶 Lambda 表达式”。

Lambda 体

Lambda 表达式的 Lambda 体是一个复合语句。 它可以包含普通函数或成员函数体中允许的任何内容。 普通函数和 lambda 表达式的主体均可访问以下变量类型：

- 从封闭范围捕获变量，如前所述。
- 参数。
- 本地声明变量。
- 类数据成员（在类内部声明并且捕获 `this` 时）。
- 具有静态存储持续时间的任何变量（例如，全局变量）。

以下示例包含通过值显式捕获变量 `n` 并通过引用隐式捕获变量 `m` 的 lambda 表达式：

C++

```
// 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;
}
```

Output

```
5
0
```

由于变量 `n` 是通过值捕获的，因此在调用 lambda 表达式后，变量的值仍保持 `0` 不变。 `mutable` 规范允许在 Lambda 中修改 `n`。

Lambda 表达式只能捕获具有自动存储持续时间的变量。 但是，可以在 Lambda 表达式体中使用具有静态持续存储时间的变量。 以下示例使用 `generate` 函数和 lambda 表达式为 `vector` 对象中的每个元素赋值。 lambda 表达式将修改静态变量以生成下一个元素的值。

C++

```
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 isn't thread-safe and is shown for illustration only
}
```

有关详细信息，请参阅 generate。

下面的代码示例使用上一示例中的函数，并添加了使用 C++ 标准库算法 `generate_n` 的 Lambda 表达式的示例。 该 lambda 表达式将 `vector` 对象的元素指派给前两个元素之和。 使用了 **`mutable`** 关键字，使 Lambda 表达式主体可以修改 Lambda 表达式通过值捕获的外部变量 `x` 和 `y` 的副本。 由于 lambda 表达式通过值捕获原始变量 `x` 和 `y`，因此它们的值在 lambda 执行后仍为 `1`。

C++

```
// 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 isn't 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);
}
```

Output

```
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
```

有关详细信息，请参阅 generate_n。

`constexpr` Lambda 表达式

Visual Studio 2017 版本 15.3 及更高版本（在 /std:c++17 模式和更高版本中可用）：在常量表达式中允许初始化捕获或引入的每个数据成员时，可以将 Lambda 表达式声明为 `constexpr`（或在常量表达式中使用它）。

```C++
int y = 32;
auto answer = [y]() constexpr
{
    int x = 10;
    return y + x;
};

constexpr int Increment(int n)
{
    return [n] { return n + 1; }();
}
```

如果 Lambda 结果满足 `constexpr` 函数的要求，则 Lambda 是隐式的 `constexpr`：

```C++
auto answer = [](int n)
{
    return 32 + n;
};

constexpr int response = answer(10);
```

如果 Lambda 是隐式或显式的 `constexpr`，则转换为函数指针将生成 `constexpr` 函数：

```C++
auto Increment = [](int n)
{
    return n + 1;
};

constexpr int(*inc)(int) = Increment;
```

Microsoft 专用

以下公共语言运行时 (CLR) 托管实体中不支持 Lambda：`ref class`、`ref struct`、`value class` 或 `value struct`。

如果你使用 Microsoft 专用的修饰符（例如 __declspec），可以紧接在 `parameter-declaration-clause` 后将其插入 Lambda 表达式。 例如：

```
C++

auto Sqr = [](int t) __declspec(code_seg("PagedMem")) -> int { return t*t;
};
```

若要确定 Lambda 是否支持某个特定修饰符，请参阅 Microsoft 专用修饰符一文中有关该修饰符的部分。

Visual Studio 支持 C++11 标准 Lambda 功能和无状态 Lambda。 无状态 Lambda 可转换为使用任意调用约定的函数指针。

请参阅

C++ 语言参考
C++ 标准库中的函数对象
函数调用
for_each

Lambda 表达式语法

本文演示了 lambda 表达式的语法和结构化元素。 有关 Lambda 表达式的说明，请参阅 Lambda 表达式。

函数对象与 lambda

当你编写代码时，可能会使用函数指针和函数对象来解决问题和执行计算，尤其是当使用 C++ 标准库算法时。 函数指针和函数对象各有利弊。例如，函数指针具有最低的语法开销，但不保持范围内的状态，函数对象可保持状态，但需要类定义的语法开销。

lambda 结合了函数指针和函数对象的优点并避免其缺点。 与函数对象一样，lambda 是灵活的并且可以保持状态，但与函数对象不同之处在于其简洁的语法不需要显式类定义。 使用 lambda，你可以编写出比等效的函数对象代码更简洁、更不容易出错的代码。

以下示例将比较 lambda 的用途和函数对象的用途。 第一个示例使用 lambda 向控制台打印 `vector` 对象中的每个元素是偶数还是奇数。 第二个示例使用函数对象来完成相同任务。

示例 1：使用 lambda

此示例将一个 lambda 传递给 for_each 函数。 该 lambda 打印一个结果，该结果指出 `vector` 对象中的每个元素是偶数还是奇数。

Code

C++

```
// 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;
}
```

Output

```
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.
```

注释

在此示例中，for_each 函数的第三个参数是一个 lambda。 `[&evenCount]` 部分指定表达式的捕获子句， `(int n)` 指定参数列表，剩余部分指定表达式的主体。

示例 2：使用函数对象

有时 lambda 过于庞大，无法在上一示例的基础上大幅度扩展。 下一示例将函数对象（而非 lambda）用于 for_each 函数以产生与示例 1 相同的结果。 两个示例都在 `vector` 对象中存储偶数的个数。 为保持运算的状态， `FunctorClass` 类通过引用存储 `m_evenCount` 变量作为成员变量。 为了执行运算， `FunctorClass` 将实现函数调用运算符 operator()。 Microsoft C++ 编译器生成的代码与示例 1 中的 lambda 代码在大小和性能上相差无几。 对于类似本文中示例的基本问题，较为简单的 lambda 设计可能优于函数对象设计。 但是，如果你认为该功能在将来可能需要重大扩展，则使用函数对象设计，这样代码维护会更简单。

有关 operator() 的详细信息，请参阅**函数调用**。 有关 for_each 函数的详细信息，请参阅 **for_each**。

Code

C++

```
// 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;
}
```

Output

```
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.
```

另请参阅

Lambda 表达式
Lambda 表达式的示例
generate
generate_n
for_each
异常规范 (throw)
编译器警告（等级 1）C4297
Microsoft 专用的修饰符

Last updated on 2025/11/07

Lambda 表达式的示例

本文演示如何在你的程序中使用 lambda 表达式。 有关 lambda 表达式的概述，请参阅 Lambda 表达式。 有关 lambda 表达式结构的详细信息，请参阅 Lambda 表达式语法。

声明 Lambda 表达式

示例 1

由于 lambda 表达式已类型化，所以你可以将其指派给 `auto` 变量或 function 对象，如下所示：

C++

```
// 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;
}
```

此示例产生以下输出：

Output

```
5
7
```

备注

有关详细信息，请参阅 auto、function 类和函数调用。

虽然 lambda 表达式多在函数的主体中声明，但是可以在初始化变量的任何地方声明。

示例 2

Microsoft C++ 编译器将在声明而非调用 lambda 表达式时，将表达式绑定到捕获的变量。 以下示例显示一个通过值捕获局部变量 `i` 并通过引用捕获局部变量 `j` 的 lambda 表达式。 由于 lambda 表达式通过值捕获 `i`，因此在程序后面部分中重新指派 `i` 不影响该表达式的结果。 但是，由于 lambda 表达式通过引用捕获 `j`，因此重新指派 `j` 会影响该表达式的结果。

C++

```
// 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;
}
```

此示例产生以下输出：

Output

```
47
```

[本文内容]

调用 Lambda 表达式

你可以立即调用 Lambda 表达式，如下面的代码片段所示。 第二个代码片段演示如何将 lambda 作为自变量传递给标准库算法，例如 `find_if`。

示例 1

以下示例声明的 lambda 表达式将返回两个整数的总和并使用自变量 `5` 和 `4` 立即调用该表达式：

C++

```
// 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;
}
```

此示例产生以下输出：

Output

```
9
```

示例 2

以下示例将 Lambda 表达式作为自变量传递给 `find_if` 函数。 如果 lambda 表达式的参数是偶数，则返回 `true`。

C++

```
// 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;
    }
}
```

此示例产生以下输出：

Output

```
The first even number in the list is 42.
```

备注

有关 `find_if` 函数的详细信息，请参阅 find_if。 有关执行公共算法的 C++ 标准库函数的详细信息，请参阅 <algorithm>。

[本文内容]

嵌套 Lambda 表达式

示例

你可以将 lambda 表达式嵌套在另一个中，如下例所示。 内部 lambda 表达式将其自变量与 2 相乘并返回结果。 外部 lambda 表达式通过其自变量调用内部 lambda 表达式并在结果上加 3。

C++

```
// 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;
}
```

此示例产生以下输出：

Output

```
13
```

备注

在该示例中，`[](int y) { return y * 2; }` 是嵌套的 lambda 表达式。

[本文内容]

高阶 Lambda 函数

示例

许多编程语言支持“高阶函数”的概念。高阶函数是一个 lambda 表达式，它采用另一个 lambda 表达式作为其自变量，或返回 lambda 表达式。 你可以使用 function 类，使得 C++ lambda 表达式具有类似高阶函数的行为。 以下示例显示返回 `function` 对象的 lambda 表达式和采用 `function` 对象作为其参数的 lambda 表达式。

C++

```
// 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;
}
```

此示例产生以下输出：

Output

```
30
```

[本文内容]

在函数中使用 Lambda 表达式

示例

你可以在函数的主体中使用 lambda 表达式。 lambda 表达式可以访问该封闭函数可访问的任何函数或数据成员。 你可以显式或隐式捕获 `this` 指针，以提供对封闭类的函数和数据成员的访问路径。 Visual Studio 2017 版本 15.3 或更新版本（及更新版本可用）：在原始对象超出范围后，当可能会执行代码的异步或并行操作将使用 lambda 时，按值捕获 /std:c++17（`this`）`[*this]`。

可以在函数中显式使用 `this` 指针，如下所示：

C++

```
// 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; });
}
```

你也可以隐式捕获 `this` 指针：

C++

```
void ApplyScale(const vector<int>& v) const
{
   for_each(v.begin(), v.end(),
      [=](int n) { cout << n * _scale << endl; });
}
```

以下示例显示封装小数位数值的 `Scale` 类。

C++

```
// 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. doesn't modify the vector.
    Scale s(3);
    s.ApplyScale(values);
}
```

此示例产生以下输出：

Output

```
3
6
9
12
```

备注

`ApplyScale` 函数使用 lambda 表达式打印小数位数值与 `vector` 对象中的每个元素的乘积。 lambda 表达式隐式捕获 **`this`** 指针，以便访问 `_scale` 成员。

[本文内容]

配合使用 Lambda 表达式和模板

示例

由于 lambda 表达式已类型化，因此你可以将其与 C++ 模板一起使用。 下面的示例显示 `negate_all` 和 `print_all` 函数。 `negate_all` 函数将一元 **`operator-`** 应用于 `vector` 对象中的每个元素。 `print_all` 函数将 `vector` 对象中的每个元素打印到控制台。

C++

```cpp
// 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);
}
```

此示例产生以下输出：

Output

```
34
-43
56
After negate_all():
-34
43
-56
```

备注

有关 C++ 模板的详细信息，请参阅模板。

[本文内容]

处理异常

示例

lambda 表达式的主体遵循结构化异常处理 (SEH) 和 C++ 异常处理的原则。 你可以在 lambda 表达式主体中处理引发的异常或将异常处理推迟至封闭范围。 以下示例使用 `for_each` 函数和 lambda 表达式将一个 `vector` 对象的值填充到另一个中。 它使用 `try`/`catch` 块处理对第一个矢量的无效访问。

C++

```
// 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;
    };
}
```

此示例产生以下输出：

Output

```
Caught 'invalid vector<T> subscript'.
```

备注

有关异常处理的详细信息，请参阅异常处理。

[本文内容]

配合使用 Lambda 表达式和托管类型 (C++/CLI)

示例

lambda 表达式的捕获子句不能包含具有托管类型的变量。 但是，你可以将具有托管类型的实际参数传递到 lambda 表达式的形式参数列表。 以下示例包含一个 lambda 表达式，它通过值捕获局部非托管变量 `ch`，并采用 System.String 对象作为其参数。

C++

```
// 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");
}
```

此示例产生以下输出：

Output

```
Hello!
```

备注

你还可以配合使用 lambda 表达式和 STL/CLR 库。 有关详细信息，请参阅 STL/CLR 库参考。

> ⓘ **重要**
>
> 以下公共语言运行时 (CLR) 托管实体中不支持 Lambda： `ref class`、 `ref struct`、 `value class` 以及 `value struct`。

[本文内容]

另请参阅

Lambda 表达式
Lambda 表达式语法
auto
function 类

find_if

<algorithm>

函数调用

模板

异常处理

STL/CLR 库参考

Last updated on 2025/11/07

C++ 中的 constexpr lambda 表达式

Visual Studio 2017 版本 15.3 及更高版本 （在 /std:c++17 模式和更高版本中可用）： 在常量表达式中允许初始化捕获或引入的每个数据成员时，可以将 lambda 表达式声明为 `constexpr`，或在常量表达式中使用它。

C++

```
int y = 32;
auto answer = [y]() constexpr
{
    int x = 10;
    return y + x;
};

constexpr int Increment(int n)
{
    return [n] { return n + 1; }();
}
```

如果 Lambda 结果满足 `constexpr` 函数的要求，则 Lambda 是隐式的 `constexpr`：

C++

```
auto answer = [](int n)
{
    return 32 + n;
};

constexpr int response = answer(10);
```

如果 lambda 是隐式或显式的 `constexpr`，并且将其转换为函数指针，则生成的函数也是 `constexpr`：

C++

```
auto Increment = [](int n)
{
    return n + 1;
};

constexpr int(*inc)(int) = Increment;
```

另请参阅

C++ 语言参考
C++ 标准库中的函数对象

函数调用

for_each

Last updated on 2025/11/07

数组 (C++)

数组是相同类型的对象序列，它们占据一块连续的内存区。 传统的 C 样式数组是许多 bug 的根源，但至今仍很常用，尤其是在较旧的代码库中。 在新式 C++ 中，我们强烈建议使用 std::vector 或 std::array，而不是本部分所述的 C 样式数组。 这两种标准库类型都将其元素存储为连续的内存块。 但是，它们提供更高的类型安全性，并支持保证指向序列中有效位置的迭代器。 有关详细信息，请参阅容器。

堆栈声明

在 C++ 数组声明中，数组大小在变量名称之后指定，而不是像在其他某些语言中那样的在类型名称之后指定。 以下示例声明了要在堆栈上分配的 1000 个双精度值的数组。 元素数量必须以整数文本或常量表达式的形式提供。 这是因为，编译器必须知道要分配多少堆栈空间；它不能使用在运行时计算的值。 为数组中的每个元素分配默认值 0。 如果你不指定默认值，则每个元素最初包含恰好位于该内存位置的任意随机值。

C++

```
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] << " ";
}
```

数组中的第一个元素是第 0 个元素。 最后一个元素是 (n-1) 元素，其中 n 是数组可以包含的元素数量。 声明中的元素数量必须是整数类型，且必须大于 0。 你需要负责确保程序永远不会将大于 `(size - 1)` 的值传递给下标运算符。

仅当数组是 `struct` 或 `union` 中的最后一个字段并启用了 Microsoft 扩展（未设置 `/Za` 或 `/permissive-`）时，零大小的数组才是合法的。

分配和访问基于堆栈的数组的速度比基于堆的数组更快。 但是，堆栈空间是有限的。 数组元素的数量不能太大，以免占用过多的堆栈内存。 多大的数量算作“太大”取决于程序。 可以使用分析工具来确定数组是否太大。

堆声明

你可能需要一个太大的、以致无法在堆栈上分配的数组，或者需要一个其大小在编译时未知的数组。 可以使用 new[] 表达式在堆上分配此数组。 运算符返回指向第一个元素的指针。 下标运算符处理指针变量的方式与处理基于堆栈的数组的方式相同。 还可以使用指针算术将指针移到数组中的任意元素。 你需要负责确保：

- 始终保留原始指针地址的副本，以便可以在不再需要数组时删除内存。
- 不会将指针地址递增或递减至超过数组边界。

以下示例演示在运行时如何在堆上定义一个数组。 其中演示了如何使用下标运算符和指针算术来访问数组元素：

C++

```
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);
}
```

初始化数组

可以在循环中、以每次一个元素的方式或者在单个语句中初始化数组。 以下两个数组的内容是相同的：

C++

```
    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 };
```

将数组传递给函数

将数组传递给函数时，该数组将作为指向第一个元素的指针传递，无论它是基于堆栈的数组还是基于堆的数组。 指针不包含任何其他大小或类型信息。 此行为称为指针衰减。 将数组传递给函数时，始终必须在单独的参数中指定元素数量。 此行为还意味着将数组传递给函数时不会复制数组元素。 若要防止函数修改元素，请将参数指定为指向 `const` 元素的指针。

以下示例演示了一个接受数组和长度的函数。 指针指向原始数组而不是副本。 由于参数不是 `const`，因此该函数可以修改数组元素。

C++

```
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]
    }
}
```

将数组参数 `p` 声明并定义为 `const`，使其在函数块中为只读：

C++

```
void process(const double *p, const size_t len);
```

也可以用这些方式声明相同的函数，而无需改变行为。 数组仍作为指向第一个元素的指针传递：

C++

```
// 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);
```

多维数组

从其他数组构造的数组是多维数组。 通过按顺序放置多个括起来的常数表达式来指定这些多维数组。 例如，考虑此声明：

C++

```
int i2[5][7];
```

它指定类型为 `int` 的数组，从概念上以五行七列的二维矩阵排列，如下图所示：

可以声明具有初始化表达式列表的多维数组（如初始化表达式中所述）。 在这些声明中，可以省略指定第一维的边界的常数表达式。 例如：

C++

```
// 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 }
};
```

前面的声明定义四列三行的数组。 行表示工厂，列表示从工厂装运到的市场。 值是从工厂运输到市场的成本。 忽略数组的第一个维度，但编译器会通过检查该初始值设定项来填充它。

对 n 维数组类型使用间接寻址运算符 (*) 将生成 n-1 维数组。 如果 n 为 1，则将生成标量（或数组元素）。

C++ 数组按行优先顺序存储。 行优先顺序意味着最后一个下标变化最快。

示例

还可以在函数声明中省略多维数组第一个维的边界规范，如下所示：

C++

```
// 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;
}
```

Output

```
The minimum cost to Market 3 is: 17.29
```

编写函数 `FindMinToMkt`，以便添加不需要更改任何代码而仅需重新编译的新工厂。

初始化数组

具有类构造函数的对象数组由构造函数初始化。 如果初始化表达式列表中的项少于数组中的元素，则默认的构造函数将用于剩余元素。 如果没有为类定义默认构造函数，初始化表达式列表必须完整，即数组中的每个元素都必须有一个初始化表达式。

考虑定义了两个构造函数的 `Point` 类：

C++

```
// 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()
{
}
```

`aPoint` 的第一个元素是使用构造函数 `Point( int, int )` 构造的；剩余的两个元素是使用默认构造函数构造的。

静态成员数组（是否为 `const`）可在其定义中进行初始化（类声明的外部）。 例如：

C++

```
// 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()
{
}
```

访问数组元素

您可以使用数组下标运算符（`[ ]`）访问数组的各个元素。 如果使用不带下标的一维数组的名称，它将评估为指向数组第一个元素的指针。

C++

```
// 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.
}
```

使用多维数组时，可以在表达式中使用各种组合。

C++

```
// 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.
}
```

在前面的代码中，`multi` 是 **`double`** 类型的三维数组。 `p2multi` 指针指向大小为三的 **`double`** 类型的数组。 在此示例中，该数组与一个、两个和三个下标一起使用。 尽管更为常见的是指定所有下标（如在 `cout` 语句中），但有时选择数组元素的特定子集会很有用，如 `cout` 后面的语句中所示。

重载下标运算符

与其他运算符相似，下标运算符（`[]`）也可由用户重新定义。 如果没有重载下标运算符，下标运算符的默认行为是使用以下方法组合数组名称和下标：

`*((array_name) + (subscript))`

像涉及指针类型的所有加法中一样，缩放将自动执行以调整类型的大小。 结果值不是来自 的 n 个字节，而是数组的第 n 个元素`array_name`。 有关此转换的详细信息，请参阅加法运算符。

同样，对于多维数组，将使用以下方法获取地址：

`((array_name) + (subscript1 * max2 * max3 * ... * maxn) + (subscript2 * max3 * ... * maxn) + ... + subscriptn))`

表达式中的数组

当数组类型的标识符出现在 **`sizeof`**、address-of（`&`）或引用的初始化以外的表达式中时，该标识符将转换为指向第一个数组元素的指针。 例如：

C++

```
char szError1[] = "Error: Disk drive not ready.";
char *psz = szError1;
```

指针 `psz` 指向数组 `szError1` 的第一个元素。 与指针不同，数组不是可修改的左值。 因此，以下赋值是非法的：

C++

```
szError1 = psz;
```

另请参阅

std::array

Last updated on 2025/11/07

引用 (C++)

与指针相似的是，引用将存储位于内存中其他位置的对象的地址。 与指针不同，初始化之后的引用无法引用不同的对象或设置为 null。 有两种类型的引用：引用命名变量的 *lvalue* 引用和引用*临时对象*的 rvalue 引用。 `&` 运算符表示 lvalue 引用，`&&` 运算符表示 rvalue 引用或通用引用（rvalue 或 lvalue），具体取决于上下文。

可以通过以下语法声明引用：

> [*storage-class-specifiers*][*cv-qualifiers*] *type-specifiers* [*ms-modifier*] *声明符* [`=` *表达式*] `;`

可以使用指定引用的任何有效声明符。 除非引用是对函数或数组类型的引用，否则应用以下简化语法：

> [*storage-class-specifiers*][*cv-qualifiers*] *type-specifiers* [`&` or `&&`] [*cv-qualifiers*] *identifier* [`=` *expression*] `;`

使用以下序列声明引用：

1. 声明说明符：
   - 可选存储类说明符。
   - 可选 `const` 和/或 `volatile` 限定符。
   - 类型说明符：类型的名称。
2. 声明符：
   - 可选的 Microsoft 专用修饰符。 有关详细信息，请参阅 Microsoft 专用修饰符。
   - `&` 运算符或 `&&` 运算符。
   - 可选 `const` 和/或 `volatile` 限定符。
   - 标识符。
3. 可选初始值设定项。

指向数组和函数的指针的更复杂的声明符形式也适用于对数组和函数的引用。 有关详细信息，请参阅指针。

多个声明符和初始值设定项可能出现在一个声明说明符后面的逗号分隔的列表中。 例如：

```
C++
```

```
int &i;
int &i, &j;
```

引用、指针和对象可以一起声明：

C++

```
int &ref, *ptr, k;
```

引用保留对象的地址，但语法行为与对象一样。

在下面的程序中，请注意对象的名称 `s` 和对象的引用 `SRef` 可在程序中以相同的方式使用：

示例

C++

```
// references.cpp
#include <stdio.h>
struct S {
   short i;
};

int main() {
   S  s;   // Declare the object.
   S& SRef = s;   // Declare and initialize 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);
}
```

Output

```
3
3
4
4
```

另请参阅

引用类型函数自变量
引用类型函数返回
对指针的引用

Last updated on 2025/11/07

Lvalue 引用声明符： `&`

保留对象的地址，但行为方式在语法上与对象相似。

语法

`lvalue-reference-type-id`?

`type-specifier-seq` **`&`** `attribute-specifier-seq` 选择 `ptr-abstract-declarator` 选择

注解

您可以将左值引用视为对象的另一名称。 左值引用声明由说明符的可选列表后跟一个引用声明符组成。 引用必须初始化且无法更改。

地址可转换为给定指针类型的任何对象也可转换为相似的引用类型。 例如，地址可转换为类型 `char *` 的任何对象也可转换为类型 `char &`。

不要将引用声明与 address-of 运算符的用法混淆。 `&`identifier 前面有 或 **`int`** 之类的类型时，identifier 将声明为对该类型的引用**`char`**。 `&`identifier 前面没有类型时，用法就是 address-of 运算符的用法。

示例

以下示例通过声明 `Person` 对象和对该对象的引用演示了引用声明符。 由于 `rFriend` 是对 `myFriend` 的引用，因此更新任一变量都将更改同一对象。

C++

```
// 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;
}
```

Output

```
Bill is 40
```

另请参阅

参考
引用类型函数参数
引用类型函数返回
对指针的引用

Last updated on 2025/11/07

引用声明符： `&&`

保留对右值表达式的引用。

语法

`rvalue-reference-type-id`?

`type-specifier-seq` **`&&`** `attribute-specifier-seq` 选择`ptr-abstract-declarator`选择

注解

利用右值引用，您可以将左值与右值区分开。 lvalue 引用和 rvalue 引用在语法和语义上相似，但它们遵循的规则稍有不同。 有关 lvalue 和 rvalue 的详细信息，请参阅 Lvalue 和 Rvalue。 有关详细信息，请参阅lvalue 引用声明符： &。

下面的章节介绍了 rvalue 引用如何支持“移动语义”和“完美转移”的实现。

移动语义

Rvalue 引用支持“移动语义”的实现，这可以显著提高应用程序的性能。 利用移动语义，你可以编写将资源（如动态分配的内存）从一个对象转移到另一个对象的代码。 移动语义很有效，因为它允许从临时对象（无法在程序中的其他位置引用）转移资源。

若要实现移动语义，通常可以向类提供“移动构造函数”，或者提供移动赋值运算符 () `operator=`。 其源是右值的复制和赋值操作随后会自动利用移动语义。 与默认复制构造函数不同，编译器不提供默认移动构造函数。 有关如何编写和使用移动构造函数详细信息，请参阅移动构造函数和移动赋值运算符。

您还可以重载普通函数和运算符以利用移动语义。 Visual Studio 2010 将移动语义引入到 C++ 标准库。 例如， `string` 类实现了使用移动语义的操作。 请考虑以下串联几个字符串并输出结果的示例：

C++

```
// 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;
}
```

在 Visual Studio 2010 之前，每次对 `operator+` 的调用都会分配和返回新的临时 `string` 对象（rvalue）。 `operator+` 不能将一个字符串附加到另一个字符串，因为它不知道源字符串是 lvalue 还是 rvalue。 如果两个源字符串都是 lvalue，则它们可能会在程序中的其他位置引用，因此不能被修改。 可以利用 rvalue 引用将 `operator+` 修改为采用 rvalue，而 rvalue 不能在程序中的其他位置引用。 通过此更改，`operator+` 现在可以将一个字符串附加到另一个字符串。 这可以显著减少 `string` 类必须执行的动态内存分配的数量。 有关 `string` 类的详细信息，请参阅 basic_string 类。

当编译器不能使用返回值优化 (RVO) 或命名返回值优化 (NRVO) 时，移动语义也很有用。 在这些情况下，如果类型定义了移动构造函数，则编译器将调用该函数。

若要更好地了解移动语义，请考虑将元素插入 `vector` 对象的示例。 如果超出 `vector` 对象的容量，则 `vector` 对象必须为其元素重新分配足够的内存，然后将所有元素复制到其他内存位置，以便为插入的元素腾出空间。 当插入操作复制元素时，它首先创建一个新元素。 然后它调用复制构造函数将数据从上一个元素复制到新元素。 最后，它会销毁上一个元素。 利用移动语义，可以直接移动对象而不必执行成本高昂的内存分配和复制操作。

若要在 `vector` 示例中利用移动语义，则可以编写将数据从一个对象移动到另一个对象的移动构造函数。

有关在 Visual Studio 2010 中引入移动语义到 C++ 标准库的详细信息，请参阅 C++ 标准库。

完美转移

完美转发可减少对重载函数的需求，并有助于避免转发问题。 当编写将引用作为其参数的泛型函数时，会引发“转移问题”。 如果它将这些参数传递 (或 *转移*) 到另一个函数，例如，如果它采用类型 `const T&` 的参数，则被调用的函数无法修改该参数的值。 如果泛型函数采用类型 `T&` 的参数，则无法使用 rvalue（如临时对象或整数文本）来调用该函数。

通常，若要解决此问题，则必须提供为其每个参数采用 `T&` 和 `const T&` 的重载版本的泛型函数。 因此，重载函数的数量将基于参数的数量呈指数方式增加。 rvalue 引用允许编写一个接受任意参数的函数版本。 然后，该函数可以将它们转移到另一个函数，就像直接调用了另一个函数一样。

请考虑以下声明了四个类型 `W` `X`、`Y` 和 `Z` 的示例。 每个类型的构造函数采用 `const` 和非 `const` lvalue 引用的不同组合作为其参数。

```
C++
```

```
struct W
{
   W(int&, int&) {}
};

struct X
{
   X(const int&, int&) {}
};

struct Y
{
   Y(int&, const int&) {}
};

struct Z
{
   Z(const int&, const int&) {}
};
```

假定您要编写生成对象的泛型函数。 以下示例演示了一种编写此函数的方式：

C++

```
template <typename T, typename A1, typename A2>
T* factory(A1& a1, A2& a2)
{
   return new T(a1, a2);
}
```

以下示例演示了对 `factory` 函数的有效调用：

C++

```
int a = 4, b = 5;
W* pw = factory<W>(a, b);
```

但是，以下示例不包含对 `factory` 函数的有效调用。 这是因为 `factory` 接受可修改 lvalue 引用作为它的参数，但它通过使用 rvalue 调用：

C++

```
Z* pz = factory<Z>(2, 2);
```

通常，若要解决此问题，您必须为 `factory` 和 `A&` 的参数的每个组合创建一个重载版本的 `const A&` 函数。 利用右值引用，您可以编写一个版本的 `factory` 函数，如以下示例所示：

C++

```
template <typename T, typename A1, typename A2>
T* factory(A1&& a1, A2&& a2)
{
   return new T(std::forward<A1>(a1), std::forward<A2>(a2));
}
```

此示例使用右值引用作为 `factory` 函数的参数。 std::forward 函数的用途是将 factory 函数的参数转移给模板类的构造函数。

以下示例演示了使用修改后的 `main` 函数创建 `factory`、`W`、`X` 和 `Y` 类的实例的 `Z` 函数。 修改后的 `factory` 函数会将其参数（左值和右值）转发给适当的类构造函数。

C++

```
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;
}
```

rvalue 引用的属性

您可以重载采用左值引用和右值引用的函数。

通过重载函数来采用 `const` lvalue 引用或 rvalue 引用，可以编写代码来区分不可更改的对象（lvalue）和可修改的临时值（rvalue）。 可以将对象传递给采用 rvalue 引用的函数，除非该对象标记为 `const`。 以下示例演示了函数 `f`，该函数将被重载以采用左值引用和右值引用。 `main` 函数同时使用左值和右值来调用 `f`。

C++

```
// 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 can't 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());
}
```

该示例产生下面的输出：

Output

```
In f(const MemoryBlock&). This version can't modify the parameter.
In f(MemoryBlock&&). This version can modify the parameter.
```

在此示例中，对 `f` 的第一个调用将局部变量（左值）作为其自变量传递。 对 `f` 的第二个调用将临时对象作为其自变量传递。 由于无法在程序中的其他位置引用临时对象，因此调用会绑定到采用 rvalue 引用的 `f` 重载版本，该版本可以随意修改对象。

编译器将已命名的右值引用视为左值，而将未命名的右值引用视为右值。

采用 rvalue 引用作为参数的函数将参数视为函数正文中的 lvalue。 编译器将命名的 rvalue 引用视为 lvalue。 这是因为命名对象可由某个程序的多个部分引用。 允许程序的多个部分从该对象修改或删除资源是危险的。 例如，如果程序的多个部分尝试从同一对象转移资源，则只有第一个部分能成功转移。

以下示例演示了函数 `g`，该函数将被重载以采用左值引用和右值引用。 函数 `f` 采用右值引用作为其参数（已命名的右值引用），并返回右值引用（未命名的右值引用）。 在从 `g` 到 `f` 的调用中，重载决策选择采用左值引用的 `g` 版本，因为 `f` 的主体将其参数视为左值。 在从 `g` 到 `main` 的调用中，重载决策选择采用右值引用的 `g` 版本，因为 `f` 返回右值引用。

C++

```
// 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()));
}
```

该示例产生下面的输出：

C++

```
In g(const MemoryBlock&).
In g(MemoryBlock&&).
```

在此示例中，`main` 函数将 rvalue 传递给 `f`。 `f` 的主体将其命名参数视为左值。 从 `f` 到 `g` 的调用会将参数绑定到左值引用（第一个重载版本的 `g`）。

- **可以将 lvalue 强制转换为 rvalue 引用。**

C++ 标准库 std::move 函数可以将某个对象转换为对该对象的 rvalue 引用。 也可以使用 `static_cast` 关键字将 lvalue 强制转换为 rvalue 引用，如以下示例所示：

C++

```
// 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));
}
```

该示例产生下面的输出：

C++

```
In g(const MemoryBlock&).
In g(MemoryBlock&&).
```

函数模板会推导出其模板自变量类型，然后使用引用折叠规则。

将其参数传递（或“转发”）给另一个函数的函数模板是一种常见模式。 了解模板类型推导如何对采用 rvalue 引用的函数模板起作用，这很重要。

如果函数参数是右值，则编译器将参数推导为右值引用。 例如，假设将 rvalue 引用（针对 `X` 类型的对象）传递给采用类型 `T&&` 作为其参数的函数模板。 模板参数推导推断 `T` 为 `X`，因此该参数具有类型 `X&&`。 如果函数参数是 lvalue 或 **`const`** lvalue，则编译器将其类型推导为该类型的 lvalue 引用或 **`const`** lvalue 引用。

以下示例声明了一个结构模板，然后针对不同引用类型对其进行了专用化。 `print_type_and_value` 函数采用右值引用作为其参数，并将它转发给适当专用版本的 `S::print` 方法。 `main` 函数演示了调用 `S::print` 方法的各种方式。

C++

```
// 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());
}
```

该示例产生下面的输出：

C++

```
print<T&>: first
print<const T&>: second
print<T&&>: third
print<const T&&>: fourth
```

为了解析每个对 `print_type_and_value` 函数的调用，编译器首先会执行模板参数推导。 然后，编译器再用推导出的模板参数替换参数类型时应用引用折叠规则。 例如，将局部变量 `s1` 传递给 `print_type_and_value` 函数将导致编译器生成以下函数签名：

C++

```
print_type_and_value<string&>(string& && t)
```

编译器使用引用折叠规则将签名缩短：

C++

```
print_type_and_value<string&>(string& t)
```

此版本的 `print_type_and_value` 函数随后将其参数转发到正确的专用版本的 `S::print` 方法。

下表汇总了模板自变量类型推导的引用折叠规则：

展开表

展开类型	折叠类型
`T& &`	`T&`
`T& &&`	`T&`
`T&& &`	`T&`
`T&& &&`	`T&&`

模板自变量推导是实现完美转发的重要因素。 完美转移部分更详细地介绍了完美转移。

总结

右值引用可将左值和右值区分开。 为了提高应用程序的性能，它们可以消除不必要的内存分配和复制操作需求。 它们还允许编写接受任意参数的函数。 该函数可以将它们转移到另一个函数，就像直接调用了另一个函数一样。

另请参阅

带一元运算符的表达式
Lvalue 引用声明符：&
Lvalues 和 rvalues
移动构造函数和移动赋值运算符 (C++)
C++ 标准库

Last updated on 2025/11/07

引用类型函数自变量

向函数传递引用而非大型对象的效率通常更高。 这使编译器能够在保持已用于访问对象的语法的同时传递对象的地址。 请考虑以下使用了 `Date` 结构的示例：

C++

```
// 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;
}
```

前面的代码显示通过引用传递的结构的成员是通过成员选择运算符 (.) 访问的，而不是通过指针成员选择运算符 (->) 访问的。

尽管作为引用类型传递的参数遵循了非指针类型的语法，但它们仍然保留了指针类型的一个重要特征：除非被声明为 `const`，否则它们是可以修改的。 由于上述代码的目的不是修改对象 `date`，因此更合适的函数原型是：

```
C++
long DateOfYear( const Date& date );
```

此原型将确保函数 `DateOfYear` 不会更改其参数。

任何其原型采用引用类型的函数都能接受其所在位置的相同类型的对象，因为存在从 typename 到 typename& 的标准转换。

另请参阅

参考

Last updated on 2025/11/07

引用类型函数返回

可将函数声明为返回引用类型。 做出此类声明有两个原因：

- 返回的信息是一个返回引用比返回副本更有效的足够大的对象。
- 函数的类型必须为左值。
- 引用的对象在函数返回时不会超出范围。

正如通过引用传递大型对象 to 函数会更有效一样，通过引用返回大型对象 from 函数也会更有效。 引用返回协议使得不必在返回前将对象复制到临时位置。

当函数的计算结果必须为左值时，引用返回类型也可能很有用。 大多数重载运算符属于此类别，尤其是赋值运算符。 重载运算符中介绍了重载运算符。

示例

请考虑 `Point` 示例：

C++

```
// 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";
}
```

输出

Output

```
x = 7
y = 9
```

请注意，函数 `x` 和 `y` 被声明为返回引用类型。 这些函数可在赋值语句的每一端上使用。

另请注意在 main 中，ThePoint 对象停留在范围中，因此其引用成员仍处于活动状态，可以安全地访问。

除以下情况之外，引用类型的声明必须包含初始值设定项：

- 显式 `extern` 声明
- 类成员的声明
- 类中的声明
- 函数的自变量或函数的返回类型的声明

返回局部变量地址时的注意事项

如果在局部范围中声明某个对象，则该对象会在函数返回时销毁。 如果函数返回对该对象的引用，则当调用方尝试使用 null 引用时，该引用可能会在运行时导致访问冲突。

C++

```
// C4172 means Don't do this!!!
Foo& GetFoo()
{
    Foo f;
    ...
```

```
    return f;
} // f is destroyed here
```

在这种情况下，编译器将发出一个警告：`warning C4172: returning address of local variable or temporary`。 在简单程序中，如果调用方在覆盖内存位置之前访问引用，则有时可能不会发生访问冲突。 这纯属运气。 请注意该警告。

另请参阅

参考

Last updated on 2025/11/07

对指针的引用

声明对指针的引用的方式与声明对对象的引用差不多。 对指针的引用是一个可像常规指针一样使用的可修改的值。

示例

此代码示例演示了使用指向指针的指针与使用对指针的引用之间的差异。

虽然 `Add1` 和 `Add2` 函数的调用方式不同，但它们在功能上是等效的。 二者的区别是，`Add1` 使用双间接寻址，而 `Add2` 利用了对指针的引用的便利性。

C++

```
// 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 );
   }
}
```

Output

```
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.
```

另请参阅

参考

Last updated on 2025/11/07

指针 (C++)

指针是一个变量，可以存储一个对象的内存地址。 指针在 C 和 C++ 中广泛用于三个主要用途：

- 在堆上分配新对象，
- 将函数传递给其他函数
- 循环访问数组或其他数据结构中的元素。

在 C 样式编程中，原始指针用于所有这些场景。 但是，原始指针会导致许多严重的编程错误。 因此，强烈建议不要使用它们，除非它们提供了显著的性能优势，并且对于哪个指针是负责删除对象的指针没有歧义。 新式 C++ 提供了智能指针用于分配对象，提供了迭代器用于遍历数据结构，还提供了 Lambda 表达式用于传递函数。 通过使用这些语言和库设施，而不是原始指针，可使程序更安全、更易于调试，以及更易于理解和维护。 有关详细信息，请参阅智能指针、迭代器和 Lambda 表达式。

本部分内容

- 原始指针
- 固定和可变指针
- new 和 delete 运算符
- 智能指针
- 如何：创建和使用 unique_ptr 实例
- 如何：创建和使用 shared_ptr 实例
- 如何：创建和使用共享 weak_ptr 实例
- 如何：创建和使用 CComPtr 和 CComQIPtr 实例

另请参阅

迭代器
Lambda 表达式

Last updated on 2025/11/07

原始指针 (C++)

指针是一种变量。 它将对象的地址存储在内存中，并用于访问该对象。 原始指针是指其生存期不受封装对象控制的指针，例如*智能指针*。 可以为原始指针分配另一个非指针变量的地址，也可以为其分配 nullptr 值。 未分配值的指针包含随机数据。

还可以取消引用指针以检索其指向的对象的值。 成员访问运算符提供对对象成员的访问权限。

C++

```
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
```

指针可以指向类型化对象或指向 `void`。 当程序在内存中的堆上分配对象时，它会以指针的形式接收该对象的地址。 此类指针称为“拥有指针”。 当不再需要堆分配的对象时，必须使用拥有指针（或其副本）显式释放该对象。 未能释放内存会导致内存泄漏，并使该内存位置无法供计算机上的任何其他程序使用。 必须使用 `new`（或 `delete`）释放使用 `delete[]` 分配的内存。 有关详细信息，请参阅 new 和 delete 运算符。

C++

```
MyClass* mc = new MyClass(); // allocate object on the heap
mc->print(); // access class member
delete mc; // delete object (please don't forget!)
```

指针（如果未声明为 `const`）可以递增或递减，以指向内存中的另一个位置。 此操作称为“指针算术”。 它用于在 C 风格编程中循环访问数组或其他数据结构中的元素。 `const` 指针不能指向不同的内存位置，从这个意义上说，它类似于引用。 有关详细信息，请参阅 const 和 volatile 指针。

C++

```
// 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.
```

在 64 位操作系统上，指针的大小为 64 位。 系统的指针大小决定了它可以拥有的可寻址内存量。 指针的所有副本都指向同一内存位置。 指针（以及引用）在 C++ 中广泛用于向/从函数传递较大的对象。 复制对象的地址通常比复制整个对象更加高效。 定义函数时，除非希望函数修改对象，否则请将指针参数指定为 `const`。 通常， `const` 引用是将对象传递给函数的首选方式，除非对象的值可能为 `nullptr`。

指向函数的指针使函数能够传递给其他函数。 它们用于 C 风格编程中的“回调”。 新式 C++ 使用 lambda 表达式来实现此目的。

初始化和成员访问

下面的示例展示了如何声明、初始化和使用原始指针。 它使用 `new` 初始化，以指向堆上分配的对象，必须显式删除（`delete`）该对象。 该示例还展示了与原始指针相关的一些危险。 （请记住，此示例是 C 风格的编程，而不是新式 C++！）

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(pmc);
    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(*pmc);
    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
}
```

指针算术和数组

指针和数组密切相关。 当数组按值传递给函数时，它将作为指向第一个元素的指针传递。 以下示例演示了指针和数组的以下重要属性：

- `sizeof` 运算符返回数组的总大小（以字节为单位）
- 若要确定元素数目，请将总字节数除以一个元素的大小
- 当数组被传递给函数时，它会衰减为指针类型

- 当 `sizeof` 运算符应用于指针时，它将返回指针大小，例如，x86 上为 4 个字节，x64 上为 8 个字节

C++

```
#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);
}
```

某些算术运算可用于非 `const` 指针，以使其指向另一个内存位置。 指针使用 `++`、`+=`、`-=` 和 `--` 运算符递增和递减。 此方法可用于数组，在非类型化数据的缓冲区中尤其有用。 `void*` 按一个 `char` 的大小（1 个字节）递增。 类型化指针按其指向的类型的大小递增。

以下示例演示如何使用指针算术访问 Windows 上位图中的单个像素。 请注意 `new` 和 `delete` 以及取消引用运算符的使用。

C++

```
#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 = 0x0; // 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*` 指针

指向 `void` 的指针仅指向原始内存位置。 有时需要使用 `void*` 指针，例如在 C++ 代码和 C 函数之间传递时。

将类型化指针强制转换为 `void` 指针时，内存位置的内容保持不变。 但是，类型信息会丢失，因此无法执行递增或递减操作。 例如，可以将内存位置从 `MyClass*` 强制转换为 `void*`，然后再转换回 `MyClass*`。 此类操作本质上容易出错，需要非常小心以避免错误。 新式 C++ 几乎在所有情况下都不鼓励使用 `void` 指针。

C++

```
//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"
    delete(mc);

    // 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(pvoid);
}
```

指向函数的指针

在 C 风格的编程中，函数指针主要用于将函数传递给其他函数。 此方法使调用方能够在不修改函数的情况下自定义函数的行为。 在新式 C++ 中，lambda 表达式提供了相同的功能，并且提供了更高的类型安全性和其他优势。

函数指针声明指定指向函数必须具有的签名：

C++

```
// 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);
```

以下示例展示了函数 `combine`，该函数将接受 `std::string` 并返回 `std::string` 的任何函数作为参数。 根据传递给 `combine` 的函数，它将在前面或后面添加字符串。

C++

```
#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";
}
```

另请参阅

智能指针
间接运算符： *
地址运算符：
欢迎回到 C++

Last updated on 2025/11/07

固定和可变指针

const 和 volatile 关键字可更改处理指针的方式。 `const` 关键字指定指针在初始化后无法修改；此后指针将受到保护，防止进行修改。

`volatile` 关键字指定与后跟的名称关联的值可由用户应用程序中的操作以外的操作修改。 因此， `volatile` 关键字对于声明共享内存中可由多个进程访问的对象或用于与中断服务例程通信的全局数据区域很有用。

如果某个名称被声明为 `volatile`，则每当程序访问该名称时，编译器都会重新加载内存中的值。 这将显著减少可能的优化。 但是，当对象的状态可能意外更改时，这是保证可预见的程序性能的唯一方法。

若要将指针指向的对象声明为 `const` 或 `volatile`，请使用以下形式的声明：

C++

```
const char *cpch;
volatile char *vpch;
```

若要将指针的值（即指针中存储的实际地址）声明为 `const` 或 `volatile`，请使用以下形式的声明：

C++

```
char * const pchc;
char * volatile pchv;
```

C++ 语言会阻止将允许修改声明为 `const` 的对象或指针的赋值。 此类赋值会移除用来声明对象或指针的信息，从而违反原始声明的意图。 请考虑以下声明：

C++

```
const char cch = 'A';
char ch = 'B';
```

假定前面声明了两个对象（`cch` 类型的 和 `ch` 类型的），以下声明/初始化将是有效的：

C++

```
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;
```

以下声明/初始化存在错误。

C++

```
char *pch2 = &cch;   // Error
char *const pch3 = &cch;   // Error
```

`pch2` 的声明声明了一个可以用来修改常量对象的指针，因此不允许使用。 `pch3` 的声明指定指针是常量，而不是对象；与不允许使用 `pch2` 的原因相同，也不允许使用该声明。

以下八个赋值显示了通过指针进行的赋值以及对前面的声明的指针值的更改；现在，假设 `pch1` 到 `pch8` 的初始化是正确的。

C++

```
*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
```

声明为 **`volatile`** 或 **`const`** 和 **`volatile`** 的组合的指针遵循相同的规则。

指向 **`const`** 对象的指针通常用于函数声明中，如下所示：

C++

```
errno_t strcpy_s( char *strDestination, size_t numberOfElements, const char
*strSource );
```

前面的语句声明了函数 strcpy_s，其中三个自变量中的两个是指向 **`char`** 的类型指针。 由于参数是按引用而不是按值传递的，因此，如果未将 `strDestination` 声明为 `strSource`，则该函数可以自由修改 `strSource` 和 **`const`**。 将 `strSource` 声明为 **`const`** 可向调用方保证调用的函数无法更改 `strSource`。

> ⓘ **注意**
>
> 由于存在从 typename* 到 **`const`** typename* 的标准转换，因此将 `char *` 类型的自变量传递到 strcpy_s 是合法的。 但是，反之则不行；不存在从对象或指针中移除 **`const`** 特性的隐

式转换。

给定类型的 `const` 指针可以分配给同一类型的指针。 但是，非 `const` 类型的指针不能赋给 `const` 指针。 以下代码显示了正确和错误的赋值：

C++

```
// const_pointer.cpp
int *const cpObject = 0;
int *pObject;

int main() {
pObject = cpObject;
cpObject = pObject;   // C3892
}
```

以下示例显示了当有指针指向某个指向对象的指针时如何将对象声明为 const。

C++

```
// 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;
}
```

另请参阅

指针
原始指针

Last updated on 2025/11/07

`new` 和 `delete` 运算符

C++ 支持使用 new 和 delete 运算符动态分配和解除分配对象。 这些运算符为来自称为“自由存储”（也称为“堆”）的池中的对象分配内存。 `new` 运算符调用特殊函数 operator new，`delete` 运算符调用特殊函数 operator delete。

有关包含 C 运行时库和 C++ 标准库中的库文件的列表，请参阅 CRT 库功能。

`new` 运算符

编译器将如下语句转换为对函数 `operator new` 的调用：

```
C++
char *pch = new char[BUFFER_SIZE];
```

如果请求的存储空间为零字节，`operator new` 将返回指向不同对象的指针。 也就是说，重复调用 `operator new` 会返回不同的指针。

如果分配请求的内存不足，`operator new` 会引发 `std::bad_alloc` 异常。 或者，如果使用了 placement 形式 `nullptr`，或者链接在非引发的 支持中，它将返回 `new(std::nothrow)`。 `operator new` 有关详细信息，请参阅分配失败行为。

下表中描述了 `operator new` 函数的两个范围。

`operator new` 函数的范围

展开表

运算符	Scope
`::operator new`	全局
class-name `::operator new`	类

`operator new` 的第一个自变量必须为 `size_t` 类型，且返回类型始终为 `void*`。

在使用 `operator new` 运算符分配内置类型的对象、不包含用户定义的 `new` 函数的类类型的对象和任何类型的数组时，将调用全局 `operator new` 函数。 在使用 `new` 运算符分配类类型的对象时（其中定义了 `operator new`），将调用该类的 `operator new`。

为类定义的 `operator new` 函数是静态成员函数（不能是虚函数），该函数隐藏此类类型的对象的全局 `operator new` 函数。 考虑 `new` 用于分配内存并将内存设为给定值的情况：

C++

```
#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;
}
```

用括号包含的提供给 `new` 的自变量将作为 `Blanks::operator new` 自变量传递给 `chInit`。 但是，全局 `operator new` 函数将被隐藏，从而导致以下代码生成错误：

C++

```
Blanks *SomeBlanks = new Blanks;
```

编译器在类声明中支持成员数组 `new` 和 `delete` 运算符。 例如：

C++

```
class MyClass
{
public:
   void * operator new[] (size_t)
   {
      return 0;
   }
   void   operator delete[] (void*)
   {
   }
};

int main()
{
```

```
    MyClass *pMyClass = new MyClass[5];
    delete [] pMyClass;
}
```

分配失败行为

C++ 标准库中的 `new` 函数支持自 C++98 以来在 C++ 标准中指定的行为。 如果分配请求的内存不足，`operator new` 会引发 std::bad_alloc 异常。

较旧的 C++ 代码会为失败的分配返回 null 指针。 如果你的代码需要非引发版本的 `new`，请将程序链接到 *`nothrownew.obj`*。 *`nothrownew.obj`* 文件将全局 `operator new` 替换为分配失败时返回 `nullptr` 的版本。 `operator new` 不再引发 `std::bad_alloc`。 有关 *`nothrownew.obj`* 和其他链接器选项文件的详细信息，请参阅链接选项。

不能将检查全局 `operator new` 异常的代码与检查同一个应用程序中的 null 指针的代码混合使用。 但是，仍可以创建不同行为的类本地 `operator new`。 这种可能性意味着编译器在默认情况下必须以防御方式行事，并在 `new` 调用中包含对 null 指针返回的检查。 有关优化这些编译器检查的方法的详细信息，请参阅 /Zc:throwingnew。

处理内存不足

从 `new` 表达式中测试失败分配的方式取决于是使用标准异常机制还是使用 `nullptr` 返回。 标准 C++ 要求分配器引发 `std::bad_alloc` 或派生自 `std::bad_alloc` 的类。 可以处理此类异常，如以下示例所示：

C++

```
#include <iostream>
#include <new>
using namespace std;
#define BIG_NUMBER 10000000000LL
int main() {
   try {
      int *pI = new int[BIG_NUMBER];
   }
   catch (bad_alloc& ex) {
      cout << "Caught bad_alloc: " << ex.what() << endl;
      return -1;
   }
}
```

使用 `nothrow` 格式的 `new` 时，可以测试分配失败，如以下示例所示：

C++

```
#include <iostream>
#include <new>
using namespace std;
#define BIG_NUMBER 10000000000LL
int main() {
   int *pI = new(nothrow) int[BIG_NUMBER];
   if ( pI == nullptr ) {
      cout << "Insufficient memory" << endl;
      return -1;
   }
}
```

使用 `nothrownew.obj` 文件替换全局 `operator new` 时，可以测试失败的内存分配，如下所示：

C++

```
#include <iostream>
#include <new>
using namespace std;
#define BIG_NUMBER 10000000000LL
int main() {
   int *pI = new int[BIG_NUMBER];
   if ( !pI ) {
      cout << "Insufficient memory" << endl;
      return -1;
   }
}
```

可以为失败的内存分配请求提供处理程序。 可以编写自定义恢复例程来处理此类失败。 例如，它可以释放一些保留内存，然后允许分配再次运行。 有关详细信息，请参阅 _set_new_handler。

`delete` 运算符

可使用 `new` 运算符释放使用 `delete` 运算符动态分配的内存。 delete 运算符调用 `operator delete` 函数，该函数将内存释放回可用池。 使用 `delete` 运算符也会导致调用类析构函数（如果存在）。

存在全局和类范围的 `operator delete` 函数。 只能为给定类定义一个 `operator delete` 函数；如果定义了该函数，它会隐藏全局 `operator delete` 函数。 始终为所有类型的数组调用全局 `operator delete` 函数。

全局 `operator delete` 函数。 全局 `operator delete` 函数和类成员 `operator delete` 函数存在两种形式：

C++

```
void operator delete( void * );
void operator delete( void *, size_t );
```

给定类中只存在前面两种形式中的一个。 第一种形式采用 `void *` 类型的单个自变量，其中包含指向要解除分配的对象的指针。 第二个形式（大小经过调整的解除分配）采用两个自变量，第一个是指向要解除分配的内存块的指针，第二个是解除分配的字节数。 这两种形式的返回类型为 `void` （`operator delete` 无法返回值）。

第二种形式的意图是加快搜索要删除的对象的正确大小类别。 此信息通常不会存储在分配本身附近，并且可能未缓存。 当基类中的 `operator delete` 函数用于删除派生类的对象时，第二个形式非常有用。

`operator delete` 函数是静态的，因此它不能是虚拟的。 `operator delete` 函数服从访问控制，如成员访问控制中所述。

以下示例显示了旨在记录内存的分配和解除分配的用户定义的 `operator new` 和 `operator delete` 函数：

C++

```
#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;
}
```

前面的代码可用于检测“内存溢出”，即在自由储存中分配但从未释放过的内存。 若要检测泄漏，应重新定义全局 `new` 和 `delete` 运算符以计算内存的分配和解除分配。

编译器在类声明中支持成员数组 `new` 和 `delete` 运算符。 例如：

C++

```
// 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;
}
```

Last updated on 2025/11/07

智能指针（现代 C++）

在现代 C++ 编程中，标准库包含智能指针，该指针用于确保程序不存在内存和资源泄漏且是异常安全的。

智能指针的使用

智能指针是在 `std` 头文件中的 < 命名空间中定义的。 它们对 RAII 或“获取资源即初始化”编程惯用法至关重要。 此习惯用法的主要目的是确保资源获取与对象初始化同时发生，从而能够创建该对象的所有资源并在某行代码中准备就绪。 实际上，RAII 的主要原则是为将任何堆分配资源（例如，动态分配内存或系统对象句柄）的所有权提供给其析构函数包含用于删除或释放资源的代码以及任何相关清理代码的堆栈分配对象。

大多数情况下，当初始化原始指针或资源句柄以指向实际资源时，会立即将指针传递给智能指针。 在现代 C++ 中，原始指针仅用于范围有限的小代码块、循环或者性能至关重要且不会混淆所有权的 Helper 函数中。

下面的示例将原始指针声明与智能指针声明进行了比较。

C++

```
void UseRawPointer()
{
    // Using a raw pointer -- not recommended.
    Song* pSong = new Song(L"Nothing on You", L"Bruno Mars");

    // Use pSong...

    // Don't forget to delete!
    delete pSong;
}

void UseSmartPointer()
{
    // Declare a smart pointer on stack and pass it the raw pointer.
    unique_ptr<Song> song2(new Song(L"Nothing on You", L"Bruno Mars"));

    // Use song2...
    wstring s = song2->duration_;
    //...

} // song2 is deleted automatically here.
```

如示例所示，智能指针是你在堆栈上声明的类模板，并可通过使用指向某个堆分配的对象的原始指针进行初始化。 在初始化智能指针后，它将拥有原始的指针。 这意味着智能指针负责删除原始指针指定的内存。 智能指针析构函数包括要删除的调用，并且由于在堆栈上声明了智能指针，当智能指针超出范围时将调用其析构函数，尽管堆栈上的某处将进一步引发异常。

通过使用熟悉的指针运算符（`->` 和 `*`）访问封装指针，智能指针类将重载这些运算符以返回封装的原始指针。

C++ 智能指针思路类似于在语言（如 C#）中创建对象的过程：创建对象后让系统负责在正确的时间将其删除。 不同之处在于，单独的垃圾回收器不在后台运行；按照标准 C++ 范围规则对内存进行管理，以使运行时环境更快速更有效。

> **ⓘ 重要**
>
> 请始终在单独的代码行上创建智能指针，而绝不在参数列表中创建智能指针，这样就不会由于某些参数列表分配规则而发生轻微泄露资源的情况。

下面的示例演示了如何使用 C++ 标准库中的 `unique_ptr` 智能指针类型将指针封装到大型对象。

C++

```
class LargeObject
{
public:
    void DoSomething(){}
};

void ProcessLargeObject(const LargeObject& lo){}
void SmartPointerDemo()
{
    // Create the object and pass it to a smart pointer
    std::unique_ptr<LargeObject> pLarge(new LargeObject());

    //Call a method on the object
    pLarge->DoSomething();

    // Pass a reference to a method.
    ProcessLargeObject(*pLarge);

} //pLarge is deleted automatically when function block goes out of scope.
```

此示例演示如何使用智能指针执行以下关键步骤。

1. 将智能指针声明为一个自动（局部）变量。 （不要对智能指针本身使用 `new` 或 `malloc` 表达式。）
2. 在类型参数中，指定封装指针的指向类型。
3. 在智能指针构造函数中将原始指针传递至 `new` 对象。 （某些实用工具函数或智能指针构造函数可为你执行此操作。）

4. 使用重载的 `->` 和 `*` 运算符访问对象。

5. 允许智能指针删除对象。

智能指针的设计原则是在内存和性能上尽可能高效。 例如，`unique_ptr` 中的唯一数据成员是封装的指针。 这意味着，`unique_ptr` 与该指针的大小完全相同，不是四个字节就是八个字节。 使用重载了 * 和 -> 运算符的智能指针访问封装指针的速度不会明显慢于直接访问原始指针的速度。

智能指针具有通过使用"点"表示法访问的成员函数。 例如，一些 C++ 标准库智能指针具有释放指针所有权的重置成员函数。 如果你想要在智能指针超出范围之前释放其内存将很有用，这会很有用，如以下示例所示：

C++

```
void SmartPointerDemo2()
{
    // Create the object and pass it to a smart pointer
    std::unique_ptr<LargeObject> pLarge(new LargeObject());

    //Call a method on the object
    pLarge->DoSomething();

    // Free the memory before we exit function block.
    pLarge.reset();

    // Do some other work...

}
```

智能指针通常提供直接访问其原始指针的方法。 C++ 标准库智能指针拥有一个用于此目的的 `get` 成员函数，`CComPtr` 拥有一个公共的 `p` 类成员。 通过提供对基础指针的直接访问，你可以使用智能指针管理你自己的代码中的内存，还能将原始指针传递给不支持智能指针的代码。

C++

```
void SmartPointerDemo4()
{
    // Create the object and pass it to a smart pointer
    std::unique_ptr<LargeObject> pLarge(new LargeObject());

    //Call a method on the object
    pLarge->DoSomething();

    // Pass raw pointer to a legacy API
    LegacyLargeObjectFunction(pLarge.get());
}
```

智能指针的类型

下一节总结了 Windows 编程环境中可用的不同类型的智能指针，并说明了何时使用它们。

C++ 标准库智能指针

使用这些智能指针作为将指针封装为纯旧 C++ 对象 (POCO) 的首选项。

- `unique_ptr`
  只允许基础指针的一个所有者。 除非你确信需要 `shared_ptr`，否则请将该指针用作 POCO 的默认选项。 可以移到新所有者，但不会复制或共享。 替换已弃用的 `auto_ptr`。 与 `boost::scoped_ptr` 比较。 `unique_ptr` 小巧高效；大小等同于一个指针且支持 rvalue 引用，从而可实现快速插入和对 C++ 标准库集合的检索。 头文件：`<memory>`。 有关详细信息，请参阅如何：创建和使用 unique_ptr 实例和 unique_ptr 类。

- `shared_ptr`
  采用引用计数的智能指针。 如果你想要将一个原始指针分配给多个所有者（例如，从容器返回了指针副本又想保留原始指针时），请使用该指针。 直至所有 `shared_ptr` 所有者超出了范围或放弃所有权，才会删除原始指针。 大小为两个指针；一个用于对象，另一个用于包含引用计数的共享控制块。 头文件：`<memory>`。 有关详细信息，请参阅如何：创建和使用 shared_ptr 实例和 shared_ptr 类。

- `weak_ptr`
  结合 `shared_ptr` 使用的特例智能指针。 `weak_ptr` 提供对一个或多个 `shared_ptr` 实例拥有的对象的访问，但不参与引用计数。 如果你想要观察某个对象但不需要其保持活动状态，请使用该实例。 在某些情况下，需要断开 `shared_ptr` 实例间的循环引用。 头文件：`<memory>`。 有关详细信息，请参阅如何：创建和使用 weak_ptr 实例和 weak_ptr 类。

COM 对象的智能指针（经典 Windows 编程）

当你使用 COM 对象时，请将接口指针包装到适当的智能指针类型中。 活动模板库 (ATL) 针对各种目的定义了多种智能指针。 你还可以使用 `_com_ptr_t` 智能指针类型，编译器在从 .tlb 文件创建包装器类时会使用该类型。 无需包含 ATL 标头文件时，它是最好的选择。

CComPtr 类

除非你无法使用 ATL，否则使用此类型。 使用 `AddRef` 和 `Release` 方法执行引用计数。 有关更多信息，请参阅如何：创建和使用 CComPtr 和 CComQIPtr 实例。

CComQIPtr 类

类似于 `CComPtr`，但还提供了用于在 COM 对象上调用 `QueryInterface` 的简化语法。 有关更多信息，请参阅如何：创建和使用 CComPtr 和 CComQIPtr 实例。

CComHeapPtr 类
指向使用 `CoTaskMemFree` 释放内存的对象的智能指针。

CComGITPtr 类
从全局接口表 (GIT) 获取的接口的智能指针。

_com_ptr_t 类
在功能上类似于 `CComQIPtr`，但不依赖于 ATL 标头。

POCO 对象的 ATL 智能指针

除 COM 对象的智能指针外，ATL 还为纯旧 C++ 对象 (POCO) 定义了智能指针和智能指针集合。在经典 Windows 编程中，这些类型可用于替代 C++ 标准库集合，尤其是不要求代码可移植性或不需要混合 C++ 标准库和 ATL 的编程模型时。

CAutoPtr 类
通过转移副本所有权增强唯一所有权的智能指针。 等同于已弃用的 `std::auto_ptr` 类。

CHeapPtr 类
使用 C malloc 函数分配的对象的智能指针。

CAutoVectorPtr 类
使用 `new[]` 分配的数组的智能指针。

CAutoPtrArray 类
封装一个 `CAutoPtr` 元素数组的类。

CAutoPtrList 类
封装用于操作 `CAutoPtr` 节点列表的方法的类。

另请参阅

指针
C++ 语言参考
C++ 标准库

Last updated on 2025/11/07

如何：创建和使用 unique_ptr 实例

unique_ptr 不共享它的指针。 它无法复制到其他 `unique_ptr`，无法通过值传递到函数，也无法用于需要副本的任何 C++ 标准库算法。 只能移动 `unique_ptr`。 这意味着，内存资源所有权将转移到另一 `unique_ptr`，并且原始 `unique_ptr` 不再拥有此资源。 我们建议你将对象限制为由一个所有者所有，因为多个所有权会使程序逻辑变得复杂。 因此，当需要智能指针用于纯 C++ 对象时，可使用 `unique_ptr`，而当构造 `unique_ptr` 时，可使用 make_unique Helper 函数。

下图演示了两个 `unique_ptr` 实例之间的所有权转换。

`unique_ptr` 在 C++ 标准库的 `<memory>` 标头中定义。 它与原始指针一样高效，可在 C++ 标准库容器中使用。 将 `unique_ptr` 实例添加到 C++ 标准库容器很有效，因为通过 `unique_ptr` 的移动构造函数，不再需要进行复制操作。

示例 1

以下示例演示如何创建 `unique_ptr` 实例并在函数之间传递这些实例。

C++

```
unique_ptr<Song> SongFactory(const std::wstring& artist, const std::wstring& title)
{
    // Implicit move operation into the variable that stores the result.
    return make_unique<Song>(artist, title);
}

void MakeSongs()
{
    // Create a new unique_ptr with a new object.
    auto song = make_unique<Song>(L"Mr. Children", L"Namonaki Uta");

    // Use the unique_ptr.
    vector<wstring> titles = { song->title };

    // Move raw pointer from one unique_ptr to another.
    unique_ptr<Song> song2 = std::move(song);
```

```
    // Obtain unique_ptr from function that returns by value.
    auto song3 = SongFactory(L"Michael Jackson", L"Beat It");
}
```

这些示例说明了 `unique_ptr` 的基本特征：可移动，但不可复制。 “移动”将所有权转移到新 `unique_ptr` 并重置旧 `unique_ptr`。

示例 2

以下示例演示如何创建 `unique_ptr` 实例并在向量中使用这些实例。

C++

```
void SongVector()
{
    vector<unique_ptr<Song>> songs;

    // Create a few new unique_ptr<Song> instances
    // and add them to vector using implicit move semantics.
    songs.push_back(make_unique<Song>(L"B'z", L"Juice"));
    songs.push_back(make_unique<Song>(L"Namie Amuro", L"Funky Town"));
    songs.push_back(make_unique<Song>(L"Kome Kome Club", L"Kimi ga Iru Dake de"));
    songs.push_back(make_unique<Song>(L"Ayumi Hamasaki", L"Poker Face"));

    // Pass by const reference when possible to avoid copying.
    for (const auto& song : songs)
    {
        wcout << L"Artist: " << song->artist << L"   Title: " << song->title <<
endl;
    }
}
```

在 range for 循环中，注意 `unique_ptr` 通过引用来传递。 如果你尝试通过此处的值传递，由于删除了 `unique_ptr` 复制构造函数，编译器将引发错误。

示例 3

以下示例演示如何初始化类成员 `unique_ptr`。

C++

```
class MyClass
{
private:
    // MyClass owns the unique_ptr.
    unique_ptr<ClassFactory> factory;
public:
```

```
    // Initialize by using make_unique with ClassFactory default constructor.
    MyClass() : factory (make_unique<ClassFactory>())
    {
    }

    void MakeClass()
    {
        factory->DoSomething();
    }
};
```

示例 4

可使用 make_unique 将 `unique_ptr` 创建到数组，但无法使用 `make_unique` 初始化数组元素。

C++

```
// Create a unique_ptr to an array of 5 integers.
auto p = make_unique<int[]>(5);

// Initialize the array.
for (int i = 0; i < 5; ++i)
{
    p[i] = i;
    wcout << p[i] << endl;
}
```

有关更多示例，请参阅 make_unique。

另请参阅

智能指针（现代 C++）
make_unique

Last updated on 2025/11/07

如何：创建和使用 shared_ptr 实例

`shared_ptr` 类型是 C++ 标准库中的一种智能指针，专为多个所有者需要管理对象生命周期的方案而设计。 在您初始化一个 `shared_ptr` 之后，您可复制它，按值将其传入函数参数，然后将其分配给其他 `shared_ptr` 实例。 所有实例均指向同一个对象，并共享对一个“控制块”（每当新的 `shared_ptr` 添加、超出范围或重置时增加和减少引用计数）的访问权限。 当引用计数达到零时，控制块将删除内存资源和自身。

下图显示了指向一个内存位置的几个 `shared_ptr` 实例。

示例设置

以下示例均假设你已经包含所需标头并声明所需类型，如下所示：

C++

```
// 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
}
```

示例 1

尽可能在首次创建内存资源时使用 make_shared 函数来创建 `shared_ptr`。 `make_shared` 异常安全。 它使用同一调用为控制块和资源分配内存，这会减少构造开销。 如果不使用 `make_shared`，则必须先使用显式 **`new`** 表达式来创建对象，然后才能将其传递到 `shared_ptr` 构造函数。 以下示例演示了同时声明和初始化 `shared_ptr` 和新对象的各种方式。

C++

```
// Use make_shared function when possible.
auto sp1 = make_shared<Song>(L"The Beatles", L"Im Happy Just to Dance With You");

// Ok, but slightly less efficient.
// Note: Using new expression as constructor argument
// creates no named variable for other code to access.
shared_ptr<Song> sp2(new Song(L"Lady Gaga", L"Just Dance"));

// When initialization must be separate from declaration, e.g. class members,
// initialize with nullptr to make your programming intent explicit.
shared_ptr<Song> sp5(nullptr);
//Equivalent to: shared_ptr<Song> sp5;
//...
sp5 = make_shared<Song>(L"Elton John", L"I'm Still Standing");
```

示例 2

以下示例演示如何声明和初始化对由其他 `shared_ptr` 分配的对象具有共享所有权的 `shared_ptr` 实例。 假设 `sp2` 是已初始化的 `shared_ptr`。

C++

```
//Initialize with copy constructor. Increments ref count.
auto sp3(sp2);

//Initialize via assignment. Increments ref count.
auto sp4 = sp2;

//Initialize with nullptr. sp7 is empty.
shared_ptr<Song> sp7(nullptr);

// Initialize with another shared_ptr. sp1 and sp2
// swap pointers as well as ref counts.
sp1.swap(sp2);
```

示例 3

在你使用复制元素的算法时， `shared_ptr` 在 C++ 标准库容器中很有用。 您可以将元素包装在 `shared_ptr` 中，然后将其复制到其他容器中（请记住，只要您需要，基础内存就会一直有效）。 以下示例演示如何在向量中对 `remove_copy_if` 实例使用 `shared_ptr` 算法。

C++

```
vector<shared_ptr<Song>> v {
  make_shared<Song>(L"Bob Dylan", L"The Times They Are A Changing"),
  make_shared<Song>(L"Aretha Franklin", L"Bridge Over Troubled Water"),
  make_shared<Song>(L"Thalía", L"Entre El Mar y Una Estrella")
};

vector<shared_ptr<Song>> v2;
remove_copy_if(v.begin(), v.end(), back_inserter(v2), [] (shared_ptr<Song> s)
{
    return s->artist.compare(L"Bob Dylan") == 0;
});

for (const auto& s : v2)
{
    wcout << s->artist << L":" << s->title << endl;
}
```

示例 4

您可以使用 `dynamic_pointer_cast`、 `static_pointer_cast` 和 `const_pointer_cast` 来转换 `shared_ptr`。 这些函数类似于 **`dynamic_cast`**、 **`static_cast`** 和 **`const_cast`** 运算符。 以下示例

演示如何测试基类的 `shared_ptr` 向量中每个元素的派生类型，然后复制元素并显示有关它们的信息。

C++

```
vector<shared_ptr<MediaAsset>> assets {
  make_shared<Song>(L"Himesh Reshammiya", L"Tera Surroor"),
  make_shared<Song>(L"Penaz Masani", L"Tu Dil De De"),
  make_shared<Photo>(L"2011-04-06", L"Redmond, WA", L"Soccer field at Microsoft.")
};

vector<shared_ptr<MediaAsset>> photos;

copy_if(assets.begin(), assets.end(), back_inserter(photos), []
(shared_ptr<MediaAsset> p) -> bool
{
    // Use dynamic_pointer_cast to test whether
    // element is a shared_ptr<Photo>.
    shared_ptr<Photo> temp = dynamic_pointer_cast<Photo>(p);
    return temp.get() != nullptr;
});

for (const auto&  p : photos)
{
    // We know that the photos vector contains only
    // shared_ptr<Photo> objects, so use static_cast.
    wcout << "Photo location: " << (static_pointer_cast<Photo>(p))->location <<
endl;
}
```

示例 5

您可以通过下列方式将 `shared_ptr` 传递给其他函数：

- 按值传递 `shared_ptr`。 这将调用复制构造函数，增加引用计数，并使被调用方成为所有者。 此操作的开销很小，但此操作的开销可能很大，具体取决于要传递多少 `shared_ptr` 对象。 当调用方和被调用方之间的（隐式或显式）代码协定要求被调用方是所有者时，请使用此选项。

- 按引用或常量引用传递 `shared_ptr`。 在这种情况下，引用计数不会增加，并且只要调用方不超出范围，被调用方就可以访问指针。 或者，被调用方可以决定基于引用创建一个 `shared_ptr`，并成为一个共享所有者。 当调用方并不知道被调用方，或当您由于性能原因必须传递一个 `shared_ptr` 且希望避免复制操作时，请使用此选项。

- 传递基础指针或对基础对象的引用。 这使被调用方能够使用该对象，但它不会与调用方的 `shared_ptr` 共享对象的所有权。 请注意被调用方从传递的原始指针中创建 `shared_ptr` 的情况，因为被调用方的 `shared_ptr` 拥有一个独立于调用方 `shared_ptr` 的引用计数。 当被

调用方中的 `shared_ptr` 超出范围时，它将删除对象，使调用方的'shared_ptr'中的指针指向释放的内存。 当调用方 `shared_ptr` 超出范围时，将产生 double-free 结果。 仅当调用方和被调用方之间的协定明确指定调用方保留 `shared_ptr` 生存期的所有权时，才使用此选项。

- 在确定如何传递 `shared_ptr` 时，确定被调用方是否必须共享基础资源的所有权。 “所有者”是只要它需要就可以使基础资源一直有效的对象或函数。 如果调用方必须保证被调用方可以将指针的生命延长到其（函数）生存期以外，则请使用第一个选项。 如果您不关心被调用方是否延长生存期，则按引用传递并让被调用方复制或不复制它。

- 如果必须为帮助程序函数提供对基础指针的访问权限，并且你知道帮助程序函数将使用该指针并且将在调用函数返回前返回，则该函数不必共享基础指针的所有权。 它只需在调用方的 `shared_ptr` 的生存期内访问指针即可。 在这种情况下，按引用传递 `shared_ptr` 或传递原始指针或对基础对象的引用是安全的。 通过此方式传递将提供一个小的性能好处，并且还有助于您表达编程意图。

- 有时，例如，在一个 `std::vector<shared_ptr<T>>` 中，您可能必须将每个 `shared_ptr` 传递给 lambda 表达式主体或命名函数对象。 如果 lambda 或函数没有存储指针，则将按引用传递 `shared_ptr` 以避免调用每个元素的复制构造函数。

C++

```
void use_shared_ptr_by_value(shared_ptr<int> sp);

void use_shared_ptr_by_reference(shared_ptr<int>& sp);
void use_shared_ptr_by_const_reference(const shared_ptr<int>& sp);

void use_raw_pointer(int* p);
void use_reference(int& r);

void test() {
    auto sp = make_shared<int>(5);

    // Pass the shared_ptr by value.
    // This invokes the copy constructor, increments the reference count, and makes
the callee an owner.
    use_shared_ptr_by_value(sp);

    // Pass the shared_ptr by reference or const reference.
    // In this case, the reference count isn't incremented.
    use_shared_ptr_by_reference(sp);
    use_shared_ptr_by_const_reference(sp);

    // Pass the underlying pointer or a reference to the underlying object.
    use_raw_pointer(sp.get());
    use_reference(*sp);

    // Pass the shared_ptr by value.
```

```
    // This invokes the move constructor, which doesn't increment the reference
count
    // but in fact transfers ownership to the callee.
    use_shared_ptr_by_value(move(sp));
}
```

示例 6

以下示例演示 `shared_ptr` 如何重载各种比较运算符以在 `shared_ptr` 实例所有的内存上实现指针比较。

C++

```
// Initialize two separate raw pointers.
// Note that they contain the same values.
auto song1 = new Song(L"Village People", L"YMCA");
auto song2 = new Song(L"Village People", L"YMCA");

// Create two unrelated shared_ptrs.
shared_ptr<Song> p1(song1);
shared_ptr<Song> p2(song2);

// Unrelated shared_ptrs are never equal.
wcout << "p1 < p2 = " << std::boolalpha << (p1 < p2) << endl;
wcout << "p1 == p2 = " << std::boolalpha <<(p1 == p2) << endl;

// Related shared_ptr instances are always equal.
shared_ptr<Song> p3(p2);
wcout << "p3 == p2 = " << std::boolalpha << (p3 == p2) << endl;
```

另请参阅

智能指针（现代 C++）

Last updated on 2025/11/07

如何：创建和使用 weak_ptr 实例

有时，对象必须存储一种方法来访问 shared_ptr 的基础对象，而不会导致引用计数递增。 通常，在 `shared_ptr` 实例之间有循环引用时，会出现这种情况。

最佳设计是尽量避免指针的共享所有权。 但是，如果必须拥有 `shared_ptr` 实例的共享所有权，请避免它们之间的循环引用。 如果循环引用不可避免，甚至出于某种原因甚至更可取，请使用 weak_ptr 为一个或多个所有者提供对另一个 `shared_ptr` 所有者的弱引用。 通过使用 `weak_ptr`，可以创建一个联接到现有相关实例集的 `shared_ptr`，但前提是基础内存资源仍然有效。 `weak_ptr` 本身不参与引用计数，因此，它无法阻止引用计数变为零。 但是，可以使用 `weak_ptr` 尝试获取初始化该副本的 `shared_ptr` 的新副本。 若已删除内存，则 `weak_ptr` 的 bool 运算符返回 **`false`**。 若内存仍然有效，则新的共享指针会递增引用计数，并保证只要 `shared_ptr` 变量保留在作用域内，内存就会有效。

示例

下面的代码示例演示的案例是将 `weak_ptr` 用于确保正确删除具有循环依赖项的对象。 在检查此示例时，假设仅在考虑替代解决方案后创建该示例。 这些 `Controller` 对象表示计算机进程的一些方面，它们独立运行。 每个控制器必须随时能够查询其他控制器的状态，每个控制器都包含一个专用 `vector<weak_ptr<Controller>>`，用于实现此目的。 每个向量都包含一个循环引用，因此，使用 `weak_ptr` 实例而不是 `shared_ptr`。

C++

```
#include <iostream>
#include <memory>
#include <string>
#include <vector>
#include <algorithm>

using namespace std;

class Controller
{
public:
   int Num;
   wstring Status;
   vector<weak_ptr<Controller>> others;
   explicit Controller(int i) : Num(i), Status(L"On")
   {
      wcout << L"Creating Controller" << Num << endl;
   }

   ~Controller()
   {
      wcout << L"Destroying Controller" << Num << endl;
```

```
   }

   // Demonstrates how to test whether the
   // pointed-to memory still exists or not.
   void CheckStatuses() const
   {
      for_each(others.begin(), others.end(), [](weak_ptr<Controller> wp) {
         auto p = wp.lock();
         if (p)
         {
            wcout << L"Status of " << p->Num << " = " << p->Status << endl;
         }
         else
         {
            wcout << L"Null object" << endl;
         }
      });
   }
};

void RunTest()
{
   vector<shared_ptr<Controller>> v{
       make_shared<Controller>(0),
       make_shared<Controller>(1),
       make_shared<Controller>(2),
       make_shared<Controller>(3),
       make_shared<Controller>(4),
   };

   // Each controller depends on all others not being deleted.
   // Give each controller a pointer to all the others.
   for (int i = 0; i < v.size(); ++i)
   {
      for_each(v.begin(), v.end(), [&v, i](shared_ptr<Controller> p) {
         if (p->Num != i)
         {
            v[i]->others.push_back(weak_ptr<Controller>(p));
            wcout << L"push_back to v[" << i << "]: " << p->Num << endl;
         }
      });
   }

   for_each(v.begin(), v.end(), [](shared_ptr<Controller> &p) {
      wcout << L"use_count = " << p.use_count() << endl;
      p->CheckStatuses();
   });
}

int main()
{
   RunTest();
   wcout << L"Press any key" << endl;
   char ch;
```

```
    cin.getline(&ch, 1);
}
```

Output

```
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
```

作为试验，请将向量 `others` 修改为一个 `vector<shared_ptr<Controller>>`，然后在输出中，请注意返回时 `RunTest` 不会调用析构函数。

另请参阅

智能指针（现代 C++）

Last updated on 2025/11/07

如何：创建和使用 CComPtr 和 CComQIPtr 实例

在经典 Windows 编程中，库通常作为 COM 对象（更准确地说是 COM 服务器）实现。 很多 Windows 操作系统组件都作为 COM 服务器实现，因此，很多参与者以这种形式提供库。 有关 COM 的基础知识的信息，请参阅 Component Object Model (COM)。

在实例化组件对象模型 (COM) 对象时，请将接口指针存储在 COM 智能指针中，后者将使用 `AddRef` 和 `Release` 执行引用计数。 如果您使用了活动模板库 (ATL) 或 Microsoft 基础类库 (MFC)，则使用 `CComPtr` 智能指针。 如果没有使用 ATL 或 MFC，则使用 `_com_ptr_t`。 由于没有等效于 `std::unique_ptr`的 COM，请对单个所有者和多个所有者的情况都使用这些智能指针。 `CComPtr` 和 `ComQIPtr` 都支持具有 rvalue 引用的移动操作。

示例：CComPtr

以下示例演示如何使用 `CComPtr` 实例化 COM 对象并获取指向其接口的指针。 请注意，`CComPtr::CoCreateInstance` 成员函数用于创建 COM 对象，而不是同名的 Win32 函数。

C++

```
void CComPtrDemo()
{
    HRESULT hr = CoInitialize(NULL);

    // Declare the smart pointer.
    CComPtr<IGraphBuilder> pGraph;

    // Use its member function CoCreateInstance to
    // create the COM object and obtain the IGraphBuilder pointer.
    hr = pGraph.CoCreateInstance(CLSID_FilterGraph);
    if(FAILED(hr)){ /*... handle hr error*/ }

    // Use the overloaded -> operator to call the interface methods.
    hr = pGraph->RenderFile(L"C:\\Users\\Public\\Music\\Sample Music\\Sleep
Away.mp3", NULL);
    if(FAILED(hr)){ /*... handle hr error*/ }

    // Declare a second smart pointer and use it to
    // obtain another interface from the object.
    CComPtr<IMediaControl> pControl;
    hr = pGraph->QueryInterface(IID_PPV_ARGS(&pControl));
    if(FAILED(hr)){ /*... handle hr error*/ }

    // Obtain a third interface.
    CComPtr<IMediaEvent> pEvent;
    hr = pGraph->QueryInterface(IID_PPV_ARGS(&pEvent));
    if(FAILED(hr)){ /*... handle hr error*/ }
```

```
    // Use the second interface.
    hr = pControl->Run();
    if(FAILED(hr)){ /*... handle hr error*/ }

    // Use the third interface.
    long evCode = 0;
    pEvent->WaitForCompletion(INFINITE, &evCode);

    CoUninitialize();

    // Let the smart pointers do all reference counting.
}
```

`CComPtr` 及其相关内容是 ATL 的一部分，在 <atlcomcli.h> 中定义。 `_com_ptr_t` 在 <comip.h> 中声明。 当为类型库生成包装器类时，编译器将创建 `_com_ptr_t` 的专用化。

示例：CComQIPtr

ATL 还提供了 `CComQIPtr`，它具有用于查询 COM 对象以检索额外接口的更简单的语法。 但是，建议采用 `CComPtr` ，因为它能够执行 `CComQIPtr` 执行的所有操作，并且在语义上与原始 COM 接口指针更一致。 如果您可以使用 `CComPtr` 来查询接口，新接口指针将被放在输出参数中。 如果调用失败，则会返回 HRESULT，它是典型的 COM 模式。 如果使用 `CComQIPtr`，返回值将为指针本身，并且当调用失败时，内部 HRESULT 返回值将无法访问。 以下两行显示了 `CComPtr` 和 `CComQIPtr` 中的错误处理机制的差异。

C++

```
// CComPtr with error handling:
CComPtr<IMediaControl> pControl;
hr = pGraph->QueryInterface(IID_PPV_ARGS(&pControl));
if(FAILED(hr)){ /*... handle hr error*/ }

// CComQIPtr with error handling
CComQIPtr<IMediaEvent> pEvent = pControl;
if(!pEvent){ /*... handle NULL pointer error*/ }

// Use the second interface.
hr = pControl->Run();
if(FAILED(hr)){ /*... handle hr error*/ }
```

示例：IDispatch

`CComPtr` 提供了针对 IDispatch 的专用化，使它能够存储指向 COM 自动化组件的指针并使用后期绑定调用接口方法。 `CComDispatchDriver` 是 `CComQIPtr<IDispatch, &IIDIDispatch>` （可隐式

转换为 `CComPtr<IDispatch>`）的 typedef。 因此，当这三个名称中的任何一个出现在代码中时，它都与 `CComPtr<IDispatch>`等效。 以下示例演示如何使用 `CComPtr<IDispatch>`获取指向 Microsoft Word 对象模型的指针。

C++

```
void COMAutomationSmartPointerDemo()
{
    CComPtr<IDispatch> pWord;
    CComQIPtr<IDispatch, &IID_IDispatch> pqi = pWord;
    CComDispatchDriver pDriver = pqi;

    HRESULT hr;
    _variant_t pOutVal;

    CoInitialize(NULL);
    hr = pWord.CoCreateInstance(L"Word.Application", NULL, CLSCTX_LOCAL_SERVER);
    if(FAILED(hr)){ /*... handle hr error*/ }

    // Make Word visible.
    hr = pWord.PutPropertyByName(_bstr_t("Visible"),  &_variant_t(1));
    if(FAILED(hr)){ /*... handle hr error*/ }

    // Get the Documents collection and store it in new CComPtr
    hr = pWord.GetPropertyByName(_bstr_t("Documents"), &pOutVal);
    if(FAILED(hr)){ /*... handle hr error*/ }

    CComPtr<IDispatch> pDocuments = pOutVal.pdispVal;

    // Use Documents to open a document
    hr = pDocuments.Invoke1 (_bstr_t("Open"),
&_variant_t("c:\\users\\public\\documents\\sometext.txt"),&pOutVal);
    if(FAILED(hr)){ /*... handle hr error*/ }

    CoUninitialize();
}
```

另请参阅

智能指针（现代 C++）

Last updated on 2025/11/19

MSVC 中的异常处理

异常是一个可能超出程序的控制范围的错误条件，它会阻止程序继续沿其常规执行路径执行。 某些操作（包括对象创建、文件输入/输出以及从其他模块中进行的函数调用）都可能是异常的来源，即便程序正常运行也是如此。 可靠代码可预见并处理异常。 若要检测逻辑错误，请使用断言而不是异常（请参阅使用断言）。

异常类型

Microsoft C++ 编译器 (MSVC) 支持三类异常处理：

- C++ 异常处理

  对于大多数 C++ 程序，应使用 C++ 异常处理。 它是类型安全的，可确保在展开堆栈时调用析构函数。

- 结构化异常处理

  Windows 提供自己的异常机制，称为结构化异常处理 (SEH)。 建议不要将该机制用于 C++ 或 MFC 编程。 仅在非 MFC C 程序中使用 SEH。

- MFC 异常

  自版本 3.0 起，MFC 已使用 C++ 异常。 它仍支持其较早的异常处理宏，这些宏在形式上与 C++ 异常类似。 有关将 MFC 宏和 C++ 异常组合的建议，请参阅异常：使用 MFC 宏和 C++ 异常。

使用 /EH 编译器选项来指定要在 C++ 项目中使用的异常处理模型。 标准 C++ 异常处理 (**/EHsc**) 是 Visual Studio 中新 C++ 项目中的默认值。

不建议混合使用异常处理机制。 例如，不要将结构化异常处理用于 C++ 异常。 专门使用 C++ 异常处理可以使代码的移植性更强，并且它使你可以处理任何类型的异常。 有关结构化异常处理的缺点的详细信息，请参阅结构化异常处理。

本节内容

- 现代 C++ 处理异常和错误的最佳做法
- 如何设计以实现异常安全性
- 如何连接异常和非异常代码
- try、catch 和 throw 语句

- Catch 块如何计算
- 异常和堆栈展开
- 异常规范
- noexcept
- 未处理的 C++ 异常
- 混合使用 C（结构化）和 C++ 异常
- 结构化异常处理 (SEH) (C/C++)

另请参阅

C++ 语言参考
x64 异常处理
异常处理 (C++/CLI 和 C++/CX)

Last updated on 2025/11/07

现代 C++ 处理异常和错误的最佳做法

在新式 C++ 中，在大多数情况下，报告和处理逻辑错误与运行时错误的首选方式是使用异常。 当堆栈可能在检测错误的函数与具有错误处理上下文的函数之间包含多个函数调用时，这种方式尤其有用。 异常为检测错误的代码提供正式的、妥善定义的方式，以将信息向上传递到调用堆栈。

对异常代码使用异常

程序错误通常分为两类：

- 编程错误导致的逻辑错误。 例如，“索引超出范围”错误。
- 超出程序员控制的运行时错误。 例如，“网络服务不可用”错误。

在 C 样式的编程和 COM 中，错误报告的管理方式是返回一个表示错误代码或特定函数的状态代码的值，或者设置一个全局变量，调用方可以在每次执行函数调用后选择性地检索该变量来查看是否报告了错误。 例如，COM 编程使用 `HRESULT` 返回值将错误传达给调用方。 Win32 API 提供 `GetLastError` 函数来检索调用堆栈报告的最后一个错误。 在这两种情况下，都需要由调用方识别代码并相应地做出响应。 如果调用方未显式处理错误代码，则程序可能会在不发出警告的情况下崩溃。 或者，它可能会继续使用错误的数据执行，并生成错误的结果。

新式 C++ 中优先使用异常的原因如下：

- 异常会强制调用代码识别并处理错误状态。 未经处理的异常会停止程序执行。
- 异常跳转到调用堆栈中可以处理错误的位置。 中间函数可以让异常传播。 这些函数不必与其他层协调。
- 引发异常后，异常堆栈展开机制将根据妥善定义的规则销毁范围内的所有对象。
- 异常可以在检测错误的代码与处理错误的代码之间实现明确的分离。

以下简化示例演示了 C++ 中引发和捕获异常的必要语法：

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;
}
```

C++ 中的异常类似于 C# 和 Java 等语言中的异常。 在 `try` 块中，如果引发某个异常，类型与该异常的类型匹配的第一个关联 块将捕获该异常`catch`。 换言之，执行将从 `throw` 语句跳转到 `catch` 语句。 如果未找到可用的 catch 块，则调用 `std::terminate` 并且程序会退出。 在 C++ 中可以引发任何类型；但是，我们建议引发直接或间接派生自 `std::exception` 的类型。 在前面的示例中，异常类型 invalid_argument 是在标准库的 <stdexcept> 头文件中定义的。 C++ 既不提供也不需要 `finally` 块来确保在引发异常时释放所有资源。 资源采集是使用智能指针的初始化 (RAII) 习语，它提供所需的功能来清理资源。 有关详细信息，请参阅如何：针对异常安全性进行设计。 有关 C++ 堆栈展开机制的信息，请参阅异常和堆栈展开。

基本准则

在任何编程语言中实现可靠的错误处理都颇有挑战性。 尽管异常提供多项功能来支持妥善的错误处理，但它们不能代你解决一切问题。 为了实现异常机制的优势，请在设计代码时考虑到异常。

- 使用断言检查应始终为 true 或始终为 false 的条件。 使用异常来检查可能发生的错误，例如公共函数参数的输入验证错误。 有关详细信息，请参阅异常与断言部分。
- 当处理错误的代码与通过一个或多个中间函数调用检测错误的代码分离时，请使用异常。 当处理错误的代码与检测错误的代码紧密耦合时，请考虑是否在性能关键型循环中使用错误代码。
- 对于每个可能引发或传播异常的函数，请提供三项异常保证之一：强保证、基本保证或 nothrow (`noexcept`) 保证。 有关详细信息，请参阅如何：针对异常安全性进行设计。
- 通过值引发异常，通过引用捕获异常。 不要捕获无法处理的异常。
- 不要使用 C++11 中已弃用的异常规范。 有关详细信息，请参阅异常规范和 noexcept 部分。
- 使用适用的标准库异常类型。 从 exception 类层次结构派生自定义的异常类型。

- 不要允许异常从析构函数或内存解除分配函数中逃逸。

异常和性能

如果未引发异常，则异常机制的性能开销极低。 如果引发异常，则堆栈遍历和展开的开销与函数调用的开销大致相当。 进入 `try` 块后，需要使用其他数据结构来跟踪调用堆栈，如果引发异常，则还需要使用更多指令来展开堆栈。 但是，在大多数情况下，性能和内存占用的开销并不高。 异常对性能的不利影响可能仅在内存受限的系统上才比较明显。 或者，这种影响在性能关键型循环中（其中的错误可能经常发生，并且处理错误的代码与报告错误的代码之间存在紧密耦合）可能比较明显。 无论在哪种情况下，不进行分析和测量就不可能知道异常的实际开销。 即使在开销很高的极少数情况下，也可将这种开销与设计良好的异常策略所提供的更高正确性、更方便的维护性和其他优势进行权衡。

异常与断言

异常和断言是用于检测程序中运行时错误的两种不同机制。 如果所有代码都正确，可以使用 `assert` 语句来测试开发过程中始终应为 true 或 false 的条件。 使用异常来处理此类错误是没有意义的，因为错误指示的是代码中必须修复的问题。 它并不表示程序在运行时必须从中恢复的状态。 `assert` 在语句中停止执行，以便可以在调试器中检查程序状态。 异常从第一个适当的 catch 处理程序继续执行。 即使代码正确，也可以使用异常来检查在运行时可能发生的错误状态，例如“找不到文件”或“内存不足”。异常可以处理这些状态，即使恢复只是将消息输出到日志并结束程序。 始终使用异常来检查公共函数的参数。 即使函数没有错误，也可能无法完全控制用户传递给它的参数。

C++ 异常与 Windows SEH 异常

C 和 C++ 程序都可以使用 Windows 操作系统中的结构化异常处理 (SEH) 机制。 SEH 中的概念类似于 C++ 异常中的概念，只不过 SEH 使用 `__try`、`__except` 和 `__finally` 构造，而不是 `try` 和 `catch`。 在 Microsoft C++ 编译器 (MSVC) 中，为 SEH 实现了 C++ 异常。 但是，在编写 C++ 代码时，请使用 C++ 异常语法。

有关 SEH 的详细信息，请参阅结构化异常处理 (C/C++)。

异常规范和 `noexcept`

C++ 中引入了异常规范，作为一种用于指定函数可能引发的异常的方式。 但经证实，异常规范会在实践中造成问题，并且在 C++11 草案标准中已遭弃用。 我们建议不要使用除 `throw`（表示函数不允许异常逃逸）以外的 `throw()` 异常规范。 如果必须使用已弃用 `throw( type-name )` 形

式的异常规范，则 MSVC 支持会受限制。 有关详细信息，请参阅异常规范 (throw)。 C++11 中引入了 `noexcept` 说明符作为 `throw()` 的首选替代项。

另请参阅

如何：异常和非异常代码之间的接口
C++ 语言参考
C++ 标准库

Last updated on 2025/11/07

操作说明：异常安全性设计

异常机制的优势之一是执行以及异常相关数据将直接从引发异常的语句跳至处理异常的第一个 catch 语句。 处理程序可以是调用堆栈中任意数量的级别。 在 try 语句和 throw 语句之间调用的函数无需了解与所引发异常有关的任何信息。 但是，这些函数必须进行设计，以便它们在异常可能从下向上传播时“意外地”超出范围，而这样做不会留下部分创建的对象、泄漏的内存或处于不稳定状态的数据结构。

基本技术

可靠的异常处理策略需要细致的思考，并且应该是设计过程的一部分。 一般而言，大部分异常都是在软件模块的低层检测到和引发的，但这些层通常没有足够的上下文来处理错误或将消息显示给最终用户。 在中间层，函数可在必须检测异常对象时，或者它们具有要为最终捕获异常的上层提供的其他有用信息时，捕获和重新引发异常。 函数仅当能够完全恢复异常时才应捕获和“吞并”异常。 在大多数情况下，中间层的正确行为是让异常传播到调用堆栈。 甚至在最高层，如果未经处理的异常使某个程序处于无法保证其正确性的状态，则适当的操作可能是让该异常终止该程序。

无论函数如何处理异常，为了帮助确保函数是“异常安全的”，必须通过下列基本规则设计函数。

保持资源类简单

将手动资源管理封装到类中时，请使用仅管理单个资源的类。 通过保持类简单，可以降低引入资源泄漏的风险。 如有可能，请使用智能指针，如以下示例所示。 对于突出显示使用 `shared_ptr` 时的差异，此示例是特意模拟的，非常简单。

C++

```
// 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) { }
};
```

使用 RAII 习语管理资源

要实现异常安全状态，函数必须确保销毁其使用 `malloc` 或 **`new`** 分配的对象以及关闭或释放所有资源（如文件句柄），即使引发异常时也是如此。 资源获取即初始化 (RAII) 习语使此类资源的管理依赖于自动变量的生命期。 当函数超出范围时，要么正常返回；要么因为异常，调用所有完全构造的自动变量的析构函数。 RAII 包装器对象（如智能指针）将在其析构函数中调用合适的 delete 或 close 函数。 在异常安全的代码中，将每个资源的所有权立即传递给某种 RAII 对象至关重要。 请注意，`vector`、`string`、`make_shared`、`fstream` 和类似的类将为你处理资源获取。 但 `unique_ptr` 和传统的 `shared_ptr` 构造例外，因为资源获取是由用户而不是对象执行的；因此，这些构造将视为资源释放即析构，但作为 RAII 是不可靠的。

三个异常保证

一般来说，异常安全的讨论与一个函数可提供的三个异常保证有关：无故障保证、增强保证和基本保证。

无故障保证

无故障（或“无引发”）保证是一个函数可提供的最有力的保证。 此保证声明，该函数将不会引发异常或允许异常传播。 但是，您无法可靠地提供此类包装，除非 (a) 您知道该函数调用的所有函数也是无故障的，或 (b) 您知道将在引发的所有异常到达该函数之前捕获这些异常，或者 (c) 您知道如何捕获和正确地处理可能到达该函数的所有异常。

增强保证和基本保证均依赖析构函数无故障这一假定。 标准库中的所有容器和类型保证其析构函数不会引发。 还有一个相反的需求：标准库需求为其提供的用户定义的类型，例如，作为模板自变量，必须具有未引发的析构函数。

增强保证

增强保证声明，如果函数因异常超出范围，则将不会泄漏内存并且不会修改程序状态。 一个提供增强保证的函数主要是一个提交或回滚语义的事务：它要么完全成功，要么无任何效果。

基本保证

基本保证是三个保证是最弱的一个。 但是，当增强保证对于内存消耗或性能来说很昂贵时，此保证可能是最佳选择。 基本保证声明，如果出现异常，内存不会泄漏并且对象将仍处于可用状态，即使可能已修改数据仍是如此。

异常安全类

类可帮助确保其自己的异常安全，方式为防止自身进行部分构造或部分销毁，即使它由不安全的函数使用也是如此。 如果类构造函数在完成之前就存在，则绝不会创建对象，并且绝不会调用其析构函数。 虽然在异常之前已初始化的自动变量将调用其析构函数，但智能指针或类似自动变量未管理的自动分配的内存或资源将泄漏。

内置类型均是无故障的，标准库类型支持最低级别的基本保证。 为必须是异常安全的任何用户定义类型遵循这些准则：

- 使用智能指针或其他 RAII 类型的包装器管理所有资源。 在类析构函数中避免资源管理功能，因为如果构造函数引发异常，则将不会调用析构函数。 但是，如果类是仅控制一个资源的专用资源管理器，则可接受使用析构函数管理资源。

- 了解无法在派生类构造函数中吞并基类构造函数中引发的异常。 如果要转换并再次引发派生构造函数中的基类异常，请使用 try 函数块。
- 考虑是否将所有类状态存储在包装在一个智能指针中的数据成员中，尤其是在类具有“允许初始化失败”概念时。虽然 C++ 允许未初始化的数据成员，但它不支持未初始化或部分初始化的类实例。 构造函数必须成功或失败；如果构造函数未完成运行，则将不会创建任何对象。
- 不允许任何异常从析构函数转义。 C++ 的基本原理是，析构函数绝不会允许异常传播到调用堆栈。 如果析构函数必须执行潜在引发异常的操作，则它必须使用 try catch 块如此做并吞并异常。 标准库将为其定义的所有析构函数提供此保证。

另请参阅

现代 C++ 处理异常和错误的最佳做法
如何：异常和非异常代码之间的接口

Last updated on 2025/11/07

如何：异常和非异常代码之间的接口

本文介绍如何在 C++ 代码中实现一致的异常处理，以及如何在异常边界将异常和错误代码进行互相转换。

有时 C++ 代码必须与不使用异常的代码（非异常代码）进行交互。 此类接口称为异常边界。 例如，您可能希望在 C++程序中调用 Win32 的函数 `CreateFile`。 `CreateFile` 不会引发异常。 相反，它会设置 `GetLastError` 函数可能检索到的错误代码。 如果你的 C++ 程序很重要，你可能更愿意部署一致的基于异常的错误处理策略。 此外，你可能不想仅仅因为你与非异常代码进行交互而放弃异常。 你也不希望在 C ++ 代码中将基于异常的错误策略与基于非异常的错误策略混合。

从 C++ 调用非异常函数

当您从 C++ 调用非异常函数时，您的想法是将该函数包装在检测任何错误的 C++函数中，然后可能引发异常。 当设计此类包装器函数时，首先应确定提供的异常保证的类型：noexcept、strong 或 basic。 其次，设计函数以使得异常引发时能够正确发布所有资源（例如文件句柄）。 通常，它意味着你可使用智能指针或类似的资源管理器来拥有资源。 有关设计注意事项的详细信息，请参阅如何：设计以实现异常安全性。

示例

以下示例演示使用 Win32 `CreateFile` 和 `ReadFile`函数在内部打开并读取两个文件的 C++ 函数。 `File` 类是文件句柄的“资源获取即是初始化”(RAII) 包装器。 其构造函数检测到“找不到文件”条件，并引发了在 C++ 可执行文件的调用堆栈（本示例中为 `main()` 函数）中向上传播错误的异常。 如果在完全构造好 `File` 对象后引发了异常，析构函数将自动调用 `CloseHandle` 以释放文件句柄。 （如果你愿意，可以将活动模板库 (ATL) `CHandle` 类用于此同一目的，或者将 `unique_ptr` 与自定义删除函数一起使用。） 调用 Win32 和 CRT API 的函数检测到错误，然后使用本地定义的 `ThrowLastErrorIf` 函数引发 C++ 异常，该异常反过来使用派生自 `Win32Exception` 类的 `runtime_error` 类。 本示例中的所有函数提供了强力的异常保证：当这些函数中的任何点上引发异常时，资源不会泄露，程序状态也不会修改。

C++

```
// 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
    }
}
```

从非异常代码调用异常代码

声明为 `extern "C"` 的 C++ 函数可由 C 程序调用。 C++ COM 服务器可由采用任意数量的不同语言编写的代码使用。 在使用 C++ 实现将由非异常代码调用的可识别异常的公共函数时，C++ 函数不得允许将任何异常传播回调用方。 此类调用方无法捕获或处理 C++ 异常。 程序可能会终止、泄露资源或导致未定义的行为。

我们建议让 `extern "C"` C++ 函数专门捕获它知道如何处理的所有异常，并根据需要将异常转换为调用方能理解的错误代码。 如果 C++函数并非了解所有可能的异常，它应将 `catch(...)` 块作为最后一个处理程序。 在这种情况下，最好向调用方报告错误，因为你的程序可能处于未知且无法恢复的状态。

以下示例演示了一个函数，该函数假定可能引发的任何异常是 `Win32Exception` 或者属于派生自 `std::exception` 的异常类型。 该函数将捕获这些类型的任何异常并将错误信息作为 Win32 错误代码传递给调用方。

C++

```
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;
}
```

从异常转换为错误代码时，有一个潜在问题：错误代码包含的信息通常不及异常可存储的信息丰富。 要解决此问题，可为每个可能引发的具体异常类型提供一个 `catch` 块，并在异常转换为错误代码前执行日志记录以记录异常的详细信息。 如果多个函数都使用同一组 `catch` 块，此方法可能产生重复代码。 避免代码重复的一个好方法是，将这些块重构到一个实现 `try` 和 `catch` 块并接受 `try` 块中调用的函数对象的专用实用工具函数中。 在每个公用函数中，将代码传递到实用工具函数以作为 lambda 表达式。

C++

```
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;
}
```

以下示例演示如何编写函数定义的 lambda 表达式。 lambda 表达式通常比调用命名函数对象的代码更容易内联读取。

C++

```
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;
    });
}
```

若要详细了解 lambda 表达式，请参阅 lambda 表达式。

从异常代码通过非异常代码调用异常代码

在无法识别异常的代码中引发异常是可能的，但不建议这样做。 例如，C++ 程序可能会调用一个使用你提供的回调函数的库。 在某些情况下，可以在原始调用方可处理的非异常代码中从回调函数引发异常。 但是，异常能够成功运行的环境是严格的。 必须以保留堆栈展开语义的方式编译库代码。 无法识别异常的代码不能执行任何可能捕获 C++ 异常的操作。 此外，调用方和回调之间的库代码无法分配本地资源。 例如，无法识别异常的代码不能有指向已分配堆内存的局部变量。 当堆栈展开时，这些资源会遭到泄露。

必须满足以下要求才能在可识别非异常的代码中引发异常：

- 你可以使用 `/EHs` 在可识别非异常的代码中构建整个代码路径，
- 在堆栈展开时，不会有任何本地分配的资源出现泄露，
- 代码没有任何用于捕获所有异常的 `__except` 结构化异常处理程序。

由于在非异常代码中引发异常很容易出错，并且可能会增加调试难度，因此不建议采用这种方式。

另请参阅

现代 C++ 处理异常和错误的最佳做法
如何：设计以实现异常安全性

Last updated on 2025/11/07

try、throw 和 catch 语句 (C++)

若要在 C++ 中实现异常处理，可以使用 `try`、`throw` 和 `catch` 表达式。

首先，使用 `try` 程序块将可能引发异常的一个或多个语句封闭起来。

`throw` 表达式发出信号，异常条件（通常是错误）已在 `try` 程序块中发生。 可以使用任何类型的对象作为 `throw` 表达式的操作数。 该对象一般用于传达有关错误的信息。 大多数情况下，建议使用 std::exception 类或标准库中定义的派生类之一。 如果其中的类不合适，建议你从 `std::exception` 派生自己的异常类。

若要处理可能引发的异常，请在 `catch` 程序块之后立即实现一个或多个 `try` 程序块。 每个 `catch` 程序块都会指定它能处理的异常类型。

以下示例将显示 `try` 程序块及其处理程序。 假设 `GetNetworkResource()` 通过网络连接获取数据，并且两个异常类型是从 `std::exception` 派生的用户定义的类。 请注意，异常由 `const` 语句中的 `catch` 引用捕获。 我们建议你通过值引发异常并通过常数引用将其捕获。

示例

C++

```
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");
    // ...
}
```

注解

`try` 子句后的代码是代码的受保护部分。 `throw` 表达式将引发（即引起）异常。 `catch` 子句后的代码块是异常处理程序。 如果 和 `throw` 表达式中的类型兼容，该处理程序将捕获引发的异常 `catch`。 有关管理 `catch` 程序块中的类型匹配的规则的列表，请参阅 Catch 程序块如何计算。 如果 `catch` 语句指定省略号 (...) 而非类型，则 `catch` 程序块将处理每种类型的异常。 当使用 /EHa 选项编译时，异常可包括 C 结构化异常和系统生成或应用程序生成的异步异常，例如内存保护、被零除和浮点冲突。 由于 `catch` 程序块按编程顺序处理来查找匹配类型，所以尽量不要使用省略号处理程序来处理关联的 `try` 程序块。 请谨慎使用 `catch(...)`；除非 catch 块知道如何处理捕获的特定异常，否则禁止程序继续执行。 `catch(...)` 块一般用于在程序停止执行前记录错误和执行特殊的清理工作。

没有操作数的 `throw` 表达式将重新引发当前正在处理的异常。 建议在重新引发异常时采用此形式，因为这将保留原始异常的多态类型信息。 此类表达式只应在 `catch` 处理程序中或从 `catch` 处理程序调用的函数中使用。 重新引发的异常对象是原始异常对象，而不是副本。

C++

```
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;
}
```

另请参阅

现代 C++ 处理异常和错误的最佳做法
关键字
未处理的 C++ 异常
__uncaught_exception

Last updated on 2025/11/07

Catch 块的计算方式 (C++)

虽然通常建议您引发派生自 std::exception 的类型，但 C++ 使您能够引发任何类型的异常。 可以通过指定与引发的异常相同的类型的 `catch` 处理程序或通过可捕获任何类型的异常的处理程序来捕获 C++ 异常。

如果引发的异常的类型是类，它还具有基类（或类），则它可由接受异常类型的基类和对异常类型的基的引用的处理程序捕获。 请注意，当异常由引用捕获时，会将其绑定到实际引发的异常对象；否则，它将为一个副本（与函数的参数大致相同）。

引发异常时，将由以下类型的 `catch` 处理程序捕获该异常：

- 可以接受任何类型的处理程序（使用省略号语法）。
- 接受与异常对象相同的类型的处理程序；由于它是副本，因此 `const` 和 `volatile` 修饰符将被忽略。
- 接受对与异常对象相同的类型的引用的处理程序。
- 接受对与异常对象相同类型的 `const` 或 `volatile` 形式的引用的处理程序。
- 接受与异常对象相同的类型的基类的处理程序；由于它是副本，因此 `const` 和 `volatile` 修饰符将被忽略。 基类的 `catch` 处理程序不得在派生类的 `catch` 处理程序之前。
- 接受对与异常对象相同的类型的基类的引用的处理程序。
- 接受与异常对象相同的类型的基类的 `const` 或 `volatile` 形式的引用的处理程序。
- 接受可通过标准指针转换规则将引发的指针对象转换为的指针的处理程序。

`catch` 处理程序出现的顺序是有意义的，因为给定 `try` 块的处理程序按它们的出现顺序进行检查。 例如，将基类的处理程序放置在派生类的处理程序的前面是错误的。 找到一个匹配的 `catch` 处理程序后，不会检查后续处理程序。 因此，省略号 `catch` 处理程序必须是其 `try` 块的最后一个处理程序。 例如：

C++

```
// ...
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.
}
```

在此示例中，省略号 `catch` 处理程序是唯一被检查的处理程序。

另请参阅

现代 C++ 处理异常和错误的最佳做法

Last updated on 2025/11/07

C++ 中的异常和堆栈展开

在 C++ 异常机制中，控制从 throw 语句移至可处理引发类型的第一个 catch 语句。 在到达 catch 语句时，throw 语句和 catch 语句之间的范围内的所有自动变量将在名为“堆栈展开”的过程中被销毁。 在堆栈展开中，执行将继续，如下所示：

1. 控制通过正常顺序执行到达 `try` 语句。 执行 `try` 块内的受保护部分。

2. 如果执行受保护部分的过程中未引发异常，将不会执行 `catch` 块后面的 `try` 子句。 执行将在关联的 `catch` 块后的最后一个 `try` 子句后面的语句上继续。

3. 如果在执行受保护部分的过程中或在受保护的部分调用的任何例程中引发异常（直接或间接），则从通过 `throw` 操作数创建的对象中创建异常对象。 （这意味着可能涉及复制构造函数。）此时，编译器会在权限更高的执行上下文中查找可处理所引发类型的异常的 `catch` 子句，或查找可以处理任何类型异常的 `catch` 处理程序。 按照 `catch` 处理程序在 `try` 块后面的显示顺序检查这些处理程序。 如果未找到适当的处理程序，则检查下一个动态封闭的 `try` 块。 此过程将继续，直到检查最外面的封闭 `try` 块。

4. 如果仍未找到匹配的处理程序，或者在展开过程中但在处理程序获得控制前发生异常，则调用预定义的运行时函数 `terminate`。 如果在引发异常后但在展开开始前发生异常，则调用 `terminate`。

5. 如果找到匹配的 `catch` 处理程序，并且它通过值进行捕获，则通过复制异常对象来初始化其形参。 如果它通过引用进行捕获，则初始化参数以引用异常对象。 在初始化形参后，堆栈的展开过程将开始。 这包括对与 `try` 处理程序关联的 `catch` 块的开头和异常的引发站点之间完全构造（但尚未析构）的所有自动对象的析构。 析构按照与构造相反的顺序发生。 执行 `catch` 处理程序且程序会在最后一个处理程序之后（即，在不是 `catch` 处理程序的第一个语句或构造处）恢复执行。 控制只能通过引发的异常进入 `catch` 处理程序，而绝不会通过 `goto` 语句或 `case` 语句中的 `switch` 标签进入。

堆栈展开示例

以下示例演示引发异常时如何展开堆栈。 线程执行将从 `C` 中的 throw 语句跳转到 `main` 中的 catch 语句，并在此过程中展开每个函数。 请注意创建 `Dummy` 对象的顺序，并且会在它们超出范围时将其销毁。 还请注意，除了包含 catch 语句的 `main` 之外，其他函数均未完成。 函数 `A` 绝不会从其对 `B()` 的调用返回，并且 `B` 绝不会从其对 `C()` 的调用返回。 如果取消注释 `Dummy` 指针和相应的 delete 语句的定义并运行程序，请注意绝不会删除该指针。 这说明了当函数不提供异常保证时会发生的情况。 有关详细信息，请参阅如何：针对异常进行设计。 如果注释掉 catch 语句，则可以观察当程序因未经处理的异常而终止时将发生的情况。

```
C++
```

```
#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.

*/
```

Last updated on 2025/11/07

异常规范 (throw、noexcept) (C++)

异常规范是一项 C++ 语言功能，指示程序员对可由函数传播的异常类型的意图。 可以使用*异常规范*指定函数可以或不可以因异常退出。 编译器可以使用此信息来优化对函数的调用，并在意外异常脱离函数时终止程序。

在 C++17 之前，有两种异常规范。 *noexcept 规范*是 C++11 中的新增规范。 它指定可以脱离函数的潜在异常集是否为空。 动态异常规范（`throw(optional_type_list)` 规范）在 C++11 中已弃用，并已在 C++17 中删除，但 `throw()` 除外，它是 `noexcept(true)` 的别名。 此异常规范原本用来提供有关可从函数引发哪些异常的摘要信息，但在实际应用中发现此规范存在问题。 证明确实有一定用处的一个动态异常规范是无条件 `throw()` 规范。 例如，函数声明：

```
C++
void MyFunction(int i) throw();
```

告诉编译器函数不引发任何异常。 但是，在 /std:c++14 模式下，如果函数确实引发异常，这可能会导致未定义的行为。 因此，建议使用 noexcept 运算符而不是上述运算符：

```
C++
void MyFunction(int i) noexcept;
```

下表总结了 Microsoft C++ 的异常规范实现：

展开表

异常规范	含义

`noexcept`
`noexcept(true)`
`throw()`
	函数不会引发异常。 在 **`/std:c++14`** 模式（默认）下，**`noexcept`** 和 `noexcept(true)` 是等效的。 从声明为 **`noexcept`** 或 `noexcept(true)` 的函数引发异常时，将调用 std::terminate。 当在 `throw()` 模式下从声明为 **`/std:c++14`** 的函数引发异常时，结果为未定义的行为。 未调用任何特定函数。 这与 C++14 标准不同，后者要求编译器调用 std::unexpected。
	Visual Studio 2017 版本 15.5 及更高版本：在 **`/std:c++17`** 模式下，**`noexcept`**、`noexcept(true)` 和 `throw()` 都是等效的。 在 **`/std:c++17`** 模式下，`throw()` 是 `noexcept(true)` 的别名。 在 **`/std:c++17`** 模式和更高版本模式中，当从用上述任何规范声明的函数引发异常时，将按 C++17 标准要求调用 std::terminate。

`noexcept(false)`
`throw(...)`
无规范
	函数可以引发任何类型的异常。

`throw(type)`	（C++14 和更早版本）函数可以引发 `type` 类型的异常。 编译器接受语法，但将其解释为 `noexcept(false)`。 在 **`/std:c++17`** 模式和更高版本模式中，编译器发出警告

异常规范	含义
	C5040。

如果在应用程序中使用异常处理，则调用堆栈中必须有一个函数，其在引发的异常退出标记 `noexcept`、`noexcept(true)` 或 `throw()` 的函数的外部范围之前处理这些异常。 如果在引发异常的函数与处理异常的函数之间调用的任何函数都指定为 `noexcept`， `noexcept(true)` （或在 `throw()` 模式下为 `/std:c++17`），则当 noexcept 函数传播异常时，程序将终止。

函数的异常行为基于以下因素：

- 设置了哪种语言标准编译模式。
- 您是否在 C 或 C++ 下编译函数。
- 所使用的 /EH 编译器选项。
- 是否显式指定异常规范。

不允许对 C 函数使用显式异常规范。 假定 C 函数在 `/EHsc` 下不引发异常，并且可能会在 `/EHs`、`/EHa` 或 `/EHac` 下引发结构化异常。

下表总结了 C++ 函数是否可能会在各种编译器异常处理选项下引发：

展开表

函数	`/EHsc`	`/EHs`	`/EHa`	`/EHac`
没有异常规范的 C++ 函数	是	是	是	是
带有 `noexcept`、`noexcept(true)` 或 `throw()` 异常规范的 C++ 函数	否	否	是	是
带有 `noexcept(false)`、`throw(...)` 或 `throw(type)` 异常规范的 C++ 函数	是	是	是	是

示例

C++

```
// 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();
}
```

Output

```
About to throw 1
in handler
About to throw 1
Caught exception from f4
```

```
About to throw 1
in handler
```

另请参阅

try、throw 和 catch 语句 (C++)
现代 C++ 处理异常和错误的最佳做法

Last updated on 2025/11/07

`noexcept` （C++）

C++11： 指定函数是否可能会引发异常。

语法

`noexcept-specifier`:

  `noexcept`

  `noexcept-expression`

  `throw` **`(`** **`)`**

`noexcept-expression`:

  `noexcept` **`(`** *`constant-expression`* **`)`**

参数

`constant-expression`
类型的常量表达式 **`bool`** ， 表示潜在异常类型的集是否为空。 无条件版本等效于 `noexcept(true)`.

注解

这是 *`noexcept-expression`* 一种 *异常规范*：函数声明的后缀，该后缀表示一组可能由异常处理程序匹配的任何异常退出函数的类型。 生成 **`true`** 时 *`constant_expression`* ， 一元条件运算符 `noexcept(constant_expression)` 及其无条件同义词 **`noexcept`** 指定可以退出函数的潜在异常类型集为空。 也就是说，该函数永远不会引发异常，也绝不允许在异常范围之外传播异常。 当生成 **`false`** 或 *`constant_expression`* 缺少异常规范（除析构函数或解除分配函数外）时，运算符 `noexcept(constant_expression)` 表示可以退出该函数的潜在异常集是所有类型的异常集。

仅当它调用的所有函数 **`noexcept`** （直接或间接）也是 **`noexcept`** **`const`** 或 。 编译器不一定检查每个代码路径是否存在可能升至 **`noexcept`** 函数的异常。 如果异常确实退出标记 **`noexcept`** 的函数的外部作用域， std::terminate 将立即调用，并且不能保证将调用任何作用域内对象的析构函数。 使用 **`noexcept`** 而不是动态异常说明符 `throw()`。 *动态异常规范*或 `throw(optional_type_list)` 规范在 C++11 中已弃用，并在 C++17 中删除，但该 `throw()` 规范是别名 `noexcept(true)`。 建议应用于 **`noexcept`** 任何永不允许异常传播调用堆栈的函数。 声明 **`noexcept`** 函数时，它使编译器能够在多个不同的上下文中生成更高效的代码。 有关详细信息，请参阅 异常规范。

示例：

复制其参数的函数模板可能在复制的对象是普通旧数据类型（POD）的条件上声明 `noexcept` 。此类函数可以如下声明：

C++

```
#include <type_traits>

template <typename T>
T copy_object(const T& obj) noexcept(std::is_pod<T>)
{
   // ...
}
```

另请参阅

现代 C++ 处理异常和错误的最佳做法
异常规范 （throw， noexcept）

Last updated on 2025/11/07

未处理的 C++ 异常

如果无法找到当前异常的匹配处理程序（或省略号 `catch` 处理程序），则调用预定义的 `terminate` 运行时函数。 （还可以在任何处理程序中显式调用 `terminate`。） `terminate` 的默认操作是调用 `abort`。 如果你希望 `terminate` 在退出应用程序之前调用程序中的某些其他函数，则用被调用函数的名称作为其单个自变量调用 `set_terminate` 函数。 您可以在程序的任何点调用 `set_terminate`。 `terminate` 例程总是调用指定为 `set_terminate` 的参数的最后一个函数。

示例

以下示例引发 `char *` 异常，但不包含用于捕获类型 `char *` 的异常的指定处理程序。 对 `set_terminate` 的调用指示 `terminate` 调用 `term_func`。

C++

```
// 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;
}
```

输出

Output

```
term_func was called by terminate.
```

`term_func` 函数最好是通过调用 `exit` 来终止程序或当前线程。 如果它没有这样做，而是返回到其调用方，则调用 `abort`。

另请参阅

现代 C++ 处理异常和错误的最佳做法

Last updated on 2025/11/07

混合使用 C（结构化）和 C++ 异常

若要编写可移植代码，建议不要在 C++ 程序中使用结构化异常处理 (SEH)。 但是，你有时可能希望使用 /EHa 进行编译并将结构化异常和 C++ 源代码组合在一起，并且需要用于处理这两种异常的某个设备。 由于结构化异常处理程序没有对象或类型化异常的概念，因此它无法处理 C++ 代码引发的异常。 但是，C++ `catch` 处理程序可以处理结构化异常。 C 编译器不接受 C++ 异常处理语法（`try`、`throw`、`catch`），但 C++ 编译器支持结构化异常处理语法（`__try`、`__except`、`__finally`）。

若要了解如何将结构化异常作为 C++ 异常处理，请参阅 _set_se_translator。

如果混合使用结构化异常和 C++ 异常，请注意以下潜在问题：

- 不能在同一函数中混合 C++ 异常和结构化异常。
- 始终执行终止处理程序（`__finally` 块），甚至在引发异常后的展开过程中也是如此。
- C++ 异常处理可以捕获并保留使用 /EH 编译器选项（此选项启用展开语义）编译的所有模块中的展开语义。
- 可以存在一些不为所有对象调用析构函数的情况。 例如，尝试通过未初始化的函数指针调用函数时，可能会出现结构化异常。 如果函数参数是在调用之前构造的对象，则在堆栈展开期间不会调用这些对象的析构函数。

后续步骤

- 在C++ 程序中使用 setjmp 或 longjmp

  查看关于在 C++ 程序中使用 `setjmp` 和 `longjmp` 的信息。

- 处理 C++ 中的结构性异常

  请参阅使用 C++ 处理结构化异常的方法的示例。

另请参阅

现代 C++ 处理异常和错误的最佳做法

Last updated on 2025/11/07

使用 setjmp 和 longjmp

当 setjmp 和 longjmp 一起使用时，它们提供了一种执行非本地 `goto` 的方法。 它们通常在 C 代码中用于将执行控制传递给先前调用的例程中的错误处理或恢复代码，而不使用标准调用或返回约定。

> **⊗ 注意**
>
> 由于 `setjmp` 和 `longjmp` 不支持在 C++ 编译器之间以可移植的方式正确销毁堆栈帧对象，并且它们可能会因阻止优化局部变量而降低性能，因此不建议在 C++ 程序中使用。 建议改用 `try` 和 `catch` 构造。

如果决定在 C++ 程序中使用 `setjmp` 和 `longjmp`，还应包括 <setjmp.h> 或 <setjmpex.h> 以确保函数和结构化异常处理之间的正确交互 (SEH) 或 C++ 异常处理。

Microsoft 专用

如果使用 /EH 选项编译 C++ 代码，则会在堆栈展开期间调用本地对象的析构函数。 但是，如果使用 **/EHs** 或 **/EHsc** 进行编译，并且其中一个函数使用 noexcept 调用 `longjmp`，则该函数的析构函数展开可能不会发生，具体取决于优化器状态。

在可移植代码中，执行 `longjmp` 调用时，标准不明确保证正确销毁基于帧的对象，并且其他编译器对此可能也不支持。 你应了解一点，在警告级别 4，对 `setjmp` 的调用会导致出现警告 C4611：“_setjmp”和 C++ 对象销毁之间的交互是不可移植的。

结束 Microsoft 专用

另请参阅

混合使用 C（结构化）和 C++ 异常

Last updated on 2025/11/07

处理 C++ 中的结构性异常

C 结构化异常处理 (SEH) 和 C++ 异常处理的主要区别是，C++ 异常处理模型处理的是多个类型，而 C 结构化异常处理模型处理的是一个类型（具体来说就是 `unsigned int`）的异常。 即，C 异常由无符号整数值标识，而 C++ 异常由数据类型标识。 当 C 中引发了结构化异常时，每个可能的处理程序都将执行筛选器来检查 C 异常上下文，并确定是接受该异常、将其传递给其他处理程序还是忽略它。 当 C++ 中引发了异常时，该异常可以是任何类型。

第二个区别是，C 结构化异常处理模型被称为是“异步的”，因为异常是从属在正常控制流之后发生的。 C++ 异常处理机制是完全“同步的”，这意味着异常仅在被引发时发生。

使用 /EHs 或 /EHsc 编译器选项时，C++ 异常处理程序均不会处理结构化异常。 这些异常仅由 `__except` 结构化异常处理程序或 `__finally` 结构化终止处理程序进行处理。 有关详细信息，请参阅结构化异常处理 (C/C++)。

在 /EHa 编译器选项下，如果在 C++ 程序中引发了 C 异常，可以由结构化异常处理程序与其关联筛选器一起处理，或者由 C++ `catch` 处理程序进行处理，具体取决于哪一种更加动态接近异常上下文。 例如，此示例 C++ 程序在 C++ `try` 上下文中引发了一个 C 异常：

示例 - 在 C++ catch 块中捕获 C 异常

C++

```
// 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;
```

```
    }
}
```

Output

```
In finally.
Caught a C exception.
```

C 异常包装类

在与上述类似的简单示例中，C 异常只能由省略号 (...) 处理程序捕获。 `catch` 有关类型或异常性质的信息不传递给该处理程序。 尽管此方法有效，但在某些情况下，你可能需要定义两个异常处理模型之间的转换，使每个 C 异常与一个特定类关联。 要进行转换，你可定义 C 异常“包装”类，可使用该类或从中进行派生来将特定类类型特性化为 C 异常。 这样，每个 C 异常都可以由特定的 C++ `catch` 处理程序单独处理，而不是在单个处理程序中处理所有异常。

您的包装器类可能有一个接口，该接口包含一些成员函数，用来确定异常的值以及访问 C 异常模型提供的扩展异常上下文信息。 你可能还希望定义一个默认构造函数、一个接受 `unsigned int` 参数（用于提供基础 C 异常表示形式）的构造函数和一个按位复制构造函数。 下面是 C 异常包装类的一个可能的实现：

C++

```cpp
// 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;
   }
};
```

要使用此类，请安装每次引发 C 异常时由内部异常处理机制调用的自定义 C 异常转换函数。 在转换函数中，可引发可由适当匹配的 C++ `SE_Exception` 处理程序捕获的任意类型的异常（可能是 `SE_Exception` 类型，或派生自 `catch` 的类类型）。 转换函数可能直接返回，这表示它没有处理异常。 如果转换函数本身引发了 C 异常，则会调用 terminate。

若要指定自定义转换函数，请使用转换函数的名称作为 _set_se_translator 函数的单一参数来调用它。 对于具有 `try` 块的堆栈上的每个函数调用，将调用一次编写的转换器函数。 没有默认转

换函数；如果你未通过调用 _set_se_translator 来指定转换函数，C 异常只能由省略号 处理程序捕获。`catch`

示例 - 使用自定义转换函数

例如，以下代码安装了自定义转换函数，然后引发了由 `SE_Exception` 类包装的 C 异常：

C++

```
// 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() );
    }
}
```

Output

```
In trans_func.
In finally
Caught a __try exception with SE_Exception.
nSE = 0xc0000094
```

另请参阅

混合使用 C（结构化）和 C++ 异常

Last updated on 2025/11/07

结构化异常处理 （C/C++）

结构化异常处理（SEH）是 C 的Microsoft扩展，C++用于处理某些异常代码情况（如硬件故障）正常。 尽管 Windows 和 Microsoft C++支持 SEH，但我们建议在 C++ 代码中使用 ISO 标准C++异常处理。 它使代码更具可移植性和灵活性。 但是，若要维护现有代码或特定类型的程序，仍可能需要使用 SEH。

特定于Microsoft：

语法

> `try-except-statement` ：
>
> `__try` *compound-statement* `__except` `(` *filter-expression* `)` *compound-statement*
>
> `try-finally-statement` ：
>
> `__try` *compound-statement* `__finally` *compound-statement*

注解

使用 SEH，可以确保在执行意外终止时正确释放资源（如内存块和文件）。 还可以通过使用不依赖 `goto` 语句或精心测试返回代码的简洁结构化代码来处理特定问题（例如内存不足）。

try-except本文中提到的语句try-finally是 C 和 C++ 语言的Microsoft扩展。 它们通过使应用程序能够在事件之后控制程序，从而支持 SEH，否则将终止执行。 尽管 SEH 适用于C++源文件，但它并非专为C++而设计。 如果在使用 /EHa 或 /EHsc 选项编译的C++程序中使用 SEH，将调用本地对象的析构函数，但其他执行行为可能不是预期的。 有关插图，请参阅本文后面的示例。 在大多数情况下，我们建议使用 ISO 标准 C++异常处理，而不是 SEH。 通过使用C++异常处理，可以确保代码更易于移植，并且可以处理任何类型的异常。

如果有使用 SEH 的 C 代码，则可以将其与使用C++异常处理C++代码混合使用。 有关信息，请参阅 C++中的句柄结构化异常。

有两种 SEH 机制：

- 异常处理程序或 `__except` 块，这些处理程序可以根据值响应或消除异常 `filter-expression` 。 有关详细信息，请参阅 try-except 语句。
- 无论异常是否导致终止，始终调用终止处理程序或`__finally`块。 有关详细信息，请参阅 try-finally 语句。

这两种类型的处理程序是不同的，但通过称为 *展开堆栈*的进程密切相关。 发生结构化异常时，Windows 将查找当前处于活动状态的最新安装的异常处理程序。 处理程序可以执行以下三项作

之一：

- 无法识别异常并将控制权传递给其他处理程序（`EXCEPTION_CONTINUE_SEARCH`）。
- 识别异常，但将其消除（`EXCEPTION_CONTINUE_EXECUTION`）。
- 识别异常并处理它（`EXCEPTION_EXECUTE_HANDLER`）。

识别异常的异常处理程序可能不在发生异常时运行的函数中。 它可能位于堆栈上更高得多的函数中。 堆栈帧上的当前正在运行的函数和所有其他函数将终止。 在此过程中，堆栈将 *解开*。 也就是说，终止函数的局部非静态变量将从堆栈中清除。

在展开堆栈时，作系统会调用为每个函数编写的任何终止处理程序。 通过使用终止处理程序，可以清理因异常终止而保持打开状态的资源。 如果已输入关键部分，可以在终止处理程序中退出它。 程序关闭时，可以执行其他管家任务，例如关闭和删除临时文件。

后续步骤

- 编写异常处理程序
- 编写终止处理程序
- 处理C++中的结构化异常

示例：

如前所述，如果在C++程序中使用 SEH 并使用或`/EHsc`选项编译`/EHa`本地对象的析构函数。 但是，如果在使用C++异常，在执行期间的行为可能不是预期的行为。 此示例演示了这些行为差异。

C++

```
#include <stdio.h>
#include <Windows.h>
#include <exception>

class TestClass
{
public:
    ~TestClass()
    {
        printf("Destroying TestClass!\n");
    }
};

__declspec(noinline) void TestCPPEX()
```

```
{
#ifdef CPPEX
    printf("Throwing C++ exception\n");
    throw std::exception("");
#else
    printf("Triggering SEH exception\n");
    volatile int *pInt = 0x00000000;
    *pInt = 20;
#endif
}

__declspec(noinline) void TestExceptions()
{
    TestClass d;
    TestCPPEX();
}

int main()
{
    __try
    {
        TestExceptions();
    }
    __except(EXCEPTION_EXECUTE_HANDLER)
    {
        printf("Executing SEH __except block\n");
    }

    return 0;
}
```

如果用于 **`/EHsc`** 编译此代码，但本地测试控制宏 `CPPEX` 未定义， `TestClass` 则析构函数不会运行。 输出如下所示：

Output

```
Triggering SEH exception
Executing SEH __except block
```

如果使用 **`/EHsc`** 编译代码并使用 `CPPEX` `/DCPPEX` （以便引发C++异常）， `TestClass` 则析构函数将运行，输出如下所示：

Output

```
Throwing C++ exception
Destroying TestClass!
Executing SEH __except block
```

如果使用 **`/EHa`** 编译代码， `TestClass` 析构函数将执行是使用标准C++ `throw` 表达式还是 SEH 引发异常。 也就是说，是否 `CPPEX` 定义。 输出如下所示：

Output

```
Throwing C++ exception
Destroying TestClass!
Executing SEH __except block
```

有关详细信息，请参阅/EH（异常处理模型）。

特定于 END Microsoft

另请参阅

异常处理
关键字
<exception>
错误和异常处理
结构化异常处理 （Windows）

Last updated on 2025/11/07

编写异常处理程序

异常处理程序通常用于响应特定错误。 您可以使用异常处理语法筛选出您不知道如何处理的所有异常。 其他异常应该传递到为了查找特定异常而编写的其他处理程序（可能在运行库或操作系统中）。

异常处理程序使用 try-except 语句。

你想进一步了解什么?

- try-except 语句
- 编写异常筛选器
- 引发软件异常
- 硬件异常
- 对异常处理程序的限制

另请参阅

结构化异常处理 （C/C++）

Last updated on 2025/11/07

`try-except` 语句

该 `try-except` 语句是一个 **特定于Microsoft的** 扩展，支持 C 和 C++ 语言中的结构化异常处理。

C++

```
// . . .
__try {
    // guarded code
}
__except ( /* filter expression */ ) {
    // termination code
}
// . . .
```

语法

> *`try-except-statement`*:
>
> **`__try`** *`compound-statement`* **`__except (`** *`expression`* **`)`** *`compound-statement`*

注解

该 `try-except` 语句是 C 语言和C++语言的Microsoft扩展。 它使目标应用程序能够控制正常终止程序执行的事件的时间。 此类事件称为 *结构化异常*，或短时间内 *的异常* 。 处理这些异常的机制称为 *结构化异常处理* （SEH）。

有关相关信息，请参阅 try-finally 语句。

异常可以是基于硬件的，也可以是基于软件的。 即使应用程序无法完全从硬件或软件异常中恢复，结构化异常处理也很有用。 SEH 可以显示错误信息并捕获应用程序的内部状态，以帮助诊断问题。 对于不容易重现的间歇性问题，它特别有用。

> **注意**
>
> 结构化异常处理适用于 C 和C++源文件的 Win32。 但是，它不是专门为C++设计的。 可以使用C++异常处理来确保代码更易于移植。 此外，C++异常处理更灵活，因为它可以处理任何类型的异常。 对于C++程序，建议使用本机C++异常处理： try、catch 和 throw 语句。

子句后面的 `__try` 复合语句是 *正文* 或 *受保护的* 节。 该 `__except` 表达式也称为 *筛选器* 表达式。 其值确定异常的处理方式。 子句后面的 `__except` 复合语句是异常处理程序。 处理程序指

定在执行正文节期间引发异常时要执行的作。 执行继续，如下所示：

1. 执行受保护的节。
2. 如果在执行受保护的节期间未发生异常，则子句后 `__except` 语句将继续执行。
3. 如果在执行受保护的节期间或受保护的节调用的任何例程中发生异常，则会 `__except` 计算表达式。 有三种可能的值：
   - `EXCEPTION_CONTINUE_EXECUTION` （-1）异常将被消除。 在发生异常时继续执行。
   - `EXCEPTION_CONTINUE_SEARCH` （0） 无法识别异常。 继续搜索处理程序的堆栈，首先搜索包含 `try-except` 语句，然后搜索具有下一个最高优先级的处理程序。
   - `EXCEPTION_EXECUTE_HANDLER` （1） 识别异常。 通过执行 `__except` 复合语句将控制权传输到异常处理程序，然后在块之后 `__except` 继续执行。

表达式 `__except` 的计算结果为 C 表达式。 限制为单个值、条件表达式运算符或逗号运算符。 如果需要更广泛的处理，表达式可以调用一个例程，该例程返回上面列出的三个值之一。

每个应用程序可以有自己的异常处理程序。

跳入 `__try` 语句无效，但跳出一个语句是有效的。 如果在执行 `try-except` 语句的过程中终止进程，则不会调用异常处理程序。

为了与以前的版本兼容，_try、_except和_leave是同`__try` `__except`义词， `__leave`除非指定编译器选项 /Za （禁用语言扩展）。

`__leave` 关键字

关键字 `__leave` 仅在语句的 `try-except` 受保护的节内有效，其效果是跳转到受保护的节的末尾。 异常处理程序之后的第一个语句将继续执行。

语句 `goto` 也可以跳出受保护的部分，它不会降低性能，就像 **在** try-finally 语句中那样。 这是因为堆栈展开不会发生。 但是，我们建议使用 `__leave` 关键字而不是 `goto` 语句。 原因是，如果受保护的节很大或很复杂，则不太可能犯编程错误。

结构化异常处理内部函数

结构化异常处理提供两个可用于 `try-except` 语句的内部函数： GetExceptionCode 和 GetExceptionInformation。

`GetExceptionCode` 返回异常的代码（32 位整数）。

内部函数 `GetExceptionInformation` 返回指向包含有关异常的其他信息的 EXCEPTION_POINTERS 结构的指针。 通过此指针，可以访问硬件异常时存在的计算机状态。 结构如下所示：

C++

```
typedef struct _EXCEPTION_POINTERS {
    PEXCEPTION_RECORD ExceptionRecord;
    PCONTEXT ContextRecord;
} EXCEPTION_POINTERS, *PEXCEPTION_POINTERS;
```

指针类型`PEXCEPTION_RECORD`并在 include 文件 <winnt.h> 中定义，并在 `_CONTEXT` `_EXCEPTION_RECORD` include 文件<摘录`PCONTEXT`.h 中定义>

可以在异常处理程序中使用 `GetExceptionCode` 。 但是，只能在 `GetExceptionInformation` 异常筛选器表达式中使用。 它指向的信息通常位于堆栈上，当控件传输到异常处理程序时不再可用。

终止处理程序中提供了内部函数 AbnormalTermination 。 如果 **try-finally** 语句的正文按顺序终止，则返回 0。 在所有其他情况下，它返回 1。

<excpt.h> 为这些内部函数定义一些备用名称：

`GetExceptionCode` 等效于 `_exception_code`

`GetExceptionInformation` 等效于 `_exception_info`

`AbnormalTermination` 等效于 `_abnormal_termination`

示例：

C++

```
// 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");
}
```

输出

Output

```
hello
in try
in try
in filter.
caught AV as expected.
in finally. termination:
        abnormal
in except
world
```

另请参阅

编写异常处理程序
结构化异常处理 （C/C++）
关键字

Last updated on 2025/11/07

编写异常筛选器

您可以通过跳转到异常处理程序的级别或通过继续执行来处理异常。 可以使用*筛选*表达式来清理问题，而不是使用异常处理程序代码来处理异常并失败。 然后，通过返回 `EXCEPTION_CONTINUE_EXECUTION` (-1)，可以恢复正常流，而无需清除堆栈。

> **注意**
>
> 有些异常无法继续。 对这样的异常来说，如果*筛选*计算结果为 -1，则系统将产生新的异常。 调用 RaiseException 时，将确定异常是否会继续。

例如，下面的代码使用*筛选*表达式中的函数调用：此函数处理该问题并返回 –1 以继续正常控制流：

C++

```
// exceptions_Writing_an_Exception_Filter.cpp
#include <windows.h>
int Eval_Exception(int);
int main() {
    __try {
        ;
    }
    __except (Eval_Exception(GetExceptionCode())) {
        ;
    }
}
void HandleOverflow() {
    // Gracefully recover
}
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
    HandleOverflow();
    return EXCEPTION_CONTINUE_EXECUTION;
}
```

每当*筛选*需要执行任何复杂操作时，最好在*筛选*表达式中使用函数调用。 计算表达式将导致函数的执行，在此示例中，`Eval_Exception`。

请注意使用 GetExceptionCode 确定异常。 必须在 `__except` 语句的筛选表达式内调用此函数。 `Eval_Exception` 不能调用 `GetExceptionCode`，但是它必须具有传递给它的异常代码。

除非异常是整数或浮点溢出，否则此处理程序会将控制权传递给其他处理程序。 如果是，则处理程序调用函数（`HandleOverflow` 只是一个示例，而不是 API 函数），以适当地尝试从异常中恢复。 **`__except`** 语句块（在此示例中为空）无法执行，因为 `Eval_Exception` 绝不返回 `EXCEPTION_EXECUTE_HANDLER` (1)。

使用函数调用是处理复杂筛选器表达式的一个很好的通用方法。 其他两个有用的 C 语言功能是：

- 条件运算符
- 逗号运算符

条件运算符在这里通常很有用。 它可用于检查特定返回代码，然后返回两个不同值中的一个。 例如，仅当异常为 `STATUS_INTEGER_OVERFLOW` 时，下面代码中的筛选器才能识别异常：

```C++
__except (GetExceptionCode() == STATUS_INTEGER_OVERFLOW ? 1 : 0)
```

条件运算符在此示例中的用途主要是进行澄清，因为下面的代码将生成相同的结果：

```C++
__except (GetExceptionCode() == STATUS_INTEGER_OVERFLOW)
```

条件运算符在你希望筛选器的计算结果为 –1、 `EXCEPTION_CONTINUE_EXECUTION` 时更有用。

逗号运算符允许按顺序执行多个表达式。 然后返回最后一个表达式的值。 例如，下面的代码将异常代码存储在一个变量中，然后测试它：

```C++
__except (nCode = GetExceptionCode(), nCode == STATUS_INTEGER_OVERFLOW)
```

另请参阅

编写异常处理程序
结构化异常处理 （C/C++）

Last updated on 2025/11/07

引发软件异常

系统不会将某些最常见的程序错误源标记为异常。 例如，如果你尝试分配内存块，但没有足够的内存，则运行时或 API 函数不会引发异常，但会返回一个错误代码。

但可通过以下方式将任何条件视为异常：在代码中检测该条件，然后调用 RaiseException 函数来报告它。 通过按此方式标记错误，您可以将结构化异常处理的优点引入任何类型的运行时错误中。

对错误使用结构化异常处理：

- 为事件定义你自己的异常代码。
- 在检测到问题时调用 `RaiseException`。
- 使用异常处理筛选器来测试您定义的异常代码。

<winerror.h> 文件显示异常代码的格式。 若要确保不定义与现有异常代码发生冲突的代码，请将第三个最高有效位设置为 1。 应设置四个最高有效位，如下表所示。

展开表

Bits	建议的二进制设置	说明
31-30	11	这两个位描述代码的基本状态：11 = 错误，00 = 成功，01 = 信息性，10 = 警告。
29	1	客户端位。 为用户定义的代码将其设置为 1。
28	0	保留位。 （将其设置为 0。）

虽然“错误”设置适用于大多数异常，但如果需要，您可将前两个位设置为 11 二进制之外的设置。 请务必记住设置 29 和 28 位，如上表所示。

因此，生成的错误代码应将最高的四个位设置为十六进制 E。例如，以下定义定义了不与任何 Windows 异常代码发生冲突的异常代码。 （但是，你可能需要检查第三方 DLL 使用了哪些代码。）

C++

```
#define STATUS_INSUFFICIENT_MEM       0xE0000001
#define STATUS_FILE_BAD_FORMAT        0xE0000002
```

在定义异常代码后，可以使用它来引发异常。 例如，下面的代码引发 `STATUS_INSUFFICIENT_MEM` 异常以响应内存分配问题：

```C++
lpstr = _malloc( nBufferSize );
if (lpstr == NULL)
    RaiseException( STATUS_INSUFFICIENT_MEM, 0, 0, 0);
```

如果只需引发异常，则可以将最后三个参数设置为 0。 最后三个参数对于传递附加信息和设置阻止处理程序继续执行的标记很有用。 有关详细信息，请参阅 Windows SDK 中的 RaiseException 函数。

在异常处理筛选器中，您随后可以测试已定义的代码。 例如：

```C++
__try {
    ...
}
__except (GetExceptionCode() == STATUS_INSUFFICIENT_MEM ||
        GetExceptionCode() == STATUS_FILE_BAD_FORMAT )
```

另请参阅

编写异常处理程序
结构化异常处理 (C/C++)

Last updated on 2025/11/07

硬件异常

操作系统识别的大多数标准异常是硬件上定义的异常。 Windows 可识别少数低级别的软件异常，但这些异常通常最好通过操作系统进行处理。

Windows 将不同处理器的硬件错误映射到本节中的异常代码。 在某些情况下，处理器可能只生成这些异常中的一部分。 Windows 对异常的相关信息进行预处理并发布相应的异常代码。

Windows 识别的硬件异常在下表中进行了汇总：

展开表

异常代码	异常的原因
STATUS_ACCESS_VIOLATION	读取或写入不可访问的内存位置。
STATUS_BREAKPOINT	遇到硬件定义的断点；仅由调试器使用。
STATUS_DATATYPE_MISALIGNMENT	在没有正确对齐的地址上读取或写入数据；例如，16 位实体必须在 2 字节边界对齐。 （不适用于 Intel 80 x 86 处理器。）
STATUS_FLOAT_DIVIDE_BY_ZERO	将浮点类型除以 0.0。
STATUS_FLOAT_OVERFLOW	超过浮点类型的最大正指数。
STATUS_FLOAT_UNDERFLOW	超过浮点类型的最小负指数的大小。
STATUS_FLOATING_RESEVERED_OPERAND	使用保留的浮点格式（无效的使用格式）。
STATUS_ILLEGAL_INSTRUCTION	尝试执行处理器未定义的指令代码。
STATUS_PRIVILEGED_INSTRUCTION	执行当前计算机模式下不允许的指令。
STATUS_INTEGER_DIVIDE_BY_ZERO	将整数类型除以 0。
STATUS_INTEGER_OVERFLOW	尝试超出整数的范围的操作。
STATUS_SINGLE_STEP	以单步模式执行一条指令；仅由调试器使用。

上表中列出的很多异常应由调试器、操作系统或其他低级别代码处理。 你的代码不应处理这些错误（整数和浮点错误除外）。 因此，您通常应使用异常处理筛选器来忽略异常（计算结果为 0）。 否则，您可能阻止低级别机制进行适当的响应。 但是，可以通过编写终止处理程序，采取适当的预防措施来消除这些低级别错误的潜在影响。

另请参阅

编写异常处理程序

结构化异常处理 （C/C++）

Last updated on 2025/11/07

对于异常处理程序的限制

在代码中使用异常处理程序的主要限制是不能使用 `goto` 语句跳转到 `__try` 语句块。 相反，您必须通过常规控制流进入此语句块。 可以随意跳出 `__try` 语句块和嵌套异常处理程序。

另请参阅

编写异常处理程序
结构化异常处理 （C/C++）

Last updated on 2025/11/07

编写终止处理程序

与异常处理程序不同，无论代码的受保护块是否已正常终止，终止处理程序总是会执行。 终止处理程序的唯一用途应该是确保无论代码的节如何完成执行，内存、句柄和文件等资源都能正确关闭。

终止处理程序使用 try-finally 语句。

你想进一步了解什么?

- The try-finally 语句
- 清理资源
- 异常处理的操作的计时
- 终止处理程序的限制

另请参阅

结构化异常处理 （C/C++）

Last updated on 2025/11/07

`try-finally` 语句

该 `try-finally` 语句是一个 **特定于Microsoft的** 扩展，支持 C 和 C++ 语言中的结构化异常处理。

语法

以下语法描述了 `try-finally` 该语句：

C++

```
// . . .
__try {
    // guarded code
}
__finally {
    // termination code
}
// . . .
```

语法

> *`try-finally-statement`*:
>
> **`__try`** *`compound-statement`* **`__finally`** *`compound-statement`*

该 `try-finally` 语句是 C 和 C++ 语言的Microsoft扩展，使目标应用程序能够保证在代码块执行中断时执行清理代码。 清理包括解除分配内存、关闭文件和释放文件句柄等任务。 该 `try-finally` 语句对于具有多个位置的例程特别有用，这些例程检查了可能导致例程过早返回的错误。

有关相关信息和代码示例，请参阅 try-except 语句。 有关结构化异常处理的详细信息，请参阅结构化异常处理。 有关使用 C++/CLI 在托管应用程序中处理异常的详细信息，请参阅下面的/clr 异常处理。

> **注意**
>
> 结构化异常处理适用于 C 和C++源文件的 Win32。 但是，它不是专门为C++设计的。 可以使用C++异常处理来确保代码更易于移植。 此外，C++异常处理更灵活，因为它可以处理任何类型的异常。 对于C++程序，建议使用C++异常处理机制（trycatch和throw语句）。

子句后面的 `__try` 复合语句是受保护的节。 子句后面的 `__finally` 复合语句是终止处理程序。 处理程序指定在退出受保护的节时执行的一组作，无论是通过异常（异常终止）还是标准终止（正常终止）退出受保护的节。

通过简单的顺序执行（倒入）控制到达 `__try` 语句。 当控件进入控件 `__try`时，其关联的处理程序将变为活动状态。 如果控制流到达 try 块的末尾，则执行将按如下方式继续：

1. 调用终止处理程序。

2. 终止处理程序完成后，执行将继续在 `__finally` 语句之后执行。 但是，受保护的节结束（例如，通过 `goto` 受保护的正文或 `return` 语句），终止处理程序在控制流移出受保护的节 *之前* 执行。

   语句 `__finally` 不会阻止搜索适当的异常处理程序。

如果在块中 `__try` 发生异常，作系统必须找到异常的处理程序，否则程序将失败。 如果找到处理程序，则会执行所有 `__finally` 块，并在处理程序中恢复执行。

例如，假设一系列函数调用将函数 A 链接到函数 D，如下图所示。 每个函数都有一个终止处理程序。 如果在函数 D 中引发并在 A 中处理异常，则会按此顺序调用终止处理程序，因为系统展开堆栈：D、C、B。

终止处理程序执行的顺序

> ⓘ **注意**
>
> try-finally 的行为不同于支持使用 `finally`的其他一些语言，例如 C# 。 一个可能 `__try` 具有两者，但不能同时 `__finally` 具有和 `__except`。 如果两者一起使用在一起，则外部 try-except 语句必须包含内部 try-finally 语句。 指定每个块执行时间的规则也不同。

为了与以前的版本、`_try`和`_leave` `_finally`同义词兼容`__try` `__finally`， `__leave`除非指定编译器选项/Za（禁用语言扩展）。

__leave 关键字

关键字 `__leave` 仅在语句的 `try-finally` 受保护的节内有效，其效果是跳转到受保护的节的末尾。 在终止处理程序的第一个语句处继续执行。

语句 `goto` 也可以跳出受保护的节，但它会降低性能，因为它调用堆栈展开。 该 `__leave` 语句更高效，因为它不会导致堆栈展开。

异常终止

`try-finally`使用 longjmp 运行时函数退出语句被视为异常终止。 跳入一 `__try` 份声明是不合法的，但从一份声明中跳出来是合法的。 必须在出发点（块的正常终止`__try`）和处理异常的块（`__except`处理异常的块）之间处于活动状态的所有`__finally`语句。 它称为 *本地展开*。

`__try`如果由于任何原因（包括跳出块）而过早终止块，则系统将执行关联的`__finally`块作为展开堆栈过程的一部分。 在这种情况下，如果从块内部`__finally`调用，则AbnormalTermination函数返回`true`;否则返回 `false`。

如果在执行 `try-finally` 语句的过程中终止进程，则不会调用终止处理程序。

特定于 END Microsoft

另请参阅

编写终止处理程序
结构化异常处理 （C/C++）
关键字
终止处理程序语法

Last updated on 2025/11/07

清理资源

在终止处理程序执行期间，您在调用终止处理程序之前，可能无法知道获取了哪些资源。 `__try` 语句块可能会在所有资源被获取之前中断，因此并不会打开所有资源。

为安全起见，应检查以查看哪些资源在终止处理清理之前已打开。 建议的过程是：

1. 将句柄初始化为 NULL。
2. 在 `__try` 语句块中，获取资源。 随着资源的获取，句柄将被设置为正值。
3. 在 `__finally` 语句块中，释放其对应的句柄或标志变量是非零且非 Null 的资源。

示例

例如，以下代码在使用终止处理程序关闭三个文件并释放内存块。 这些资源是在 `__try` 语句块中获取的。 在清理资源之前，代码应先检查是否已获取资源。

C++

```
// 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();
}
```

另请参阅

编写终止处理程序
结构化异常处理 （C/C++）

Last updated on 2025/11/07

异常处理的计时：摘要

无论 `__try` 语句块如何终止，都要执行终止处理程序。 原因包括跳出 `__try` 块，一个将控制权转移出该块的 `longjmp` 语句，以及由于异常处理而展开堆栈。

> **注意**
>
> Microsoft C++ 编译器支持 `setjmp` 和 `longjmp` 两种形式的语句。 快速版本会跳过终止处理，但更高效。 若要使用此版本，请包含文件 <setjmpex.h>。 另一个版本支持上一段中所述的终止处理。 若要使用此版本，请包含文件 <setjmpex.h>。 快速版本的性能提升取决于硬件配置。

在执行任何其他代码前，操作系统将以适当的顺序执行所有终止处理程序，包括异常处理程序的主体。

当中断的原因是异常时，系统在决定要终止的内容前必须先执行一个或多个异常处理程序的筛选器部分。 事件的顺序如下：

1. 引发异常。
2. 系统会查看活动异常处理程序的层次结构，并执行优先级最高的处理程序筛选。 这是最近安装、嵌套最深的的异常处理程序，通过块和函数调用进行。
3. 如果此筛选器传递控制权（返回 0），则过程将继续，直到找到一个不传递控制权的筛选器。
4. 如果此筛选器返回 –1，则在引发异常的地方继续执行，而不会发生终止。
5. 如果筛选器返回 1，则发生以下事件：
    - 系统展开堆栈：清除抛出异常的位置以及包含异常处理程序的堆栈帧之间的所有堆栈帧。
    - 当堆栈展开时，堆栈上的所有终止处理程序都将执行。
    - 执行异常处理程序本身。
    - 控制权将交给此异常处理程序末尾后的代码行。

另请参阅

编写终止处理程序
结构化异常处理 （C/C++）

Last updated on 2025/11/07

对于终止处理程序的限制

不能使用 `goto` 语句跳入 `__try` 语句块或 `__finally` 语句块。 相反，您必须通过常规控制流进入此语句块。 （但是，可以跳出 `__try` 语句块。）此外，不能将异常处理程序或终止处理程序嵌入 `__finally` 块。

终止处理程序中允许的某些类型的代码会生成有问题的结果，因此您应小心使用它们（如果要使用）。 其中一个是跳出 `goto` 语句块的 `__finally` 语句。 如果该块作为正常终止的一部分执行，则不会发生任何异常。 但是如果系统正在展开堆栈，则该展开会停止。 然后，当前函数会获取控制权，就象异常终止不存在一样。

`return` 语句块内的 `__finally` 语句存在大致相同的情况。 控制权将返回给包含终止处理程序的函数的直接调用方。 如果系统正在展开堆栈，此进程将暂停。 然后，如果没有引发过异常，则程序将继续执行。

另请参阅

编写终止处理程序
结构化异常处理 （C/C++）

Last updated on 2025/11/07

在线程之间传输异常

Microsoft C++ 编译器 (MSVC) 支持从一个线程向另一个线程传输异常。 通过传输异常，你可以在一个线程中捕获异常，然后使该异常看似是在另一个线程中引发的。 例如，你可以使用该功能编写多线程应用程序，其中主线程将处理其辅助线程引发的所有异常。 传输异常对创建并行编程库或系统的开发人员最有用处。 为实现传输异常，MSVC 提供了 exception_ptr 类型以及 current_exception、rethrow_exception 和 make_exception_ptr 函数。

语法

C++

```
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;
}
```

参数

unspecified
用于实现 `exception_ptr` 类型的未指定的内部类。

p
引用异常的 `exception_ptr` 对象。

E
表示异常的类。

e
参数 `E` 类的实例。

返回值

`current_exception` 函数返回引用当前进行中的异常的 `exception_ptr` 对象。 如果没有进行中的异常，该函数将返回未与任何异常关联的`exception_ptr`对象。

`make_exception_ptr`函数返回引用`exception_ptr`参数指定的异常的*e*对象。

注解

方案

假设你要创建能伸缩以处理可变工作负荷的应用程序。 为了实现此目标，你要设计一个多线程应用程序，其中初始的主线程会创建所需数量的辅助线程，以完成该工作。 辅助线程可帮助主线程管理资源、平衡负载和提高吞吐量。 通过分发工作，多线程应用程序的表现优于单线程应用程序。

但是，如果辅助线程引发异常，你希望主线程予以处理。 这是因为你希望你的应用程序无论有多少辅助线程，都能以一致统一的方式处理异常。

解决方案

为处理先前的方案，C++ 标准支持在线程之间传输异常。 如果辅助线程引发异常，该异常会成为当前异常。 拿现实世界打比方，当前异常就好比处于动态的状态。 当前异常从引发之时到捕获它的异常处理程序返回之时就处于飞行状态。

辅助线程可在 `catch` 块中捕获当前异常，然后调用 `current_exception` 函数，将该异常存储在 `exception_ptr` 对象中。 `exception_ptr` 对象必须可用于辅助线程和主线程。 例如，`exception_ptr` 对象可以是全局变量，由 mutex 控制对它的访问。 术语“传输异常”指的是一个线程中的异常可以转换为可由其他线程访问的形式。

接下来，主线程调用 `rethrow_exception` 函数，该函数提取并继而引发 `exception_ptr` 对象中的异常。 异常引发后，将成为主线程中的当前异常。 也就是说，该异常看起来源自主线程。

最后，主线程可以在 `catch` 块中捕获当前异常，然后进行处理或将其抛给更高级别的异常处理程序。 或者，主线程可以忽略该异常并允许该进程结束。

大多数应用程序不必在线程之间传输异常。 但是，该功能在并行计算系统中有用，是因为该系统可以在辅助线程、处理器或内核间分配工作。 在并行计算环境中，单个专用线程可以处理辅助线程中的所有异常，并可以为任何应用程序提供一致的异常处理模型。

有关 C++ 标准委员会建议的详细信息，请在 Internet 中搜索编号为 N2179，标题为“Language Support for Transporting Exceptions between Threads”（在线程之间传输异常的语言支持）的文档。

异常处理模型和编译器选项

你的应用程序的异常处理模型决定了它是否可以捕获和传输异常。 Visual C++ 支持三种处理 C++ 异常的模型：ISO 标准 C++ 异常处理、结构化异常处理 (SEH)，以及公共语言运行时 (CLR) 异常。 使用/EH和/clr编译器选项以指定应用程序的异常处理模型。

只有编译器选项和编程语句的以下组合可以传输异常。 其他组合要么不能捕获异常，要么能捕获但不能传输异常。

- `/EHa`编译器选项与`catch`语句可以传输 SEH 和 C++ 异常。
- `/EHa`、`/EHs`和`/EHsc`编译器选项与`catch`语句可以传输 C++ 异常。
- `/clr`编译器选项与`catch`语句可以传输 C++ 异常。 `/clr`编译器选项表示`/EHa`选项的规范。 编译器不支持传输托管异常。 这是因为，从System.Exception 类派生的托管异常已经是可使用公共语言运行时工具在线程间移动的对象。

> **ⓘ 重要**
>
> 建议指定`/EHsc`编译器选项并仅捕获 C++ 异常。 如果使用`/EHa`或`/clr`编译器选项和含有省略`catch` ()的`catch(...)`语句，就可能面临安全威胁。 你可能希望使用 `catch` 语句捕获几个特定的异常。 但是， `catch(...)` 语句将捕获所有的 C++ 和 SEH 异常，包括致命的意外异常。 如果忽略意外异常或处理不当，恶意代码就可以趁此机会破坏你程序的安全性。

使用情况

后面几节将介绍如何使用 `exception_ptr` 类型以及 `current_exception`、 `rethrow_exception` 和 `make_exception_ptr` 函数传输异常。

exception_ptr 类型

使用 `exception_ptr` 对象可引用当前异常或用户指定异常的实例。 在 Microsoft 实现中，异常由EXCEPTION_RECORD结构表示。 每个 `exception_ptr` 对象包含一个异常引用字段，该字段指向表示异常的 `EXCEPTION_RECORD` 结构的副本。

声明`exception_ptr`变量时，该变量不与任何异常相关联。 也就是说，其异常引用字段为 NULL。 此类 `exception_ptr` 对象称为 *null exception_ptr*。

使用 `current_exception` 或 `make_exception_ptr` 函数可将异常指派给 `exception_ptr` 对象。 将异常指派给 `exception_ptr` 变量时，该变量的异常引用字段将指向该异常的副本。 如果没有足够的内存来复制异常，异常引用字段将指向 std::bad_alloc 异常的副本。 如果`current_exception`或`make_exception_ptr`函数因任何其他原因不能复制异常，该函数会调用terminate函数以退出当前进程。

虽然名称像是指针，但`exception_ptr`对象本身不属于指针。 它不遵循指针语义，不能与指针成员访问 (`->`) 或间接寻址 (`*`) 运算符一起使用。 `exception_ptr` 对象没有公共数据成员或成员函数。

比较

你可以使用相等 (`==`) 和不相等 (`!=`) 运算符比较两个 `exception_ptr` 对象。 这两个运算符不比较表示异常的`EXCEPTION_RECORD`结构的二进制值（位模式）。 而是比较 `exception_ptr` 对象的异常引用字段中的地址。 因此，null `exception_ptr`和 NULL 值的比较结果为相等。

current_exception 函数

调用 `current_exception` 块中的 **`catch`** 函数。 如果异常处于动态的状态，而且 **`catch`** 块可捕获该异常， `current_exception` 函数将返回引用该异常的 `exception_ptr` 对象。 否则，该函数将返回 null `exception_ptr` 对象。

详细信息

`current_exception` 函数会捕获动态异常，而不管 **`catch`** 语句是否指定 exception-declaration 语句。

如果不重新引发当前异常，会在**`catch`**块的末尾调用该异常的析构函数。 但是，即使调用析构函数中的 `current_exception` 函数，该函数仍返回引用当前异常的 `exception_ptr` 对象。

对 `current_exception` 函数的相继调用将返回引用当前异常的不同副本的 `exception_ptr` 对象。 因此，由于对象引用不同的副本，即使副本具有相同的二进制值，其比较结果也是不相等。

SEH 异常

如果使用**`/EHa`**编译器选项，可以在 C++ **`catch`**块中捕获 SEH 异常。 `current_exception` 函数返回引用 SEH 异常的 `exception_ptr` 对象。 如果调用`rethrow_exception`函数，将传输的`exception_ptr`对象作为其参数，该函数会引发 SEH 异常。

如果你在 SEH `current_exception` 终止处理程序、 `exception_ptr` 异常处理程序或 **`__finally`** 筛选器表达式中调用 **`__except`** 函数，该函数将返回 null **`__except`**。

传输的异常不支持嵌套异常。 如果处理异常时引发了另一个异常，会发生嵌套异常。 如果你捕获嵌套异常， `EXCEPTION_RECORD.ExceptionRecord` 数据成员将指向描述关联异常的 `EXCEPTION_RECORD` 结构链。 `current_exception`函数不支持嵌套异常，因为它返回`exception_ptr`数据成员调零的`ExceptionRecord`对象。

如果你捕获 SEH 异常，则必须管理 `EXCEPTION_RECORD.ExceptionInformation` 数据成员数组中的任何指针所引用的内存。 你必须确保内存在相应的 `exception_ptr` 对象的生存期内有效，并且在 `exception_ptr` 对象删除时释放内存。

你可以将结构化异常 (SE) 转换器函数与传输异常功能结合使用。 如果 SEH 异常转换为 C++ 异常， `current_exception` 函数返回的 `exception_ptr` 将引用转换后的异常，而不是原始 SEH 异常。 `rethrow_exception`函数引发转换后的异常，而不是原始异常。 有关 SE 转换器函数的详细信息，请参阅 _set_se_translator。

rethrow_exception 函数

在 `exception_ptr` 对象中存储捕获的异常后，主线程便可以处理该对象。 在主线程中，调用 `rethrow_exception` 函数，将 `exception_ptr` 对象作为其参数。 `rethrow_exception` 函数从 `exception_ptr` 对象中提取异常，然后在主线程的上下文中引发异常。 如果 函数的 p`rethrow_exception` 参数为 null `exception_ptr`，则该函数引发 std::bad_exception。

提取的异常现在是主线程中的当前异常，因此你可以像处理任何其他异常一样对其进行处理。 如果你捕获异常，可以立即处理它，或使用 **`throw`** 语句将它发送到更高级别的异常处理程序。 否则，不执行任何操作并允许默认系统异常处理程序来终止进程。

make_exception_ptr 函数

`make_exception_ptr` 函数采用类的实例作为其参数，然后返回引用该实例的 `exception_ptr`。 通常，指定异常类对象作为参数传递给 `make_exception_ptr` 函数，但任意类对象都可以是参数。

调用 `make_exception_ptr` 函数等效于引发 C++ 异常、在 **`catch`** 块中捕获它并调用 `current_exception` 函数以返回引用异常的 `exception_ptr` 对象。 Microsoft 实现的 `make_exception_ptr` 函数比调用并捕获异常更高效。

应用程序通常无需`make_exception_ptr`函数，因此，不建议使用此函数。

示例

以下示例从一个线程向另一个线程传输标准 C++ 异常和自定义 C++ 异常。

C++

```
// 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;
}
```

Output

```
exception_ptr 0: Caught an invalid_argument exception.
exception_ptr 1: Caught a  myException exception.
```

要求

标头：`<exception>`

另请参阅

异常处理
/EH （异常处理模型）
/clr（公共语言运行时编译）

Last updated on 2025/11/07

断言和用户提供的消息 (C++)

C++ 语言支持可帮助你调试应用程序的三个错误处理机制：#error 指令、static_assert 关键字和 assert Macro, _assert, _wassert 宏。 所有的三种机制都会发出错误消息，其中两个还会测试软件断言。 软件断言指定在程序的某个特定点应满足的条件。 如果编译时断言失败，编译器将发出诊断消息和编译错误。 如果运行时断言失败，操作系统将发出诊断消息并关闭应用程序。

注解

应用程序的生存期由预处理、编译和运行时阶段组成。 每个错误处理机制都会访问在这三个阶段之一中可用的调试信息。 若要有效地调试，请选择提供有关该阶段的相应信息的机制：

- #error 指令在预处理时有效。 它将无条件地发出用户指定的消息并导致编译因错误而失败。 该消息可包含由预处理器指令操作的文本，但不会计算任何生成的表达式。
- static_assert 声明在编译时有效。 它将测试由用户指定且可以转换为布尔值的整数表达式表示的软件断言。 如果表达式的计算结果为零 (false)，编译器将发出用户指定的消息，并且编译因错误而失败。

  `static_assert` 声明对调试模板尤其有用，因为模板自变量可包含在用户指定的表达式中。
- assert Macro, _assert, _wassert 宏在运行时有效。 它会计算用户指定的表达式，如果结果为零，系统将发出诊断消息并关闭应用程序。 很多其他宏（如 _ASSERT 和 _ASSERTE）与此宏类似，但它们发出不同的系统定义或用户定义的诊断消息。

另请参阅

#error 指令 (C/C++)
assert 宏、_assert、_wassert
_ASSERT、_ASSERTE、_ASSERT_EXPR 宏
static_assert
_STATIC_ASSERT 宏
模板

Last updated on 2025/11/07

static_assert

在编译时测试软件断言。 如果指定的常量表达式为 `false`，则编译器显示指定的消息（如果提供了消息），并且编译失败，错误为 C2338；否则，声明无效。

语法

```
static_assert( constant-expression, string-literal );

static_assert( constant-expression ); // C++17 (Visual Studio 2017 and later)
```

参数

constant-expression
可以转换为布尔值的整型常量表达式。 如果计算出的表达式为零 (false)，则显示 *string-literal* 参数，并且编译失败，并出现错误。 如果表达式不为零 (true)，则 `static_assert` 声明无效。

string-literal
当 *constant-expression* 参数为零时显示的消息。 该消息是编译器的基本字符集中的一个字符串；即，不是多字节或宽字符。

注解

声明的 `static_assert` 参数表示*软件断言*。 软件断言指定在程序的某个特定点应满足的条件。 如果条件为 true，则 `static_assert` 声明无效。 如果条件为 false，则断言失败，编译器在 *string-literal* 参数中显示消息，并且编译失败，出现错误。 在 Visual Studio 2017 及更高版本中，string-literal 参数是可选的。

`static_assert` 声明在编译时测试软件断言。 相反，assert 宏、_assert 和 _wassert 函数在运行时测试软件断言，并产生运行时空间或时间成本。 `static_assert` 声明对调试模板尤其有用，因为模板自变量可包含在 *constant-expression* 中。

当遇到声明时，编译器将检查 `static_assert` 声明是否存在语法错误。 如果编译器不依赖于模板参数，则编译器会立即计算 *constant-expression* 参数。 否则，在对模板进行实例化时，编译器将计算 *constant-expression* 参数。 因此，当遇到声明时，编译器可能一次发布一个诊断消息，而在对模板进行实例化时也是如此。

可以在命名空间、类或块范围中使用 `static_assert` 关键字。 （由于 `static_assert` 关键字可以在命名空间范围内使用，因此，即使它不将新名称引到程序中，但从技术上讲，它也是一个声

明。）

包含命名空间范围的 `static_assert` 说明

在下面的示例中，`static_assert` 声明具有命名空间范围。 由于编译器知道类型 `void *` 的大小，因此可以立即计算表达式。

示例：具有命名空间范围 `static_assert`

C++

```
static_assert(sizeof(void *) == 4, "64-bit code generation is not supported.");
```

具有类范围的 `static_assert` 说明

在下面的示例中，`static_assert` 声明具有类范围。 `static_assert` 验证模板参数是否为*纯旧数据* (POD) 类型。 编译器将在声明 `static_assert` 声明时检查该声明，但不计算 *constant-expression* 参数，直到在 `basic_string` 中实例化 `main()` 类模板。

示例：具有类范围 `static_assert`

C++

```
#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;
}
```

具有块范围的 `static_assert` 说明

在下面的示例中，`static_assert` 声明具有块范围。 `static_assert` 验证 VMPage 结构的大小是否与该系统的虚拟内存页大小相等。

示例： `static_assert` 在块范围内

C++

```
#include <sys/param.h> // defines PAGESIZE
class VMMClient {
public:
    struct VMPage { // ...
           };
    int check_pagesize() {
    static_assert(sizeof(VMPage) == PAGESIZE,
        "Struct VMPage must be the same size as a system virtual memory page.");
    // ...
    }
// ...
};
```

另请参阅

断言和用户提供的消息 (C++)
#error 指令 (C/C++)
assert 宏、_assert、_wassert
模板
ASCII 字符集
声明和定义

Last updated on 2025/11/07

C++中的模块概述

C++20 引入了 *模块*。 *模块*是一组源代码文件，这些文件独立于导入它们的源文件（或更确切地说是翻译单元）进行编译。

模块消除了或减少与头文件使用相关的许多问题。 它们通常会减少编译时间，有时会显著减少。 模块中声明的宏、预处理器指令和非导出名称在模块外部不可见。 它们对导入模块的翻译单元的编译没有影响。 可以按任何顺序导入模块，而无需考虑宏重新定义。 导入翻译单元中的声明不参与导入模块中的重载解析或名称查找。 编译一次模块后，结果将存储在描述所有导出的类型、函数和模板的二进制文件中。 编译器可以比头文件更快地处理该文件。 而且，编译器可以在项目中导入模块的每个位置重复使用它。

可以将模块与头文件并排使用。 C++源文件可以包含`import`模块和`#include`头文件。 在某些情况下，可以将头文件导入为模块，这比使用 `#include` 预处理器处理头文件要快。 建议尽可能在新项目中使用模块，而不是使用头文件。 对于正在开发的大型现有项目，请试验将旧标头转换为模块。 根据能否显著缩短编译时间来确定是否采用。

若要将模块与其他导入标准库的方法进行比较，请参阅 比较标头单元、模块和预编译标头。

从 Visual Studio 2022 版本 17.5 开始，将标准库导入为模块在 Microsoft C++ 编译器中都是标准化和完全实现的。 若要了解如何使用模块导入标准库，请参阅 使用模块导入C++标准库。

单分区模块

单分区模块是包含单个源文件的模块。 模块接口和实现位于同一文件中。

以下单分区模块示例演示了名为 *`Example.ixx`* 的源文件中的简单模块定义。 该 *`.ixx`* 扩展是 Visual Studio 中模块接口文件的默认扩展。 如果要使用不同的扩展，请使用 /interface 开关将其编译为模块接口。 在此示例中，接口文件包含函数定义和声明。 还可以将定义放置在一个或多个单独的模块实现文件中，如后面的示例所示，但这是单分区模块的示例。

该 `export module Example;` 语句指示此文件是调用 `Example`的模块的主接口。 **`export`** 修饰符位于 `int f()` 之前，指示当另一个程序或模块导入 `Example` 时，这个函数是可见的。

C++

```
// Example.ixx
export module Example;

#define ANSWER 42

namespace Example_NS
{
   int f_internal()
   {
```

```
    return ANSWER;
  }

  export int f()
  {
    return f_internal();
  }
}
```

该文件 *MyProgram.cpp* 使用 **import** 来访问由 Example 导出的名称。 命名空间名称 Example_NS 在此处可见，但不是所有成员，因为它们未导出。 此外，宏 ANSWER 不可见，因为宏未导出。

C++

```
// MyProgram.cpp
import std;
import Example;

using namespace std;

int main()
{
   cout << "The result of f() is " << Example_NS::f() << endl; // 42
   // int i = Example_NS::f_internal(); // C2039
   // int j = ANSWER; //C2065
}
```

声明 import 只能在全局范围内显示。 必须使用同一编译器选项编译一个模块和使用该模块的代码。

模块语法

> *module-name*:
>
> *module-name-qualifier-seq* 选择 *identifier*
>
> *module-name-qualifier-seq*:
>
> *identifier* **.**
>
> *module-name-qualifier-seq* *identifier* **.**
>
> *module-partition*:
>
> **:** *module-name*
>
> *module-declaration*:
>
> **export** 选择 **module** *module-name* *module-partition* 选择 *attribute-specifier-seq* 选择 **;**

> `module-import-declaration`:
>
> `export` 选择 `import` `module-name` `attribute-specifier-seq` 选择 `;`
>
> `export` 选择 `import` `module-partition` `attribute-specifier-seq` 选择 `;`
>
> `export` 选择 `import` `header-name` `attribute-specifier-seq` 选择 `;`

实现模块

*模块接口*导出模块名称和构成模块的公共接口的所有命名空间、类型、函数等。
*模块实现*定义模块导出的内容。
以最简单的形式，模块可以是合并模块接口和实现的单个文件。 还可以将实现放在一个或多个单独的模块实现文件中，类似于`.h`和`.cpp`文件的做法。

对于较大的模块，可以将模块的各个部分拆分为称为 *分区的*子模块。 每个分区由导出模块分区名称的模块接口文件组成。 分区也可能具有一个或多个分区实现文件。 整个模块都有一个 *主模块接口*，即模块的公共接口。 如果需要，它可以导出分区接口。

模块由一个或多个 *模块单元*组成。 模块单元是包含模块声明的翻译单元（源文件）。 有几种类型的模块单元：

- *模块接口单元*导出模块名称或模块分区名称。 模块接口单元在其模块声明中具有 `export module` 。
- *模块实现单元*不会导出模块名称或模块分区名称。 顾名思义，它实现模块。
- *主模块接口单元*导出模块名称。 模块中必须有一个且只有一个主要模块接口单元。
- *模块分区接口单元*导出模块分区名称。
- *模块分区实现单元*在其模块声明中具有模块分区名称，但没有`export`关键字。

关键字 **`export`** 仅在接口文件中使用。 实现文件可以 **`import`** 另一个模块，但不能 **`export`** 任何名称。 实现文件可以具有任何扩展名。

模块、命名空间和依赖参数的查找

模块中命名空间的规则与任何其他代码相同。 如果导出命名空间中的声明，则也会隐式导出封闭命名空间（不包括未在该命名空间中显式导出的成员）。 如果显式导出命名空间，则会导出该命名空间定义中的所有声明。

当编译器对导入转换单元中的重载解析进行自变量依赖查找时，它会考虑在定义函数参数类型时在同一转换单元（包括模块接口）中声明的函数。

模块分区

模块分区类似于模块，但以下情况除外：

- 它共享整个模块中所有声明的所有权。
- 分区接口文件导出的所有名称均由主接口文件导入和导出。
- 分区的名称必须以模块名称开头，后跟冒号 （`:`）。
- 任何分区中的声明在整个模块中可见。
- 无需采取特殊预防措施来避免单定义规则（ODR）错误。 可以在一个分区中声明名称（函数、类等），并在另一个分区中定义它。

分区实现文件的开始如下，从C++标准的角度来看，这是一个内部分区：

```
C++
module Example:part1;
```

分区接口文件的开头如下所示：

```
C++
export module Example:part1;
```

若要访问另一个分区中的声明，分区必须导入它。 但它只能使用分区名称，而不能使用模块名称：

```
C++
module Example:part2;
import :part1;
```

主接口单元必须导入并重新导出模块的所有接口分区文件，如下所示：

```
C++
export import :part1;
export import :part2;
```

主接口单元可以导入分区实现文件，但无法导出它们。 不允许这些文件导出任何名称。 此限制使模块能够在模块内部保留实现详细信息。

模块和头文件

可以通过在模块声明前放置指令 `#include` ，将头文件包含在模块源文件中。 这些文件被视为位于 *全局模块片段*中。 模块只能查看其显式包含的标头中的全局模块片段中的名称。 全局模块片段仅包含使用的符号。

```C++
// MyModuleA.cpp

#include "customlib.h"
#include "anotherlib.h"

import std;
import MyModuleB;

//... rest of file
```

可以使用传统的头文件来控制导入的模块：

```C++
// MyProgram.h
#ifdef C_RUNTIME_GLOBALS
import std.compat;
#else
import std;
#endif
```

导入的头文件

某些标头已足够自包含，可以使用 `import` 关键字将其引入。 导入的头文件和导入的模块之间的主要区别在于，头文件中的任何预处理器定义在`import`语句之后立即在执行导入的程序中可见。

```C++
import <vector>;
import "myheader.h";
```

另请参阅

使用模块导入C++标准库
module、 import、、 export
命名模块教程
比较标头单元、模块和预编译标头

Last updated on 2025/11/07

`module`、`import`、`export`

这些`module`、`import`和`export`声明在 C++20 中可用，需要编译器选项/std:c++20或更高版本。有关详细信息，请参阅 C++ 中的模块概述。

`module`

将 `module` 声明放在模块实现文件的开头，以指定文件内容属于命名模块。

```
C++
module ModuleA;
```

`export`

将 `export module` 声明用于默认情况下扩展名为 `.ixx` 的模块主接口文件。 如果要使用不同的扩展，请使用 /interface 开关将其编译为模块接口。

```
C++
export module ModuleA;
```

在接口文件中，对要作为公共接口一部分的名称使用 `export` 修饰符：

```
C++
// ModuleA.ixx

export module ModuleA;

namespace ModuleA_NS
{
   export int f();
   export double d();
   double internal_f(); // not exported
}
```

非导入名称对于导入模块的代码而言不可见：

```
C++
import ModuleA;

int main() {
  ModuleA_NS::f(); // OK
```

```
  ModuleA_NS::d(); // OK
  ModuleA_NS::internal_f(); // Ill-formed: error C2065: 'internal_f': undeclared
identifier
}
```

关键字 `export` 可能不会显示在模块实现文件中。 当 `export` 应用于命名空间名称时，将导出命名空间中的所有名称。

`import`

使用 `import` 声明使模块名称在程序中可见。 `import` 声明必须出现在 `module` 声明之后以及任何 `#include` 指令之后，但必须出现在文件中的任何声明之前。

C++

```
module ModuleA;

#include "custom-lib.h"
import std;
import myModule;

// begin declarations here:
template <class T>
class Baz
{...};
```

备注

仅当 `import` 和 `module` 出现在逻辑行的开头时，它们才会被视为关键字：

C++

```
// 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 专用

在 Microsoft C++ 中，当令牌 `import` 和 `module` 用作宏的参数时，它们始终是标识符而不是关键字。

示例

C++

```
#define foo(...) __VA_ARGS__
foo(
import // Always an identifier, never a keyword
)
```

结束 Microsoft 专用

另请参阅

C++ 中的模块概述
使用模块导入C++标准库

Last updated on 2025/11/07

教程：使用命令行中的模块导入C++标准库

了解如何使用C++库模块导入C++标准库。 这可以加快编译速度，比使用头文件、标头单元或预编译标头（PCH）更可靠。

在本教程中，了解以下内容：

- 如何从命令行将标准库作为模块导入。
- 模块的性能和可用性优势。
- 这两个标准库模块 `std` 及其 `std.compat` 之间的差异。

先决条件

本教程需要 Visual Studio 2022 17.5 或更高版本。

标准库模块简介

头文件语义可能会根据宏定义、包含它们的顺序以及编译速度缓慢而更改。 模块解决这些问题。

现在可以将标准库作为模块导入，而不是作为一堆头文件导入。 这比包括头文件、标头单元或预编译标头（PCH）更快、更可靠。

C++23 标准库引入了两个命名模块：`std` 和 `std.compat`。

- `std` 导出C++标准库命名空间 `std` 中定义的声明和名称，例如 `std::vector`。 它还会导出 C 包装器标头的内容，这些 `<cstdio>` `<cstdlib>` 标头提供类似 `std::printf()` 函数的内容。 全局命名空间中定义的 C 函数（例如 `::printf()`）不会导出。 这改进了包括 C 包装器头文件（如 ）时也包含 C 头文件 的情况，这些头文件将引入 C 全局命名空间版本。 如果导入 `std`，则这不是问题。
- `std.compat` 导出所有内容 `std`，并添加 C 运行时全局命名空间，例如 `::printf`、`::fopen`、`::size_t` `::strlen`，等等。 通过该 `std.compat` 模块，可以更轻松地使用引用全局命名空间中的许多 C 运行时函数/类型的代码库。

编译器在使用 `import std;` 或 `import std.compat;` 时导入整个标准库，并且这个过程比引入单个头文件更快。 例如，使用 `import std;`（或 `import std.compat`）引入整个标准库的速度比 `#include <vector>` 要快。

由于命名模块不会公开宏，因此当您导入 `std` 或 `std.compat` 时，诸如 `offsetof`、`va_arg` 和其他宏都不可用。 有关解决方法，请参阅 名为标准库的模块注意事项 。

关于C++模块

头文件用于在C++中的源文件之间共享声明和定义。 在标准库模块之前，你会通过指令（例如 `#include <vector>`）引入所需部分的标准库。 头文件很脆弱且难以撰写，因为它们的语义可能会根据你包括的顺序或是否定义了某些宏而更改。 它们也缓慢编译，因为它们由包含它们的每个源文件重新处理。

C++20 引入了一个名为 *模块*的新式替代方法。 在 C++23 中，我们能够利用模块支持来引入命名模块来表示标准库。

与头文件一样，模块允许跨源文件共享声明和定义。 但与头文件不同，模块并不脆弱，并且更易于撰写，因为它们的语义不会因宏定义或导入它们的顺序而更改。 编译器处理模块的速度比处理 `#include` 文件快得多，在编译时也使用更少的内存。 命名模块不公开宏定义或专用实现详细信息。

本文演示了使用标准库的新增和最佳方法。 有关使用标准库的替代方法的详细信息，请参阅 比较标头单元、模块和预编译标头。

使用 `std` 导入标准库

以下示例演示如何使用命令行编译器将标准库用作模块。 有关如何在 Visual Studio IDE 中执行此作的信息，请参阅 生成 ISO C++23 标准库模块。

该语句 `import std;` 或 `import std.compat;` 将标准库导入应用程序。 但首先，必须将命名模块的标准库编译为二进制形式。 以下步骤演示如何操作。

示例：如何生成和导入 `std`

1. 打开适用于 VS 的 x86 Native Tools 命令提示符：从 Windows **开始** 菜单中，键入 *x86 本机*，提示应显示在应用列表中。 确保提示为 Visual Studio 2022 版本 17.5 或更高版本。 如果使用错误的提示版本，则会收到错误。 本教程中使用的示例适用于 CMD shell。

2. 创建目录，例如 `%USERPROFILE%\source\repos\STLModules`，并将其设为当前目录。 如果选择没有写入访问权限的目录，则编译过程中会出现错误。

3. 用以下命令编译名为`std`的模块：

   Windows 命令提示符

   ```
   cl /std:c++latest /EHsc /nologo /W4 /c "%VCToolsInstallDir%\modules\std.ixx"
   ```

   如果收到错误，请确保使用正确的命令提示符版本。

使用与导入该生成模块的代码相同的编译器设置来编译`std`命名的模块。 如果您有一个多项目解决方案，可以编译名为 module 的标准库，然后使用编译器选项 /reference 在所有项目中引用它。

使用前面的编译器命令，编译器输出两个文件：

- `std.ifc` 是编译器查阅处理 `import std;` 语句的命名模块接口的已编译二进制表示形式。 这是仅限编译时使用的工件。 它不会随应用程序一起交付。
- `std.obj` 包含命名模块的实现。 编译示例应用以静态方式将标准库中使用的功能链接到应用程序时，请 `std.obj` 添加到命令行。

此示例中的关键命令行开关包括：

展开表

开关	Meaning
/std:c++latest	使用最新版本C++语言标准和库。 虽然在`/std:c++20`版本下提供了模块支持，但需要最新的标准库才能获取对标准库命名模块的支持。
/EHsc	使用C++异常处理，但标记 `extern "C"`的函数除外。
/W4	通常建议使用 `/W4` ，尤其是对于新项目，因为它启用所有级别 1、级别 2、级别 3 和大多数级别 4（信息性）警告，这有助于提前捕获潜在问题。 它实质上提供了类似于 lint 的警告，可帮助确保尽可能少的难以查找的代码缺陷。
/c	编译时不进行链接，因为我们此时只是生成名为二进制的模块接口。

可以使用以下开关控制对象文件名和命名模块接口文件名：

- /Fo 设置对象文件的名称。 例如，`/Fo:"somethingelse"`。 默认情况下，编译器使用与要编译的模块源文件（`.ixx`）相同的对象文件的名称。 在此示例中，对象文件名 `std.obj` 默认为，因为我们正在编译模块文件 `std.ixx`。
- /ifcOutput 设置命名模块接口文件的名称（`.ifc`）。 例如，`/ifcOutput "somethingelse.ifc"`。 默认情况下，编译器将模块接口文件 （`.ifc`） 的名称与编译的模块源文件 （`.ixx`） 使用相同的名称。 在此示例中，生成的 `ifc` 文件 `std.ifc` 默认为因为我们正在编译模块文件 `std.ixx`。

4. 首先创建一个名为`importExample.cpp`的文件，文件中包含以下内容，然后导入您构建的`std`库。

C++

```
// requires /std:c++latest
```

```
import std;

int main()
{
    std::cout << "Import the STL library for best performance\n";
    std::vector<int> v{5, 5, 5};
    for (const auto& e : v)
    {
        std::cout << e;
    }
}
```

在前面的代码中， `import std;` 替换 `#include <vector>` 和 `#include <iostream>`。 该语句 `import std;` 使所有标准库都可用于一个语句。 导入整个标准库通常比处理单个标准库头文件快得多，例如 `#include <vector>`。

5. 使用与上一步相同的目录中的以下命令编译示例：

Windows 命令提示符

```
cl /c /std:c++latest /EHsc /nologo /W4 /reference "std=std.ifc"
importExample.cpp
link importExample.obj std.obj
```

不需要在此示例中的命令行上指定 `/reference "std=std.ifc"` ，因为编译器会自动查找 `.ifc` 与语句指定的 `import` 模块名称匹配的文件。 当编译器遇到 `import std;` 时，如果它位于与源代码相同的目录中，编译器可以找到 `std.ifc`。 `.ifc`如果文件位于与源代码不同的目录中，请使用/reference编译器开关来引用它。

在此示例中，编译源代码并将模块的实现链接到应用程序是单独的步骤。 他们不必非得这样。 你可以使用 `cl /std:c++latest /EHsc /nologo /W4 /reference "std=std.ifc" importExample.cpp std.obj` 在一步中进行编译和链接。 但是，单独生成和链接可能很方便，因为在生成的链接步骤中，只需生成一次名为模块的标准库，然后就可以从项目或多个项目引用它。

如果您构建单个项目，可以通过将 `"%VCToolsInstallDir%\modules\std.ixx"` 添加到命令行来同时执行构建 `std` 标准库模块的步骤和构建您的应用程序的步骤。 将其置于使用`.cpp`模块的任何`std`文件之前。

默认情况下，输出可执行文件的名称取自第一个输入文件。 `/Fe`使用编译器选项指定所需的可执行文件名称。 本教程演示如何将 `std` 命名模块单独编译为步骤，因为标准库的命名模块只需编译一次，然后便可在您的项目或多个项目中引用。 但是，将所有东西一起构建可能很方便，如以下命令行所示：

Windows 命令提示符

```
cl /FeimportExample /std:c++latest /EHsc /nologo /W4
"%VCToolsInstallDir%\modules\std.ixx" importExample.cpp
```

根据上一个命令行，编译器将生成名为 `importExample.exe` 的可执行文件。 运行它时，它会生成以下输出：

输出

```
Import the STL library for best performance
555
```

使用 `std.compat` 导入标准库和全局 C 函数

C++标准库包括 ISO C 标准库。 该`std.compat`模块提供`std`模块的所有功能，例如 `std::vector`、 `std::cout`、 `std::printf`、 `std::scanf`等等。 但它还提供这些函数的全局命名空间版本，例如`::printf`、 `::scanf`、 `::fopen` `::size_t`等等。

命名 `std.compat` 模块是一个兼容性层，用于轻松迁移引用全局命名空间中的 C 运行时函数的现有代码。 如果要避免向全局命名空间添加名称，请使用 `import std;`。 如果需要轻松迁移使用许多不限定的（全局命名空间）C 运行时函数的代码库，请使用 `import std.compat;`。 这提供了全局命名空间 C 运行时名称，因此无需使用 `std::` 限定所有全局名称。 如果没有任何使用全局命名空间 C 运行时函数的现有代码，则无需使用 `import std.compat;`。 如果在代码中只调用了几个 C 运行时函数，则最好使用 `import std;` 并使用 `std::` 来限定那些需要的少数全局命名空间 C 运行时名称。 例如， `std::printf()`。 如果在尝试编译代码时看到类似错误 `error C3861: 'printf': identifier not found` ，请考虑使用 `import std.compat;` 导入全局命名空间 C 运行时函数。

示例：如何生成和导入 `std.compat`

在使用 `import std.compat;` 之前，必须编译在源代码表单中找到的 `std.compat.ixx`模块接口文件。 Visual Studio 提供模块的源代码，以便可以使用与项目匹配的编译器设置编译模块。 这些步骤类似于生成 `std` 命名模块。 首先构建 `std` 命名的模块，因为 `std.compat` 依赖于它。

1. 打开 VS 的本机工具命令提示符：在 Windows **开始** 菜单中，键入 *x86 本机* ，提示应显示在应用列表中。 确保提示为 Visual Studio 2022 版本 17.5 或更高版本。 如果使用错误版本的提示，则会收到编译器错误。

2. 创建一个目录以尝试此示例，例如 `%USERPROFILE%\source\repos\STLModules`，并将其设为当前目录。 如果选择一个没有写入权限的目录，则会遇到错误。

3. 使用以下命令编译`std`和`std.compat`命名的模块：

Windows 命令提示符

```
cl /std:c++latest /EHsc /nologo /W4 /c "%VCToolsInstallDir%\modules\std.ixx"
"%VCToolsInstallDir%\modules\std.compat.ixx"
```

您应该使用相同的编译器设置来编译 `std` 和 `std.compat`，这些编译器设置也将用于导入它们的代码。 如果您有一个多项目解决方案，可以先编译一次，然后使用 /reference 编译器选项在所有项目中引用这些解决方案。

如果收到错误，请确保使用正确的命令提示符版本。

编译器输出前两个步骤中的四个文件：

- `std.ifc` 是编译器查阅以处理 `import std;` 语句的已编译二进制命名模块接口。 编译器还会查阅 `std.ifc` 以处理 `import std.compat;`，因为 `std.compat` 是基于 `std`。 这是仅在编译时使用的工件。 它不会随应用程序一起交付。
- `std.obj` 包含标准库的实现。
- `std.compat.ifc` 是编译器查阅以处理 `import std.compat;` 语句的已编译二进制命名模块接口。 这是仅限编译时使用的工件。 它不会随应用程序一起交付。
- `std.compat.obj` 包含实现。 但是，大部分实现由 `std.obj` 提供。 编译示例应用时，请 `std.obj` 添加到命令行，以静态方式将使用的功能从标准库链接到应用程序。

可以使用以下开关控制对象文件名和命名模块接口文件名：

- /Fo 设置对象文件的名称。 例如，`/Fo:"somethingelse"`。 默认情况下，编译器使用与要编译的模块源文件（`.ixx`）相同的对象文件的名称。 在本示例中，对象文件名是`std.obj` `std.compat.obj`默认的，因为我们正在编译模块文件和 `std.ixx` `std.compat.ixx`。
- /ifcOutput 设置命名模块接口文件的名称（`.ifc`）。 例如，`/ifcOutput "somethingelse.ifc"`。 默认情况下，编译器将模块接口文件 （`.ifc`） 的名称与编译的模块源文件 （`.ixx`） 使用相同的名称。 在此示例中，默认生成的 `ifc` 文件是 `std.ifc` 和 `std.compat.ifc`，因为我们正在编译模块文件 `std.ixx` 和 `std.compat.ixx`。

4. 首先创建一个名为`stdCompatExample.cpp`的文件，并在其中填写以下内容以导入`std.compat`库：

C++

```
import std.compat;

int main()
{
    printf("Import std.compat to get global names like printf()\n");
```

```
    std::vector<int> v{5, 5, 5};
    for (const auto& e : v)
    {
        printf("%i", e);
    }
}
```

在前面的代码中， `import std.compat;` 替换 `#include <cstdio>` 和 `#include <vector>`。 该语句 `import std.compat;` 使标准库和 C 运行时函数可用于一个语句。 导入此命名模块（包括 C++ 标准库和 C 运行时库的全局命名空间函数）比处理单个模块（例如 `#include` 模块 `#include <vector>`）要快。

5. 使用以下命令编译示例：

Windows 命令提示符

```
cl /std:c++latest /EHsc /nologo /W4 stdCompatExample.cpp
link stdCompatExample.obj std.obj std.compat.obj
```

我们不必在命令行上指定 `std.compat.ifc` ，因为编译器会自动查找与 `.ifc` 语句中的 `import` 模块名称匹配的文件。 当编译器遇到 `import std.compat;` 时，它会找到 `std.compat.ifc`，因为我们将其放在与源代码相同的目录中，这样就不需要在命令行上指定它了。 `.ifc`如果文件位于与源代码不同的目录中，或者具有其他名称，请使用/reference编译器开关来引用它。

导入`std.compat`时，必须同时链接`std.compat`和`std.obj`，因为`std.compat`使用`std.obj`中的代码。

如果要构建单个项目，可以通过将`"%VCToolsInstallDir%\modules\std.ixx"`和`"%VCToolsInstallDir%\modules\std.compat.ixx"`按顺序添加到命令行，来组合构建`std`和`std.compat`标准库命名模块的步骤。 本教程演示了将标准库模块构建为单独的步骤，因为只需生成一次名为模块的标准库，然后就可以从项目或多个项目中引用它们。 但是，如果一次性生成它们很方便，请确保将它们放在使用它们的任何 `.cpp` 文件之前，并指定 `/Fe` 为生成 `exe` 命名，如以下示例所示：

Windows 命令提示符

```
cl /c /FestdCompatExample /std:c++latest /EHsc /nologo /W4
"%VCToolsInstallDir%\modules\std.ixx"
"%VCToolsInstallDir%\modules\std.compat.ixx" stdCompatExample.cpp
link stdCompatExample.obj std.obj std.compat.obj
```

在此示例中，编译源代码并将模块的实现链接到应用程序是单独的步骤。 他们不必非得这样。 你可以使用 `cl /std:c++latest /EHsc /nologo /W4 stdCompatExample.cpp std.obj`

`std.compat.obj` 在一步中进行编译和链接。 但是，单独生成和链接可能很方便，因为在生成的链接步骤中，只需生成一次名为模块的标准库，然后就可以从项目或多个项目引用它们。

上一个编译器命令生成一个名为`stdCompatExample.exe`的可执行文件。 运行它时，它会生成以下输出：

输出

```
Import std.compat to get global names like printf()
555
```

标准库中命名模块的注意事项

命名模块的版本控制与标头的版本控制相同。 `.ixx`命名的模块文件与标头文件一起安装，例如： `"%VCToolsInstallDir%\modules\std.ixx"`文件，在撰写本文时使用的工具版本中，解析为`C:\Program Files\Microsoft Visual Studio\2022\Enterprise\VC\Tools\MSVC\14.38.33130\modules\std.ixx`。 挑选命名模块的版本, 方法和选择头文件版本相同——通过你引用它们的目录来实现。

不要将导入的头文件单元和命名模块混合使用。 例如，不要 `import <vector>;` 和 `import std;` 在同一文件中。

不要混合和匹配导入C++标准库头文件和命名模块 `std` 或 `std.compat`。 例如，不要 `#include <vector>` 和 `import std;` 在同一文件中。 但是，可以在同一文件中包括 C 标头和导入命名模块。 例如，可以 `import std;` 和 `#include <math.h>` 在同一文件中使用。 请勿包括C++标准库版本 `<cmath>`。

不必担心重复导入模块。 也就是说，模块中不需要 `#ifndef` 样式标头防护。 编译器知道它是否已导入命名模块，并忽略重复尝试执行此作。

如果您需要使用 `assert()` 宏，请执行 `#include <assert.h>`。

如果需要使用 `errno` 宏， `#include <errno.h>`。 由于命名模块不公开宏，因此例如在需要检查来自 `<math.h>` 的错误时，这是一个解决方法。

宏（例如 `NAN`， `INFINITY` 和 `INT_MIN`）由 `<limits.h>` 定义，你可以包括这些宏。 但是，如果你`import std;`，可以使用`std::numeric_limits<double>::quiet_NaN()`和`std::numeric_limits<double>::infinity()`而不是`NAN`和`INFINITY`，并使用`std::numeric_limits<int>::min()`而不是`INT_MIN`。

概要

在本教程中，你已使用模块导入标准库。 接下来，了解如何在 命名模块教程中创建和导入自己的模块，C++。

另请参阅

比较标头单元、模块和预编译标头
C++中的模块概述
Visual Studio 中的C++模块教程
将项目移动到名为“Modules”的C++

Last updated on 2025/11/07

命名模块教程 (C++)

项目 • 2023/04/03

本教程介绍如何创建 C++20 模块。 模块替换头文件。 你将了解模块是如何改进头文件的。

本教程介绍如何执行下列操作：

- 创建和导入模块
- 创建主模块接口单元
- 创建模块分区文件
- 创建模块单元实现文件

先决条件

本教程需要 Visual Studio 2022 17.1.0 或更高版本。

当你生成本教程中的代码示例时，可能会遇到 IntelliSense 错误。 IntelliSense 引擎的工作进度即将赶上编译器的进度。 IntelliSense 错误可被忽略，不会阻止生成代码示例。 若要跟踪 IntelliSense 工作的进度，请查看此问题。

什么是 C++ 模块

头文件是 C++ 中源文件之间共享声明和定义的方式。 头文件很脆弱，难以撰写。 它们的编译方式可能会有所不同，具体取决于你包括它们的顺序，或者针对已定义或未定义的宏。 它们可能会减缓编译进度，因为它们要对包含它们的每个源文件进行重新处理。

C++20 引入了用于组件化 C++ 程序的一种新式方法：模块。

与头文件一样，模块允许跨源文件共享声明和定义。 但与头文件不同的是，模块不会泄露宏定义或专用实现详细信息。

模块更易于编写，因为它们的语义不会因为宏定义或导入的其他内容、导入顺序等而发生更改。 它们还可以更轻松地控制对使用者可见的内容。

模块提供了头文件所不具备的额外安全保证。 编译器和链接器协同工作以防止可能出现的名称冲突问题，并提供更强大的一个定义规则 (ODR) 保证。

强所有权模型可避免在链接时名称之间发生冲突，因为链接器会将导出的名称附加到导出它们的模块。 这种模式使 Microsoft Visual C++ 编译器能够阻止通过链接报告同一程序中类似名称的不同模块而导致的未定义行为。 有关详细信息，请参阅强所有权。

模块由编译为二进制文件的一个或多个源代码文件组成。 二进制文件描述了模块中的所有导出类型、函数和模板。 当源文件导入模块时，编译器会读入包含模块内容的二进制文件。 读取二进制文件比处理头文件快得多。 此外，编译器每次导入模块时都会重复使用二进制文件，从而节省更多时间。 由于一个模块只生成一次，而不是每次导入时都会生成，因此生成时间可以减少，有时会大幅减少。

更重要的是，模块没有头文件存在的易碎性问题。 导入模块不会更改模块的语义，也不会更改任何其他导入的模块的语义。 在模块中声明的宏、预处理器指令和非导出名称对导入它的源文件是不可见的。 可以按任意顺序导入模块，它不会更改模块的含义。

模块可以与头文件并行使用。 如果要迁移代码库以使用模块，此功能很方便，因为可以分阶段执行此操作。

在某些情况下，可以将头文件作为标头单元，而不是 `#include` 文件导入。 标头单元是预编译头文件 (PCH) 的推荐替代方法。 与共享 PCH 文件相比，它们更易于设置和使用，但它们提供类似的性能优势。 有关详细信息，请参阅演练：在 Microsoft Visual C++ 中生成和导入标头单元。

代码可以使用对静态库项目的项目到项目引用，自动使用同一项目中的模块或任何引用的项目。

创建项目

生成一个简单的项目时，我们将了解模块的各个方面。 项目将使用模块而不是头文件实现 API。

在 Visual Studio 2022 或更高版本中，选择“ **创建新项目** ”，然后选择 “**控制台应用** (for C++) 项目类型”。 如果此项目类型不可用，你可能在安装 Visual Studio 时没有选择“使用 C++ 的桌面开发”工作负载。 可以使用 Visual Studio 安装程序来添加 C++ 工作负载。

为新项目指定名称 `ModulesTutorial` 并创建项目。

由于模块是 C++20 功能，因此请使用 /std:c++20 或 /std:c++latest 编译器选项。 在**解决方案资源管理器**中，右键单击项目名称 `ModulesTutorial`，然后选择“**属性**”。 在项目的“属性页”对话框中，将“配置”更改为“所有配置”，将“平台”更改为“所有平台”。 在左侧的树状视图窗格中，选择“配置属性”>“常规”。 选择“C++ 语言标准”属性。 使用下拉菜单将属性值更改为“ISO C++20 标准(/std:c++20)”。 选择“确定”以接受更改。

创建主模块接口单元

模块由一个或多个文件组成。 其中一个文件必须是“主模块接口单元”。 它定义模块导出的内容;也就是说，模块的导入程序将看到的内容。 每个模块只能有一个主模块接口单元。

若要添加主模块接口单元，请在“**解决方案资源管理器**”中右键单击“**源文件**”，然后选择“**添加**>**模块**”。

在显示的“添加新项”对话框中，为新模块指定名称 `BasicPlane.Figures.ixx`，然后选择“添加”。

创建的模块文件的默认内容有两行：

```
C++

export module BasicPlane;

export void MyFunc();
```

`export module`第一行中的关键字声明此文件是模块接口单元。 此处有一个微妙之处：对于每个命名模块，必须恰好有一个模块接口单元，且未指定任何模块分区。 该模块单元称为“主模块接口单元”。

主模块接口单元用于声明在源文件导入模块时要公开的函数、类型、模板、其他模块和模块分区。 模块可以包含多个文件，但只有主模块接口文件标识要公开的内容。

将文件的内容 `BasicPlane.Figures.ixx` 替换为：

```
C++

export module BasicPlane.Figures; // the export module keywords mark this
file as a primary module interface unit
```

此行将此文件标识为主模块接口，并为模块指定一个名称：`BasicPlane.Figures`。 模块名称中的句点对编译器没有特殊含义。 句点可用于传达模块的组织方式。 如果你有多个模块文件一起工作，可以使用句点来指示分离关注点。 在本教程中，我们将使用句点来指示 API 的不同功能区域。

此名称也是“命名模块”中的“命名”的由来。 属于此模块的文件使用此名称将自己标识为命名模块的一部分。 命名模块是具有相同模块名称的模块单元的集合。

在进一步讨论之前，我们应该先谈谈我们要实现的 API。 它会影响我们接下来的选择。 API 表示不同的形状。 在本示例中，我们只打算提供几个形状：`Point` 和 `Rectangle`。 `Point` 用作更复杂的形状的一部分，例如 `Rectangle`。

为了说明模块的一些功能，我们将此 API 分成几个部分。 其中一个部分是 `Point` API。 另一个部分是 `Rectangle`。 假设此 API 将变得更为复杂。 这种划分对于分离关注点或简化代码维护很有用。

到目前为止，我们创建了将公开此 API 的主模块接口。 现在，我们来生成 `Point` API。 我们希望它成为此模块的一部分。 出于逻辑组织和潜在的生成效率的原因，我们希望使这一部分的 API 容易被理解。 为此，我们将创建“模块分区”文件。

模块分区文件是模块的一个部分，或者说是组件。 它的独特之处在于，它可以被视为模块的一个单独的部分，但只能在模块内。 模块分区不能在模块之外使用。 模块分区可用于将模块实现划分为可管理的部分。

将分区导入主模块时，其所有声明都会对主模块可见，无论它们是否已导出。 可以将分区导入到属于命名模块的任何分区接口、主模块接口或模块单元中。

创建模块分区文件

`Point` 模块分区

若要创建模块分区文件，**请在解决方案资源管理器**右键单击“**源文件**”，然后选择“**添加**>**模块**”。 将文件命名为 *`BasicPlane.Figures-Point.ixx`*，然后选择“添加”。

由于它是一个模块分区文件，因此我们已将一个连字符和分区的名称添加到模块名称中。 此约定有助于编译器在命令行情况下使用，因为编译器使用基于模块名称的名称查找规则来查找分区的已 *`.ifc`* 编译文件。 这样，你不需要提供显式 `/reference` 命令行参数即可查找属于模块的分区。 它还有助于按名称组织属于某个模块的文件，因为可以轻松查看哪些文件属于哪些模块。

将 *`BasicPlane.Figures-Point.ixx`* 的内容替换为：

```C++
export module BasicPlane.Figures:Point; // defines a module partition,
Point, that's part of the module BasicPlane.Figures

export struct Point
{
    int x, y;
};
```

文件以 `export module` 开头。 这些关键字也是主模块接口的开始方式。 此文件的不同是冒号 (`:`) 后跟模块名称，后跟分区名称。 此命名约定将文件标识为“模块分区”。 它定义了分区的模块接口，因此它不被视为主模块接口。

名称 `BasicPlane.Figures:Point` 将此分区标识为模块 `BasicPlane.Figures` 的一部分。 (请记住，名称中的句点对编译器) 没有特殊意义。 冒号指示此文件包含一个名为 `Point` 的模块分区，该分区属于模块 `BasicPlane.Figures`。 我们可以将此分区导入到属于此命名模块的其他文件中。

在此文件中， `export` 关键字使 `struct Point` 对使用者可见。

`Rectangle` 模块分区

我们将定义的下一个分区是 `Rectangle`。 使用与之前相同的步骤创建另一个模块文件：在“**解决方案资源管理器**”中，右键单击“**源文件**”，然后选择“**添加**>**模块**”。 将文件命名为“*`BasicPlane.Figures-Rectangle.ixx`*”，并选择“添加”。

将 *`BasicPlane.Figures-Rectangle.ixx`* 的内容替换为：

```C++
export module BasicPlane.Figures:Rectangle; // defines the module partition
Rectangle

import :Point;

export struct Rectangle // make this struct visible to importers
{
    Point ul, lr;
};

// These functions are declared, but will
// be defined in a module implementation file
export int area(const Rectangle& r);
export int height(const Rectangle& r);
export int width(const Rectangle& r);
```

文件以 `export module BasicPlane.Figures:Rectangle;` 开头，声明属于模块 的模块 `BasicPlane.Figures`分区。 添加到模块名称的 `:Rectangle` 将其定义为模块 `BasicPlane.Figures` 的分区。 它可以被单独导入到属于此命名模块的任何模块文件中。

接下来，`import :Point;` 演示了如何导入模块分区。 `import` 语句使模块分区中的所有导出类型、函数和模板对模块可见。 你不需要指定模块名称。 编译器知道此文件属于 `BasicPlane.Figures` 模块，因为 `export module BasicPlane.Figures:Rectangle;` 位于文件的顶部。

接下来，代码导出 `struct Rectangle` 的定义以及一些返回矩形的各种属性的函数的声明。 `export` 关键字指示是否使它前面的内容对模块的使用者可见。 它用于使函数 `area`、`height` 和 `width` 在模块之外可见。

模块分区中的所有定义和声明都对导入模块单元可见，无论它们是否具有 `export` 关键字。 `export` 关键字控制在主模块接口中导出分区时，定义、声明或 typedef 是否在模块之外可见。

名称通过几种方式对模块使用者可见：

- 将关键字 `export` 放在要导出的每个类型、函数等的前面。
- 例如，如果将放在 `export` 命名空间前面， `export namespace N { ... }`则会导出大括号中定义的所有内容。 但是，如果在模块中的其他位置定义 `namespace N { struct S {...};}`，则 `struct S` 不可用于模块的使用者。 它之所以不可用，是因为命名空间声明不在 `export` 的前面，尽管有另一个同名的命名空间在前面。
- 如果不应导出类型、函数等，则省略 `export` 关键字。 它将对属于模块的其他文件可见，但对模块的导入程序不可见。
- 使用 `module :private;` 标记专用模块分区的开头。 专用模块分区是模块的一部分，其中声明仅对该文件可见。 它们对导入此模块的文件或属于此模块的其他文件不可见。 将其视为文件本地静态部分。 此部分仅在文件中可见。
- 若要使导入的模块或模块分区可见，请使用 `export import`。 下一部分提供了一个示例。

撰写模块分区

现在我们定义了 API 的两个部分，让我们将它们组合在一起，以便导入此模块的文件可以作为一个整体访问它们。

所有模块分区都必须作为它们所属的模块定义的一部分公开。 分区是在主模块接口中公开的。 打开 *`BasicPlane.Figures.ixx`* 文件，该文件定义了主模块接口。 将其内容替换为：

```C++
export module BasicPlane.Figures; // keywords export module marks this as a primary module interface unit

export import :Point; // bring in the Point partition, and export it to consumers of this module
export import :Rectangle; // bring in the Rectangle partition, and export it to consumers of this module
```

以 开头 `export import` 的两行在这里是新的。 如此组合时，这两个关键字指示编译器导入指定的模块，并使该模块的使用者可见。 在这种情况下，模块名称中的冒号 (`:`) 表示我们正在导入模块分区。

导入的名称不包括完整的模块名称。 例如，`:Point` 分区被声明为 `export module BasicPlane.Figures:Point`。 然而，我们在此处导入的是 `:Point`。 由于我们位于模块 的主模块 `BasicPlane.Figures`接口文件中，因此隐含模块名称，并且仅指定分区名称。

到目前为止，我们定义了主模块接口，它公开了我们想要提供的 API 图面。 但我们仅声明了 `area()`、`height()` 或 `width()`，没有定义它们。 接下来，我们将通过创建模块实现文件来执行此操作。

创建模块单元实现文件

模块单元实现文件不以 *`.ixx`* 扩展名结尾，它们是普通 *`.cpp`* 文件。 通过在“解决方案资源管理器”中右键单击“源文件”，选择“添加”>“新建项”，然后选择“C++ 文件(.cpp)”来创建源文件，从而添加一个模块单元实现文件。 为新文件指定名称 *`BasicPlane.Figures-Rectangle.cpp`*，然后选择“添加”。

模块分区实现文件的命名约定遵循分区的命名约定。 但它具有 *`.cpp`* 扩展名，因为它是一个实现文件。

将 `BasicPlane.Figures-Rectangle.cpp` 文件的内容替换为：

```C++
module;

// global module fragment area. Put #include directives here

module BasicPlane.Figures:Rectangle;

int area(const Rectangle& r) { return width(r) * height(r); }
int height(const Rectangle& r) { return r.ul.y - r.lr.y; }
int width(const Rectangle& r) { return r.lr.x - r.ul.x; }
```

此文件以 `module;` 开头，其中引入了模块的一个特殊区域，称为 *全局模块片段*。 它位于命名模块的代码的前面，你可以在其中使用预处理器指令，例如 `#include`。 全局模块片段中的代码不由模块接口拥有或导出。

当你包含一个头文件时，通常不希望将其视为模块的一个导出部分。 你通常将头文件作为一个不应属于模块接口的实现详细信息包含在内。 可能有一些高级的案例需要这样做，但一般来说你不会这样做。 不会为全局模块片段中的 *`.ifc`* 指令生成单独的元数据（`#include` 文件）。 全局模块片段提供了一个很好的位置来包含头文件，如 *`windows.h`* 或在 Linux 上为 *`unistd.h`*。

我们生成的模块实现文件不包含任何库，因为它不需要它们作为其实现的一部分。 但是，如果这样做，此区域就是将执行 `#include` 指令的位置。

行 `module BasicPlane.Figures:Rectangle;` 指示此文件是命名模块 `BasicPlane.Figures`的一部分。 编译器会自动将主模块接口公开的类型和函数引入此文件中。 模块实现单元在其模块声明中的 关键字之前`module`没有 `export` 关键字。

接下来是函数 `area()`、`height()` 和 `width()` 的定义。 它们已在 *`BasicPlane.Figures-Rectangle.ixx`* 的 `Rectangle` 分区中声明。 由于此模块的主模块接口导入了 `Point` 和 `Rectangle` 模块分区，因此这些类型在模块单元实现文件中可见。 模块实现单元有一个有趣的功能：编译器会自动使相应模块主接口中的所有内容对文件可见。 不需要 `imports <module-name>`。

你在实现单元中声明的任何内容都只对它所属的模块可见。

导入模块

现在，我们将使用已定义的模块。 打开 *`ModulesTutorial.cpp`* 文件。 它是作为项目的一部分自动创建的。 它目前包含函数 `main()`。 将其内容替换为：

C++

```
#include <iostream>

import BasicPlane.Figures;

int main()
{
    Rectangle r{ {1,8}, {11,3} };

    std::cout << "area: " << area(r) << '\n';
    std::cout << "width: " << width(r) << '\n';

    return 0;
}
```

语句 `import BasicPlane.Figures;` 使模块中导出的所有函数和类型 `BasicPlane.Figures` 对此文件可见。 它可以在任何 `#include` 指令之前或之后出现。

然后，应用使用模块中的类型和函数输出定义的矩形的区域和宽度：

输出

```
area: 50
width: 10
```

模块剖析

现在，让我们更详细地了解各种模块文件。

主模块接口

模块由一个或多个文件组成。 其中一个文件定义了导入程序将看到的接口。 此文件包含“主模块接口”。 每个模块只能有一个主模块接口。 如前所述，导出的模块接口单元未指定模块分区。

默认情况下，它具有 `.ixx` 扩展名。 但是，你可以将具有任何扩展名的源文件视为模块接口文件。 为此，请将源文件属性页的“**高级**”选项卡中的“**编译**为”属性设置为“**编译为模块**” (/interface)：

模块接口定义文件的基本大纲如下：

```C++
module; // optional. Defines the beginning of the global module fragment

// #include directives go here but only apply to this file and
// aren't shared with other module implementation files.
// Macro definitions aren't visible outside this file, or to importers.
// import statements aren't allowed here. They go in the module preamble,
below.

export module [module-name]; // Required. Marks the beginning of the module
preamble

// import statements go here. They're available to all files that belong to
the named module
// Put #includes in in the global module fragment, above

// After any import statements, the module purview begins here
// Put exported functions, types, and templates here

module :private; // optional. The start of the private module partition.

// Everything after this point is visible only within this file, and isn't
// visible to any of the other files that belong to the named module.
```

此文件必须以 `module;` 开头来指示全局模块片段的开头，或以 `export module [module-name];` 开头来指示“模块 purview”的开头。

模块 purview 是你想要从模块中公开的函数、类型、模板等的位置。

它也是可以通过 `export import` 关键字公开其他模块或模块分区的位置，如 *`BasicPlane.Figures.ixx`* 文件中所示。

主接口文件必须直接或间接导出为模块定义的所有接口分区，否则程序的格式不正确。

在专用模块分区中，可以放置你希望仅在此文件中可见的内容。

模块接口单元在关键字 `module` 前面加上关键字 `export`。

有关模块语法的更深入研究，请参阅模块。

模块实现单元

模块实现单元属于命名模块。 它们所属的命名模块由 文件中的 `module [module-name]` 语句指示。 模块实现单元提供了实现详细信息，出于代码安全或其他原因，你不需要放入主模块接口或模块分区文件中。

模块实现单元可用于将大型模块分解为较小的部分，从而缩短生成时间。 最佳做法部分简要介绍了此方法。

模块实现单元文件具有 `.cpp` 扩展名。 模块实现单元文件的基本大纲如下：

```
C++

// optional #include or import statements. These only apply to this file
// imports in the associated module's interface are automatically available
to this file

module [module-name]; // required. Identifies which named module this
implementation unit belongs to

// implementation
```

模块分区文件

模块分区提供了一种将模块组件化为不同部分或 *分区*的方法。 模块分区只应在属于命名模块的文件中导入。 无法在命名模块之外导入它们。

分区具有接口文件和零个或多个实现文件。 模块分区共享整个模块中所有声明的所有权。

分区接口文件导出的所有名称都必须由主接口文件导入并重新导出 (`export import`)。 分区的名称必须以模块名称开头，后跟一个冒号，然后是分区的名称。

分区接口文件的基本大纲如下：

```
C++

module; // optional. Defines the beginning of the global module fragment

// This is where #include directives go. They only apply to this file and
aren't shared
// with other module implementation files.
// Macro definitions aren't visible outside of this file or to importers
// import statements aren't allowed here. They go in the module preamble,
below

export module [Module-name]:[Partition name]; // Required. Marks the
beginning of the module preamble

// import statements go here.
// To access declarations in another partition, import the partition. Only
use the partition name, not the module name.
// For example, import :Point;
// #include directives don't go here. The recommended place is in the global
module fragment, above
```

```
// export imports statements go here

// after import, export import statements, the module purview begins
// put exported functions, types, and templates for the partition here

module :private; // optional. Everything after this point is visible only
within this file, and isn't
                         // visible to any of the other files that belong to
the named module.
...
```

模块最佳做法

必须使用相同的编译器选项编译导入它的模块和代码。

模块命名

- 可以在模块名称中使用句点（“.”），但它们对编译器没有特殊意义。 使用它们向模块的用户传达意义。 例如，以库或项目的顶级命名空间开始。 以描述模块功能的名称结束。 `BasicPlane.Figures` 旨在传达一个关于几何平面的 API，特别是可在平面上表示的图形。
- 包含模块主接口的文件的名称通常是模块的名称。 例如，如果模块名称为 `BasicPlane.Figures`，则包含主接口的文件的名称将为 *`BasicPlane.Figures.ixx`*。
- 模块分区文件的名称通常是 `<primary-module-name>-<module-partition-name>`，其中，模块的名称后跟一个连字符（“-”），然后是分区的名称。 例如：*`BasicPlane.Figures-Rectangle.ixx`*

如果要从命令行生成，并且对模块分区使用此命名约定，则无需为每个模块分区文件显式添加 `/reference`。 编译器将根据模块的名称自动查找它们。 编译的分区文件的名称 (以 *`.ifc`* 扩展名结尾) 从模块名称生成。 请考虑模块名称 `BasicPlane.Figures:Rectangle`：编译器将预期 的 `Rectangle` 相应已编译分区文件名为 `BasicPlane.Figures-Rectangle.ifc`。 编译器使用此命名方案，通过自动查找分区的接口单元文件，更轻松地使用模块分区。

可以使用自己的约定来命名它们。 但是，需要为命令行编译器指定相应的 /reference 参数。

因素模块

使用模块实现文件和分区来分解模块，以便更轻松地维护代码，并可能更快地进行编译。

例如，将模块的实现从模块接口定义文件移出，并移入模块实现文件，这意味着对实现的更改不一定会导致导入模块的每个文件重新编译（除非有 `inline` 实现）。

通过模块分区，可以更轻松地在逻辑上对大型模块进行分解。 它们可用于缩短编译时间，以便对实现的一部分所做的更改不会导致重新编译所有模块的文件。

摘要

本教程介绍了 C++20 模块的基础知识。 你已经创建了主模块接口，定义了模块分区，并生成了模块实现文件。

另请参阅

C++ 中的模块概述
module、import、export 关键字
Visual Studio 中的 C++ 模块简介
实用的 C++20 模块和围绕 C++ 模块的工具的未来
将项目移动到 C++ 命名模块
演练：在 Microsoft Visual C++ 中生成和导入标头单元

模板 (C++)

模板是 C++ 中的泛型编程的基础。 作为强类型语言，C++ 要求所有变量都具有特定类型，由程序员显式声明或编译器推导。 但是，许多数据结构和算法无论在哪种类型上操作，看起来都是相同的。 使用模板可以定义类或函数的操作，并让用户指定这些操作应处理的具体类型。

定义和使用模板

模板是基于用户为模板参数提供的参数在编译时生成普通类型或函数的构造。 例如，可以定义如下所示的函数模板：

C++

```
template <typename T>
T minimum(const T& lhs, const T& rhs)
{
    return lhs < rhs ? lhs : rhs;
}
```

上面的代码描述了一个具有单个类型参数 T 的泛型函数的模板，其返回值和调用参数（lhs 和 rhs）都具有此类型。 可以随意命名类型参数，但按照约定，最常使用单个大写字母。 T 是模板参数；关键字 表示此参数是类型的占位符。 `typename` 调用函数时，编译器会将每个 `T` 实例替换为由用户指定或编译器推导的具体类型参数。 编译器从模板生成类或函数的过程称为“模板实例化”； 是模板 `minimum<int>` 的实例化。 `minimum<T>`

在其他地方，用户可以声明专用于 int 的模板实例。假设 get_a() 和 get_b() 是返回 int 的函数：

C++

```
int a = get_a();
int b = get_b();
int i = minimum<int>(a, b);
```

但是，由于这是一个函数模板，编译器可以从参数 a 和 b 中推导出类型，因此可以像普通函数一样调用它： `T`

C++

```
int i = minimum(a, b);
```

当编译器遇到最后一个语句时，它会生成一个新函数，在该函数中，T 在模板中的每个匹配项都替换为： `int`

C++

```
int minimum(const int& lhs, const int& rhs)
{
    return lhs < rhs ? lhs : rhs;
}
```

编译器如何在函数模板中执行类型推导的规则基于普通函数的规则。 有关详细信息，请参阅函数模板调用的重载解析。

类型参数

在上面的 `minimum` 模板中，请注意，在将类型参数 T 用于函数调用参数（在这些参数中会添加 const 和引用限定符）之前，不会以任何方式对其进行限定。

类型参数的数量没有实际限制。 以逗号分隔多个参数：

C++

```
template <typename T, typename U, typename V> class Foo{};
```

在此上下文中，关键字 **`class`** 等效于 **`typename`**。 可以将前面的示例表示为：

C++

```
template <class T, class U, class V> class Foo{};
```

可以使用省略号运算符 (...) 定义采用任意数量的零个或多个类型参数的模板：

C++

```
template<typename... Arguments> class vtclass;

vtclass< > vtinstance1;
vtclass<int> vtinstance2;
vtclass<float, bool> vtinstance3;
```

任何内置类型或用户定义的类型都可以用作类型参数。 例如，可以使用标准库中的 std::vector 来存储 **`int`**、**`double`**、std::string、`MyClass`、**`const`** `MyClass`*、`MyClass&` 等类型的变量。 使用模板时的主要限制是类型参数必须支持应用于类型参数的任何操作。 例如，如果我们使用 `minimum` 调用 `MyClass`，如以下示例所示：

C++

```
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
}
```

将生成编译器错误，因为 `MyClass` 不会为 < 运算符提供重载。

没有任何固有要求规定任何特定模板的类型参数都属于同一个对象层次结构，尽管可以定义强制实施此类限制的模板。 可以将面向对象的技巧与模板相结合；例如，可以将 Derived* 存储在向量 <Base*> 中。 请注意，自变量必须是指针

C++

```
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>());
```

`std::vector` 和其他标准库容器对 `T` 的元素施加的基本要求是 `T` 可复制且可复制构造。

非类型参数

与其他语言（如 C# 和 Java）中的泛型类型不同，C++ 模板支持非类型参数，也称为值参数。 例如，可以提供常量整型值来指定数组的长度，例如在以下示例中，它类似于标准库中的 std::array 类：

C++

```
template<typename T, size_t L>
class MyArray
{
    T arr[L];
public:
    MyArray() { ... }
};
```

记下模板声明中的语法。 `size_t` 值在编译时作为模板参数传入，必须是 `const` 或 `constexpr` 表达式。 可以如下所示使用它：

```C++
MyArray<MyClass*, 10> arr;
```

其他类型的值（包括指针和引用）可以作为非类型参数传入。 例如，可以传入指向函数或函数对象的指针，以自定义模板代码中的某些操作。

非类型模板参数的类型推导

在 Visual Studio 2017 及更高版本中，在 `/std:c++17` 模式或更高版本中，编译器会推导使用 `auto` 声明的非类型模板参数的类型：

```C++
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
```

模板作为模板参数

模板可以是模板参数。 在此示例中，MyClass2 有两个模板参数：类型名称参数 T 和模板参数 Arr：

```C++
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
};
```

由于 Arr 参数本身没有正文，因此不需要其参数名称。 事实上，从 的正文中引用 Arr 的类型名称或类参数名称是错误的。 `MyClass2` 因此，可以省略 Arr 的类型参数名称，如以下示例所示：

```C++
template<typename T, template<typename, int> class Arr>
class MyClass2
{
    T t; //OK
    Arr<T, 10> a;
};
```

默认模板自变量

类和函数模板可以具有默认自变量。 如果模板具有默认自变量，可以在使用时不指定该自变量。 例如，std::vector 模板有一个用于分配器的默认自变量：

```
C++
template <class T, class Allocator = allocator<T>> class vector;
```

在大多数情况下，默认的 std::allocator 类是可接受的，因此可以使用向量，如下所示：

```
C++
vector<int> myInts;
```

但如有必要，可以指定自定义分配器，如下所示：

```
C++
vector<int, MyAllocator> ints;
```

对于多个模板参数，第一个默认参数后的所有参数必须具有默认参数。

使用参数均为默认值的模板时，请使用空尖括号：

```
C++
template<typename A = int, typename B = double>
class Bar
{
    //...
};
...
int main()
{
    Bar<> bar; // use all default type arguments
}
```

模板特殊化

在某些情况下，模板不可能或不需要为任何类型都定义完全相同的代码。 例如，你可能希望定义在类型参数为指针、std::wstring 或派生自特定基类的类型时才执行的代码路径。 在这种情况下，可以为该特定类型定义模板的专用化。 当用户使用该类型对模板进行实例化时，编译器使用该专用化来生成类，而对于所有其他类型，编译器会选择更常规的模板。 如果专用化中的所有参数都是专用的，则称为“完整专用化”。 如果只有一些参数是专用的，则称为“部分专用化”。

C++

```
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
```

只要每个专用类型参数是唯一的，模板就可以具有任意数量的专用化。 只有类模板可能是部分专用。 模板的所有完整专用化和部分专用化都必须在与原始模板相同的命名空间中声明。

有关详细信息，请参阅模板规范。

Last updated on 2025/11/07

`typename`

项目 • 2023/10/17

在模板定义中，`typename` 向编译器提供未知标识符是类型的提示。 在模板参数列表中，它用于指定类型参数。

语法

> `typename` *`identifier`* `;`

备注

如果模板定义中的名称是依赖于模板自变量的限定名称，则必须使用 `typename` 关键字；如果限定名称是独立的，则此名称是可选的。 有关详细信息，请参阅模板和名称解析。

`typename` 可由任何类型在模板声明或定义中的任何位置使用。 不允许在基类列表中使用该关键字，除非将它用作模板基类的模板自变量。

C++

```
template <class T>
class C1 : typename T::InnerType // Error - typename not allowed.
{};
template <class T>
class C2 : A<typename T::InnerType>  // typename OK.
{};
```

`typename` 关键字也可以替代模板参数列表中的 `class` 使用。 例如，下面的语句在语义上是等效的：

C++

```
template<class T1, class T2>...
template<typename T1, typename T2>...
```

示例

C++

```
// typename.cpp
template<class T> class X
```

```
{
   typename T::Y m_y;   // treat Y as a type
};

int main()
{
}
```

另请参阅

模板
关键字

类模板

本文介绍特定于 C++ 类模板的规则。

类模板的成员函数

可以在类模板的内部或外部定义成员函数。 如果在类模板的外部定义成员函数，则会像定义函数模板一样定义它们。

C++

```
// 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()
{
}
```

就像任何模板类成员函数一样，类的构造函数成员函数的定义包含模板自变量列表两次。

成员函数可以是函数模板，并指定额外参数，如下面的示例所示。

C++

```
// 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()
{
}
```

嵌套类模板

模板可以在类或类模板中定义，在这种情况下，它们被称为成员模板。 作为类的成员模板称为嵌套类模板。 成员函数模板中讨论了作为函数的成员模板。

嵌套类模板被声明为外部类范围内的类模板。 可以在封闭类的内部或外部定义它们。

下面的代码演示普通类中的嵌套类模板。

C++

```
// 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();
}
```

以下代码使用嵌套模板类型参数来创建嵌套类模板：

C++

```
// 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()
*/
```

局部类不允许具有成员模板。

模板友元

类模板可以具有友元。 类或类模板、函数或函数模板可以是模板类的友元。 友元也可以是类模板或函数模板的专用化，但不是部分专用化。

在以下示例中，友元函数将定义为类模板中的函数模板。 此代码为模板的每个实例化生成一个友元函数版本。 如果您的友元函数与类依赖于相同的模板参数，则此构造很有用。

C++

```
// 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
*/
```

下一个示例涉及具有模板专用化的友元。 如果原始函数模板是友元，则函数模板专用化将自动为友元。

也可以只将模板的专用版本声明为友元，如以下代码中的友元声明前面的注释所示。 如果将专用化声明为友元，则必须将友元模板专用化的定义放在模板类之外。

C++

```
// 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
*/
```

下一个示例显示在类模板中声明的友元类模板。 该类模板随后用作友元类的模板参数。 友元类模板必须在声明它们的类模板外面定义。 友元模板的所有专用化或部分专用化也是原始类模板的友元。

C++

```
// 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
*/
```

重复使用模板参数

模板参数可以在模板参数列表中重复使用。 例如，以下代码是允许的：

```
C++
```

```
// 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()
{
}
```

另请参阅

模板

Last updated on 2025/11/07

函数模板

类模板可定义一系列相关类，这些类基于在实例化时传递到类的类型参数。 函数模板类似于类模板，但定义的是一系列函数。 利用函数模板，你可以指定基于相同代码但作用于不同类型或类的函数集。 以下函数模板交换两个项：

C++

```
// function_templates1.cpp
template< class T > void MySwap( T& a, T& b ) {
   T c(a);
   a = b;
   b = c;
}
int main() {
}
```

此代码定义交换自变量的值的一系列函数。 从此模板中，可以生成将交换 `int` 和 `long` 类型和用户定义类型的函数。 如果正确定义了类的复制构造函数和赋值运算符，`MySwap` 甚至会交换类。

此外，函数模板将阻止你交换不同类型的对象，因为在编译时编译器知道 a 和 b 参数的类型。

尽管非模板化函数可以使用 void 指针运行此函数，但模板版本是 typesafe。 请考虑以下调用：

C++

```
int j = 10;
int k = 18;
CString Hello = "Hello, Windows!";
MySwap( j, k );          //OK
MySwap( j, Hello );      //error
```

第二个 `MySwap` 调用触发了编译时错误，因为编译器无法生成具有不同类型的参数的 `MySwap` 函数。 如果使用了 void 指针，两个函数调用都将正确编译，但函数在运行时无法正常工作。

允许显式指定函数模板的模板参数。 例如：

C++

```
// 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.
}
```

当显式指定模板参数时，将对函数自变量执行常规隐式转换以将其转换为对应的函数模板自变量的类型。 在上面的示例中，编译器将 `j` 转换为类型 `char`。

另请参阅

模板
函数模板实例化
显式实例化
函数模板的显式专用化

Last updated on 2025/11/07

函数模板实例化

当首次为每个类型调用函数模板时，编译器会创建一个实例化。 每个实例化是专用于该类型的模板化函数版本。 每次将该函数用于该类型时，此实例化都将调用。 如果有几个相同的实例化，即使在不同的模块中，也只有该实例化的一个副本将在可执行文件中结束。

允许在函数模板中为其中的形参不依赖于模板实参的任何实参和形参对进行函数形参转换。

通过将特定类型作为自变量来声明模板，可以将函数模板显示实例化。 例如，以下代码是允许的：

C++

```
// 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()
{
}
```

另请参阅

函数模板

Last updated on 2025/11/07

显式实例化

您可以使用显式实例化来创建模板化类或函数的实例化，而不用将其实际用于您的代码。 由于这在创建使用模板进行分发的库（`.lib`）文件时非常有用，因此未实例化的模板定义不会放入对象（`.obj`）文件中。

示例

此代码针对 `MyStack` 变量和六个项显式实例化 `int`：

```
C++
template class MyStack<int, 6>;
```

该语句创建 `MyStack` 的实例化，而不保留对象的任何存储。 为所有成员生成代码。

下一行仅显式实例化构造函数成员函数：

```
C++
template MyStack<int, 6>::MyStack( void );
```

你可以使用特定类型自变量重新声明函数模板来显式实例化这些模板，如函数模板实例化中的示例所示。

可以使用 `extern` 关键字来防止自动实例化成员。 例如：

```
C++
extern template class MyStack<int, 6>;
```

同样，您可以将特定成员标记为外部且未实例化：

```
C++
extern template MyStack<int, 6>::MyStack( void );
```

你可以使用 `extern` 关键字阻止编译器在多个对象模块中生成相同的实例化代码。 如果调用该函数，则必须在至少一个链接模块中使用指定的显式模板参数来实例化该函数模板。 否则，生成程序时会出现链接器错误。

注意

专用化中的 `extern` 关键字仅适用于在类主体外定义的成员函数。 类声明中定义的函数被视为内联函数，并且始终实例化。

另请参阅

函数模板

Last updated on 2025/11/07

函数模板的显式专用化

利用函数模板，您可以通过为特定类型提供函数模板的显式专用化（重写）来定义该类型的特殊行为。 例如：

C++

```
template<> void MySwap(double a, double b);
```

此声明使您能够为 `double` 变量定义不同函数。 与非模板函数相似，也会应用标准类型转换（如将 `float` 类型的变量提升到 `double`）。

示例

C++

```
// 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()
{
}
```

另请参阅

函数模板

Last updated on 2025/11/07

函数模板的部分排序 (C++)

有多个与函数调用的自变量列表匹配的函数模板可用。 C++ 定义了函数模板的部分排序以指定应调用的函数。 由于有一些模板可能会视为专用化程度相同，因此排序只是部分的。

编译器从可能的匹配项中选择可用的专用化程度最高的函数模板。 例如，如果一个函数模板采用 `T` 类型，而另一个采用 `T*` 的函数模板可用，则称 `T*` 版本的专用化程度更高。 只要参数是指针类型，它就优于泛型 `T` 版本，即使两者都是允许的匹配项。

通过以下过程可确定一个函数模板候选项是否更加专用化：

1. 考虑两个函数模板： `T1` 和 `T2`。
2. 将 `T1` 中的参数替换为假想的唯一类型 `X`。
3. 当 `T1` 中存在参数列表时，查看 `T2` 是否为该参数列表的有效模板。 忽略任何隐式转换。
4. 对 `T1` 和 `T2` 执行相反的过程。
5. 如果一个模板是另一个模板的有效模板参数列表，但反之不成立，则认为该模板的专用化程度不如另一个模板。 如果通过使用上一步，两个模板为彼此形成有效的参数，那么它们被视为同等专用化，并且尝试使用它们时会导致调用不明确。
6. 使用以下规则：
   a. 针对特定类型的模板的专用化程度高于采用泛型类型参数的模板。
   b. 仅采用 `T*` 的模板的专用化程度比仅采用 `T` 的模板更高，因为假设类型 `X*` 是 `T` 模板参数的有效参数，但 `X` 不是 `T*` 模板参数的有效参数。
   c. `const T` 比 `T` 专用化程度更高，因为 `const X` 是 `T` 模板参数的有效参数，但 `X` 不是 `const T` 模板参数的有效参数。
   d. `const T*` 比 `T*` 专用化程度更高，因为 `const X*` 是 `T*` 模板参数的有效参数，但 `X*` 不是 `const T*` 模板参数的有效参数。

示例

以下示例按照标准中的规定运行：

C++

```
// 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.
}
```

输出

Output

```
Less specialized function called
More specialized function called
Even more specialized function for const T*
```

另请参阅

函数模板

Last updated on 2025/11/07

成员函数模板

术语成员模板引用了成员函数模板和嵌套类模板。 成员函数模板是类或类模板的成员的函数模板。

成员函数可以是多个环境中的函数模板。 类模板的所有函数都是泛型的，但却不称为成员模板或成员函数模板。 如果这些成员函数采用其自己的模板自变量，则将它们视为成员函数模板。

示例：声明成员函数模板

非模板类或类模板的成员函数模板将声明为带有其模板参数的函数模板。

C++

```
// member_function_templates.cpp
struct X
{
   template <class T> void mf(T* t) {}
};

int main()
{
   int i;
   X* x = new X();
   x->mf(&i);
}
```

示例：类模板的成员函数模板

以下示例显示类模板的成员函数模板。

C++

```
// member_function_templates2.cpp
template<typename T>
class X
{
public:
   template<typename U>
   void mf(const U &u)
   {
   }
};

int main()
{
}
```

示例：定义类外部的成员模板

C++

```
// 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()
{
}
```

示例：模板化用户定义的转换

局部类不允许具有成员模板。

成员函数模板不能是虚函数。 当使用与基类虚函数相同的名称进行声明时，成员模板函数不能从基类重写虚函数。

以下示例展示了模板化用户定义的转换：

C++

```
// 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>.
}
```

另请参阅

函数模板

Last updated on 2025/11/07

模板特殊化 (C++)

类模板可以部分专用化，生成的类仍是模板。 在类似于下面的情况下，部分专用化允许为特定类型部分自定义模板代码：

- 模板有多个类型，且只有一部分需要专用化。 结果是基于其余类型参数化的模板。
- 模板只有一个类型，但指针、引用、指向成员的指针或函数指针类型需要专用化。 专用化本身仍是指向或引用的类型上的模板。

示例：类模板的部分专用化

C++

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

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{};

int main() {
   printf_s("PTS<S>::IsPointer == %d \nPTS<S>::IsPointerToDataMember == %d\n",
           PTS<S>::IsPointer, PTS<S>:: IsPointerToDataMember);
   printf_s("PTS<S*>::IsPointer == %d \nPTS<S*>::IsPointerToDataMember == %d\n"
           , PTS<S*>::IsPointer, PTS<S*>:: IsPointerToDataMember);
   printf_s("PTS<int S::*>::IsPointer == %d \nPTS"
           "<int S::*>::IsPointerToDataMember == %d\n",
           PTS<int S::*>::IsPointer, PTS<int S::*>::
           IsPointerToDataMember);
}
```

Output

```
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
```

示例：指针类型的部分专用化

如果有一个采用任何 `T` 类型的模板集合类，则可以创建采用任何指针类型 `T*` 的部分专用化。 以下代码演示了一个集合类模板 `Bag` 以及指针类型的部分专用化，在此专用化中，该集合在将指针类型复制到数组前取消引用它们。 该集合随后存储指向的值。 对于原始模板，只有指针本身将存储在集合中，从而使数据易受删除或修改。 在此特殊指针版本的集合中，添加了在 `add` 方法中检查 null 指针的代码。

C++

```cpp
// 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);
   delete[] p;
   p = NULL;
   xp.add(p);
   xp.print();
}
```

Output

```
10 9 8
a b c
Null pointer!
3 87 8 100
```

示例：定义部分专用化，使一种类型为 `int`

以下示例定义了一个采用由任意两种类型构成的对的模板类，然后定义已专用化的模板类的部分专用化，使其中一个类型为 `int`。 该专用化定义了基于整数实现简单气泡排序的另一种排序方法。

C++

```
// 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<const char*, const char*> dict(10);
    dict.print();
    dict.add("apple", "fruit");
    dict.add("banana", "fruit");
    dict.add("dog", "animal");
    dict.print();

    Dictionary<int, const char*> dict_specialized(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();
}
```

Output

```
{apple, fruit}
{banana, fruit}
{dog, animal}
{100, apple}
{101, banana}
{103, dog}
{89, cat}

Sorted list:
{89, cat}
{100, apple}
{101, banana}
{103, dog}
```

Last updated on 2025/11/07

模板和名称解析

在模板定义中，有三种类型的名称。

- 本部声明的名称，包括模板本身的名称以及在模板定义中声明的任何名称。
- 模板定义之外的封闭范围中的名称。
- 在某种程度上依赖于模板自变量的名称（称为依赖名称）。

尽管前两个名称也属于类和函数范围，但模板定义中需要针对名称解析的特殊规则来处理依赖名称的提高的复杂性。 原因在于，在对模板进行实例化前，编译器几乎不知道这些名称，因为它们可能是依赖于所使用的模板自变量的完全不同的类型。 将根据常用规则，在模板的定义点上查找非依赖名称。 将为所有模板专用化查找这些名称（独立于模板参数）一次。 在将模板实例化并为每个专用化单独查找模板之前，将不会查找依赖名称。

如果某个类型依赖于模板参数，则该类型属于依赖类型。 具体而言，如果类型是以下项，则属于依赖类型：

- 模板自变量本身：

  ```
  C++
  T
  ```

- 具有包括依赖类型的限定的限定名：

  ```
  C++
  T::myType
  ```

- 当非限定部分标识依赖类型时的限定名：

  ```
  C++
  N::T
  ```

- 其基类型属于依赖类型的不变或可变类型：

  ```
  C++
  const T
  ```

- 基于依赖类型的指针、引用、数组或函数指针类型：

```C++
T *, T &, T [10], T (*)()
```

- 大小基于模板参数的数组：

```C++
template <int arg> class X {
int x[arg] ; // dependent type
}
```

- 从模板参数构造的模板类型：

```C++
T<int>, MyTemplate<T>
```

类型依赖和值依赖

模板参数上的名称和表达式依赖项分为类型依赖项或值依赖项，具体取决于模板参数是类型参数还是值参数。 此外，在模板自变量上有类型依赖项的模板中声明的任何标识符都被视为依赖值，使用值依赖表达式初始化的整数类型或枚举类型也是如此。

类型依赖和值依赖表达式是涉及类型依赖或值依赖的变量的表达式。 这些表达式可能有不同的语义，具体取决于用于模板的参数。

另请参阅

模板

Last updated on 2025/11/07

依赖类型的名称解析

使用 `typename` 作为模板定义中的限定名称，以告知编译器给定的限定名称标识一个类型。 有关详细信息，请参阅 typename。

C++

```
// 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.");
}
```

Output

```
Name resolved by using typename keyword.
```

针对依赖名称的名称查找将检查模板定义上下文（在下面的示例中，此上下文将查找 `myFunction(char)`）和模板实例化上下文中的名称。在下面的示例中，将在 main 中实例化模板；因此，`MyNamespace::myFunction` 从实例化的角度来看是可见的，因而将选取它作为更好的匹配。 如果重命名 `MyNamespace::myFunction`，则将调用 `myFunction(char)`。

所有名称都会得到解析，就如同它们是依赖名称一样。 尽管如此，如果存在任何可能的冲突，建议您使用完全限定名。

C++

```
// 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);
}
```

输出

Output

```
Int MyNamespace::myFunction
```

模板消除歧义

Visual Studio 2012 强制实施 C++98/03/11 标准规则以使用“template”关键字消除歧义。 在下面的示例中，Visual Studio 2010 将接受不一致性行和一致性行。 Visual Studio 2012 仅接受一致性行。

C++

```
#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;
}
```

需要符合消除歧义规则，因为默认情况下，C++ 假定 `AY::Rebind` 不是模板，因此编译器会将后面的“`<`”解释为小于。 它必须知道 `Rebind` 是模板，这样才能正确地将“`<`”分析为尖括号。

另请参阅

名称解析

Last updated on 2025/11/07

本地声明名称的名称解析

通过和不通过模板自变量都可引用模板名称本身。 在类模板的范围内，名称本身引用模板。 在模板专用化或部分专用化的范围中，名称单独引用专用化或部分专用化。 也可以通过适当的模板参数引用模板的其他专用化或部分专用化。

示例：专用化与部分专用化

以下代码演示在专用化或部分专用化的范围内以不同的方式解释类模板的名称 `A`。

C++

```
// 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>.
};
```

示例：模板参数和对象之间的名称冲突

在模板参数与另一个对象之间出现名称冲突的情况下，模板参数可隐藏，也可不隐藏。 下列规则将帮助确定优先顺序。

模板参数位于从其首先出现的位置开始一直到类或函数末尾的范围中。 如果名称再次出现在模板参数列表或基类列表中，则它将引用同一类型。 在标准 C++ 中，无法在同一范围中声明与模板参数相同的其他名称。 Microsoft 扩展允许在模板范围内重新定义模板参数。 以下示例演示在类模板的基规范中使用模板参数。

C++

```
// template_name_resolution4.cpp
// compile with: /EHsc
template <class T>
class Base1 {};

template <class T>
```

```
class Derived1 : Base1<T> {};

int main() {
   // Derived1<int> d;
}
```

示例：在类模板外定义成员函数

当在类模板外定义成员函数时，可以使用不同的模板参数名称。 如果类模板成员函数定义与声明对模板参数使用了不同的名称，并且定义中使用的名称与声明的其他成员冲突，则模板声明中的成员优先。

C++

```
// 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();
}
```

Output

```
Z::Z()
```

示例：在命名空间外定义模板或成员函数

在声明模板的命名空间外定义函数模板或成员函数时，模板自变量将优先于命名空间中其他成员的名称。

C++

```cpp
// 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();
}
```

Output

```
C<T>::g
```

示例：基类或成员名称隐藏模板参数

在位于模板类声明之外的定义中，如果模板类具有不依赖于模板自变量的基类，并且该基类或其成员之一与模板自变量的名称相同，则该基类或成员名称将隐藏模板自变量。

C++

```cpp
// 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;
}
```

Output

```
Base
1
```

另请参阅

名称解析

Last updated on 2025/11/07

函数模板调用的重载解析

函数模板可以重载具有相同名称的非模板函数。 在此方案中，编译器首先尝试使用模板自变量推导来解析函数调用，以实例化具有唯一特化的函数模板。 如果模板自变量推导失败，则编译器会考虑实例化函数模板重载和非模板函数重载来解析调用。 这些其他重载称为*候选集*。 如果模板自变量推导成功，则按照重载决策的规则，将生成的函数与候选集中的其他函数进行比较，以确定最佳匹配。 有关详细信息，请参阅函数重载。

示例：选择非模板函数

如果非模板函数是函数模板的很好的匹配，则选择非模板函数（除非已显式指定模板自变量），如以下示例中的 `f(1, 1)` 调用。

C++

```
// 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 non-template function.
   f('a', 1); // Chooses the function template.
   f<int, int>(2, 2);  // Template arguments explicitly specified.
}
```

Output

```
f(int, int)
void f(T1, T2)
void f(T1, T2)
```

示例：首选精确匹配函数模板

下一个示例阐释了一点，即如果非模板函数需要转换，则完全匹配的函数模板是首选的。

C++

```cpp
// 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 function template f(long, int) because f(int, int)
   // would require a conversion from long to int.
   f(l, i);
}
```

Output

```
void f(T1, T2)
```

另请参阅

名称解析

typename

Last updated on 2025/11/07

源代码组织（C++ 模板）

在定义类模板时，必须以这样一种方式组织源代码：在编译器需要成员定义时，可以看到成员定义。 可以选择使用包含模型或显式实例化模型。 在包含模型中，在使用模板的每个文件中都加入成员定义。 这是最简单的方法，它在可与模板一起使用的具体类型方面提供了最大的灵活性。 缺点是会增加编译时间。 如果项目或包含的文件本身很大，那么时间可能很长。 使用显式实例化方法，模板本身会实例化特定类型的具体类或类成员。 此方法可以加快编译时间，但它仅限于模板实现者提前启用的那些类的使用情况。 通常，我们建议使用包含模型，除非由于编译时间问题而存在局限性。

背景

模板与普通类不同，因为编译器不会为模板或其任何成员生成对象代码。 在使用具体类型实例化模板之前，不会生成任何内容。 当编译器遇到模板实例化（例如 `MyClass<int> mc;`）并且尚未存在具有该签名的类时，它将生成一个新类。 它还会尝试为使用的任何成员函数生成代码。 如果这些定义位于一个并未直接或间接 #included 在正在编译的 .cpp 文件内的文件中，编译器就无法看到它们。 从编译器的角度来看，这不一定是错误。 这些函数可以在另一个翻译单元中定义，链接器会在那里找到它们。 如果链接器找不到该代码，则会引发无法解析的外部错误。

包含模型

使模板定义在整个翻译单元中可见的最简单、最常见的方法是将定义放入头文件本身。 使用模板的任何 `.cpp` 文件只需 `#include` 标头。 这是标准库中使用的方法。

C++

```
#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
```

使用此方法，编译器有权访问完整的模板定义，并可以按需为任何类型实例化模板。 这很简单，而且相对容易维护。 但是，包含模型确实存在编译时间成本。 在大型程序中，此类成本可能很高，尤其是在模板标头本身 #include 其他标头的情况下。 每个 #include 标头的 `.cpp` 文件都将获得自己的函数模板和所有定义的副本。 链接器通常能够解决问题，这样最终便不会得到函数的多个定义，但是完成这项工作需要时间。 在较小的程序中，可能不需要太长的额外编译时间。

显式实例化模型

如果包含模型无法用于你的项目，并且你明确知道将用于实例化模板的类型集，则可以将模板代码分离到 `.h` 和 `.cpp` 文件中，并在 `.cpp` 文件中显式实例化模板。 此方法将生成编译器在遇到用户实例化时将看到的对象代码。

通过使用关键字 **`template`**，然后使用要实例化的实体的签名，创建显式实例化。 此实体可以是类型或成员。 如果显式实例化类型，则会实例化所有成员。

头文件 `MyArray.h` 声明模板类 `MyArray`：

C++

```
//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
```

源文件 `MyArray.cpp` 显式实例化 `template MyArray<double, 5>` 和 `template MyArray<string, 5>`：

C++

```cpp
//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>;
```

在前面的示例中，显式实例化位于 `.cpp` 文件的底部。 `MyArray` 只能用于 **`double`** 或 `String` 类型。

> **注意**
>
> 在 C++11 中，模板定义的上下文中已弃用 **`export`** 关键字。 实际上，这几乎没有影响，因为大多数编译器从不支持它。

Last updated on 2025/11/07

事件处理

COM 类支持事件处理（实现 COM 对象的 C++ 类，通常使用 ATL 类或 coclass 属性）。 有关详细信息，请参阅 COM 中的事件处理。

本机 C++ 类（不实现 COM 对象的 C++ 类）也支持事件处理。 本机 C++ 事件处理支持已弃用，将在将来的版本中删除。 有关详细信息，请参阅本机 C++ 中的事件处理。

> **注意**
>
> 本机 C++ 中的事件特性与标准 C++ 不兼容。 指定 /permissive- 一致性模式时，它们不会编译。

事件处理支持单线程和多线程使用。 它防止数据同时进行多线程访问。 可以从事件源或接收方类派生子类。 这些子类支持扩展事件溯源和接收。

Microsoft C++ 编译器包含用于声明事件和事件处理程序的特性和关键字。 事件特性和关键字可用于 CLR 程序和本机 C++ 程序中。

展开表

项目	说明
event_source	创建事件源。
event_receiver	创建事件接收器（接收器）。
__event	声明事件。
__raise	强调一个事件的调用站点。
__hook	将处理程序方法与事件关联。
__unhook	取消处理程序方法与事件的关联。

另请参阅

C++ 语言参考
关键字

Last updated on 2025/11/07

`__event` 关键词

声明事件。

> ⓘ **注意**
>
> 本机C++中的事件属性与标准C++不兼容。 指定 /permissive- 一致性模式时，它们不会编译。

语法

```
__event member-function-declarator ;
__event __interface interface-specifier ;
__event data-member-declarator ;
```

注解

特定于Microsoft关键字 `__event` 可以应用于成员函数声明、接口声明或数据成员声明。 但是，不能使用 `__event` 关键字限定嵌套类的成员。

根据事件源和接收器是本机C++、COM 还是托管 （.NET Framework），可以使用以下构造作为事件：

展开表

原生C++	COM	托管 （.NET Framework）
member 函数	-	方法
-	接口	-
-	-	数据成员

在 __hook 事件接收器中使用，将处理程序成员函数与事件成员函数相关联。 使用 `__event` 关键字创建事件后，调用该事件后，将挂钩到该事件的所有事件处理程序。

`__event` 成员函数声明不能有定义;定义是隐式生成的，因此可以像调用任何普通成员函数一样调用事件成员函数。

> ⓘ **注意**

模板化类或结构不能包含事件。

本机事件

本机事件是成员函数。 返回类型通常是 `HRESULT` 或 `void`，但可以是任何整型类型，包括一个 `enum`。 当事件使用整型返回类型时，当事件处理程序返回非零值时定义错误条件。 在这种情况下，引发的事件将调用其他委托。

```
C++
// Examples of native C++ events:
__event void OnDblClick();
__event HRESULT OnClick(int* b, char* s);
```

有关示例代码，请参阅本机C++中的事件处理。

COM 事件

COM 事件是接口。 事件源接口中成员函数的参数应 *位于参数中*，但不严格执行。 这是因为当多播时，*out* 参数无效。 如果使用 *out* 参数，则会发出级别 1 警告。

返回类型通常是 `HRESULT` 或 `void`，但可以是任何整型类型，包括 `enum`。 当事件使用整型返回类型并且事件处理程序返回非零值时，它是错误条件。 引发的事件中止对其他委托的调用。 编译器自动将事件源接口标记为生成的 IDL 中的接口 source。

COM __interface 事件源始终 `__event` 需要关键字。

```
C++
// Example of a COM event:
__event __interface IEvent1;
```

有关示例代码，请参阅 COM 中的事件处理。

托管事件

有关在新语法中编码事件的信息，请参阅 事件。

托管事件是数据成员或成员函数。 与事件一起使用时，委托的返回类型必须符合 公共语言规范。 事件处理程序的返回类型必须与委托的返回类型匹配。 有关委托的详细信息，请参阅 委托和事件。 如果托管事件是数据成员，则其类型必须是指向委托的指针。

在 .NET Framework 中，可以将数据成员视为方法本身（即 `Invoke` 其相应委托的方法）。 为此，请预定义用于声明托管事件数据成员的委托类型。 相比之下，如果尚未定义托管委托，则托管事件方法会隐式定义相应的托管委托。 例如，可以声明事件值，例如 `OnClick` 事件，如下所示：

C++

```
// Examples of managed events:
__event ClickEventHandler* OnClick;  // data member as event
__event void OnClick(String* s);  // method as event
```

隐式声明托管事件时，可以指定 `add` 和 `remove` 访问器，这些访问器会在添加或删除事件处理程序时调用。 还可以定义从类外部调用事件（引发）事件的成员函数。

示例：本机事件

C++

```
// EventHandling_Native_Event.cpp
// compile with: /c
[event_source(native)]
class CSource {
public:
   __event void MyEvent(int nValue);
};
```

示例：COM 事件

C++

```
// EventHandling_COM_Event.cpp
// compile with: /c
#define _ATL_ATTRIBUTES 1
#include <atlbase.h>
#include <atlcom.h>

[ module(dll, name="EventSource", uuid="6E46B59E-89C3-4c15-A6D8-B8A1CEC98830") ];

[ dual, uuid("00000000-0000-0000-0000-000000000002") ]
__interface IEventSource {
   [id(1)] HRESULT MyEvent();
};
[ coclass, uuid("00000000-0000-0000-0000-000000000003"),  event_source(com) ]
class CSource : public IEventSource {
public:
   __event __interface IEventSource;
   HRESULT FireEvent() {
```

```
        __raise MyEvent();
        return S_OK;
    }
};
```

另请参阅

关键字
事件处理
event_source
event_receiver
__hook
__unhook
__raise

Last updated on 2025/11/07

`__hook` 关键字

将处理程序方法与事件关联。

> ⓘ **注意**
>
> 本机 C++ 中的事件特性与标准 C++ 不兼容。 指定 /permissive- 一致性模式时，它们不会编译。

语法

C++

```
long __hook(
    &SourceClass::EventMethod,
    source,
    &ReceiverClass::HandlerMethod
    [, receiver = this]
);

long __hook(
    interface,
    source
);
```

参数

`&SourceClass::EventMethod`

指向要将事件处理程序方法挂钩到的事件方法的指针：

- 本机 C++ 事件：*`SourceClass`* 是事件源类，*`EventMethod`* 是事件。
- COM 事件：*`SourceClass`* 是事件源接口，*`EventMethod`* 是其方法之一。
- 托管事件：*`SourceClass`* 是事件源类，*`EventMethod`* 是事件。

`interface`

与 *`receiver`* 挂钩的接口名称，仅适用于 *`layout_dependent`* 属性的 event_receiver 参数为 **`true`** 的 COM 事件接收器。

`source`

指向事件源的实例的指针。 根据 `type` 中指定的代码 `event_receiver`，*`source`* 可以是以下类型之一：

- 本机事件源对象指针。
- 基于 `IUnknown` 的指针（COM 源）。
- 托管对象指针（针对托管事件）。

`&ReceiverClass::HandlerMethod`
指向要挂钩到事件的事件处理程序方法的指针。 处理程序被指定为类的方法或对其的引用。 如果未指定类名，`__hook` 会假定该类是它从中调用的类。

- 本机 C++ 事件：`ReceiverClass` 是事件接收器类，`HandlerMethod` 是处理程序。
- COM 事件：`ReceiverClass` 是事件接收器接口，`HandlerMethod` 是其处理程序之一。
- 托管事件：`ReceiverClass` 是事件接收器类，`HandlerMethod` 是处理程序。

`receiver`
（可选）指向事件接收器类的实例的指针。 如果不指定接收器，则默认为在其中调用 `__hook` 的接收器类或结构。

使用情况

可以在事件接收器类的外部的任何函数范围（包括 main）中使用。

备注

使用事件接收器中的内部函数 `__hook` 将处理程序方法与事件方法关联或挂钩。 随后会在源引发指定事件时调用指定的事件处理程序。 可以将多个处理程序挂钩到单个事件，或将多个事件挂钩到单个处理程序。

`__hook` 有两种形式。 在大多数情况下，可以使用第一个（四参数）形式，特别是对于 属性的 event_receiver 参数为 `false` 的 COM 事件接收器。

在这些情况下，当在其中一个方法上激发事件前，不需要在接口中挂钩所有方法。 只需挂钩处理事件的方法。 只能将 `__hook` 的第二种（双参数）形式用于其中 `layout_dependent = true` 的 COM 事件接收器。

`__hook` 将返回一个长值。 非零返回值表示发生了错误（托管事件引发了异常）。

编译器将检查是否存在事件以及事件签名是否与委托签名一致。

可以在事件接收器外部调用 `__hook` 和 `__unhook`，COM 事件除外。

使用 `__hook` 的替代方法是使用 += 运算符。

有关采用新语法对托管事件进行编码的信息，请参阅 event。

> **注意**
>
> 模板化的类或结构不能包含事件。

示例

有关示例，请参阅本机 C++ 中的事件处理和 COM 中的事件处理。

另请参阅

关键字
事件处理
event_source
event_receiver
__event
__unhook
__raise

Last updated on 2025/11/07

`__raise` 关键词

强调事件的调用站点。

> ⓘ **注意**
>
> 本机C++中的事件属性与标准C++不兼容。 指定 /permissive- 一致性模式时，它们不会编译。

语法

> `__raise` *`method-declarator`* `;`

注解

从托管代码中，只能从定义事件的类中引发事件。 有关详细信息，请参阅 event。

`__raise`如果调用非事件，关键字将导致发出错误。

> ⓘ **注意**
>
> 模板化类或结构不能包含事件。

示例：

C++

```
// EventHandlingRef_raise.cpp
struct E {
   __event void func1();
   void func1(int) {}

   void func2() {}

   void b() {
      __raise func1();
      __raise func1(1);  // C3745: 'int Event::bar(int)':
                         // only an event can be 'raised'
      __raise func2();   // C3745
   }
};

int main() {
```

```
    E e;
    __raise e.func1();
    __raise e.func1(1);  // C3745
    __raise e.func2();   // C3745
}
```

另请参阅

关键字
事件处理
__event
__hook
__unhook
.NET 和 UWP 的组件扩展

Last updated on 2025/11/07

`__unhook` 关键字

取消处理程序方法与事件的关联。

> ⓘ **注意**
>
> 本机 C++ 中的事件特性与标准 C++ 不兼容。 指定 /permissive- 一致性模式时，它们不会编译。

语法

C++

```
long __unhook(
   &SourceClass::EventMethod,
   source,
   &ReceiverClass::HandlerMethod
   [, receiver = this]
);

long __unhook(
   interface,
   source
);

long __unhook(
   source
);
```

参数

`&SourceClass::EventMethod`

指向从中解除挂钩事件处理程序方法的事件方法的指针：

- 本机 C++ 事件：`SourceClass` 是事件源类，`EventMethod` 是事件。
- COM 事件：`SourceClass` 是事件源接口，`EventMethod` 是其方法之一。
- 托管事件：`SourceClass` 是事件源类，`EventMethod` 是事件。

`interface`

从 receiver 中解除挂钩的接口名称，仅适用于 特性的 *layout_dependent* 参数为 event_receiver 的 COM 事件接收器 `true`。

`source`
指向事件源的实例的指针。 根据 `type` 中指定的代码 `event_receiver`，source 可以是以下类型之一：

- 本机事件源对象指针。
- 基于 `IUnknown` 的指针（COM 源）。
- 托管对象指针（针对托管事件）。

`&ReceiverClass::HandlerMethod` 指向要从事件中解除挂钩的事件处理程序方法的指针。 处理程序被指定为类的方法或对同一方法的引用；如果不指定类名，`__unhook` 会假定该类是在其中调用它的类。

- 本机 C++ 事件：`ReceiverClass` 是事件接收器类，`HandlerMethod` 是处理程序。
- COM 事件：`ReceiverClass` 是事件接收器接口，`HandlerMethod` 是其处理程序之一。
- 托管事件：`ReceiverClass` 是事件接收器类，`HandlerMethod` 是处理程序。

`receiver` （可选）指向事件接收器类实例的指针。 如果不指定接收器，则默认为在其中调用 `__unhook` 的接收器类或结构。

使用情况

可以在事件接收器类之外的任何函数范围内使用，包括 `main`。

备注

使用事件接收器中的内部函数 `__unhook` 取消处理程序方法与事件方法的关联或从事件方法“解除挂钩”处理程序方法。

`__unhook` 有三种形式。 在大多数情况下，可以使用第一种 (four-argument) 形式。 只能将 `__unhook` 的第二种 (two-argument) 形式用于 COM 事件接收器；它将解除挂钩整个事件接口。 可以使用第三种 (one-argument) 形式从指定的源中解除挂钩所有委托。

非零返回值指示已发生错误（托管事件将引发异常）。

如果对尚未挂钩的事件和事件处理程序调用 `__unhook`，它将不起作用。

在编译时，编译器将验证事件是否存在，并利用指定的处理程序执行参数类型检查。

可以在事件接收器外部调用 `__hook` 和 `__unhook`，COM 事件除外。

使用 `__unhook` 的替代方法是使用 -= 运算符。

有关采用新语法对托管事件进行编码的信息，请参阅事件。

> **注意**
>
> 模板化的类或结构不能包含事件。

示例

有关示例，请参阅本机 C++ 中的事件处理和 COM 中的事件处理。

另请参阅

关键字
event_source
event_receiver
__event
__hook
__raise

Last updated on 2025/11/07

本机 C++ 中的事件处理

在处理本机 C ++ 事件时，分别使用 event_source 和 event_receiver 特性设置事件源和事件接收器，并指定 `type` = `native`。 这些特性允许应用它们的类在本机的非 COM 上下文中激发和处理事件。

> **注意**
>
> 本机 C++ 中的事件特性与标准 C++ 不兼容。 指定 /permissive- 一致性模式时，它们不会编译。

声明事件

在事件源类中，使用方法声明上的 __event 关键字将方法声明为事件。 请确保声明方法，但不定义该方法。 如果这样做，它将生成编译器错误，因为编译器在事件中生成时隐式定义方法。 本机事件可以是带有零个或多个参数的方法。 返回类型可以是 `void` 或任何整型。

定义事件处理程序

在事件接收器类中，定义事件处理程序。 事件处理程序是具有与它们将处理的事件匹配的签名（返回类型、调用约定和自变量）的方法。

将事件处理程序挂钩到事件

同样在事件接收器类中，可使用内部函数 __hook 将事件与事件处理程序关联，并可使用 __unhook 取消事件与事件处理程序的关联。 您可将多个事件挂钩到一个事件处理程序，或将多个事件处理程序挂钩到一个事件。

触发事件

若要触发事件，调用再事件源类中声明为事件的方法。 如果处理程序已挂钩到事件，则将调用处理程序。

本机 C++ 事件代码

以下示例演示如何在本机 C++ 中激发事件。 若要编译并运行此示例，请参考代码中的注释。 若要在 Visual Studio IDE 中生成代码，请验证 `/permissive-` 选项是否已关闭。

示例

Code

C++

```
// evh_native.cpp
// compile by using: cl /EHsc /W3 evh_native.cpp
#include <stdio.h>

[event_source(native)]
class CSource {
public:
   __event void MyEvent(int nValue);
};

[event_receiver(native)]
class CReceiver {
public:
   void MyHandler1(int nValue) {
      printf_s("MyHandler1 was called with value %d.\n", nValue);
   }

   void MyHandler2(int nValue) {
      printf_s("MyHandler2 was called with value %d.\n", nValue);
   }

   void hookEvent(CSource* pSource) {
      __hook(&CSource::MyEvent, pSource, &CReceiver::MyHandler1);
      __hook(&CSource::MyEvent, pSource, &CReceiver::MyHandler2);
   }

   void unhookEvent(CSource* pSource) {
      __unhook(&CSource::MyEvent, pSource, &CReceiver::MyHandler1);
      __unhook(&CSource::MyEvent, pSource, &CReceiver::MyHandler2);
   }
};

int main() {
   CSource source;
   CReceiver receiver;

   receiver.hookEvent(&source);
   __raise source.MyEvent(123);
   receiver.unhookEvent(&source);
}
```

输出

Output

```
MyHandler2 was called with value 123.
MyHandler1 was called with value 123.
```

另请参阅

事件处理

Last updated on 2025/11/07

COM 中的事件处理

在 COM 事件处理中，使用 event_source 和 event_receiver 特性分别设置事件源和事件接收器，同时指定 `type` = `com`。 这些特性为自定义接口、调度接口和双重接口注入适当的代码。 借助注入的代码，特性化类可触发事件并通过 COM 连接点处理事件。

> ⓘ **注意**
>
> 本机 C++ 中的事件特性与标准 C++ 不兼容。 指定 /permissive- 一致性模式时，它们不会编译。

声明事件

在事件源类中，在接口声明上使用 __event 关键字以将该接口的方法声明为事件。 当您将该接口的事件作为接口方法调用时，将激发这些事件。 事件接口上的方法可以有零个或多个参数（应全部是 in 参数）。 返回类型可以是 void 或任何整型。

定义事件处理程序

在事件接收器类中，定义事件处理程序。 事件处理程序是具有与它们将处理的事件匹配的签名（返回类型、调用约定和自变量）的方法。 对于 COM 事件，调用约定不必匹配。 有关详细信息，请参阅下面的依赖于布局的 COM 事件。

将事件处理程序挂钩到事件

同样在事件接收器类中，可使用内部函数 __hook 将事件与事件处理程序关联，并可使用 __unhook 取消事件与事件处理程序的关联。 您可将多个事件挂钩到一个事件处理程序，或将多个事件处理程序挂钩到一个事件。

> ⓘ **注意**
>
> 通常，有两种方法使 COM 事件接收器能够访问事件源接口定义。 第一种是共享公共头文件，如下所示。 第二种是将 #import 与 `embedded_idl` 导入限定符结合使用，以便让事件源类型库写入到保留了特性生成的代码的 .tlh 文件。

触发事件

若要激发事件，只需调用在事件源类中使用 `__event` 关键字声明的接口中的方法。 如果处理程序已挂钩到事件，则将调用处理程序。

COM 事件代码

下面的示例演示如何在 COM 类中激发事件。 若要编译并运行此示例，请参考代码中的注释。

C++

```
// evh_server.h
#pragma once

[ dual, uuid("00000000-0000-0000-0000-000000000001") ]
__interface IEvents {
   [id(1)] HRESULT MyEvent([in] int value);
};

[ dual, uuid("00000000-0000-0000-0000-000000000002") ]
__interface IEventSource {
   [id(1)] HRESULT FireEvent();
};

class DECLSPEC_UUID("530DF3AD-6936-3214-A83B-27B63C7997C4") CSource;
```

接着是服务器：

C++

```
// evh_server.cpp
// compile with: /LD
// post-build command: Regsvr32.exe /s evh_server.dll
#define _ATL_ATTRIBUTES 1
#include <atlbase.h>
#include <atlcom.h>
#include "evh_server.h"

[ module(dll, name="EventSource", uuid="6E46B59E-89C3-4c15-A6D8-B8A1CEC98830") ];

[coclass, event_source(com), uuid("530DF3AD-6936-3214-A83B-27B63C7997C4")]
class CSource : public IEventSource {
public:
   __event __interface IEvents;

   HRESULT FireEvent() {
      __raise MyEvent(123);
      return S_OK;
   }
};
```

再然后是客户端：

C++

```
// evh_client.cpp
// compile with: /link /OPT:NOREF
#define _ATL_ATTRIBUTES 1
#include <atlbase.h>
#include <atlcom.h>
#include <stdio.h>
#include "evh_server.h"

[ module(name="EventReceiver") ];

[ event_receiver(com) ]
class CReceiver {
public:
   HRESULT MyHandler1(int nValue) {
      printf_s("MyHandler1 was called with value %d.\n", nValue);
      return S_OK;
   }

   HRESULT MyHandler2(int nValue) {
      printf_s("MyHandler2 was called with value %d.\n", nValue);
      return S_OK;
   }

   void HookEvent(IEventSource* pSource) {
      __hook(&IEvents::MyEvent, pSource, &CReceiver::MyHandler1);
      __hook(&IEvents::MyEvent, pSource, &CReceiver::MyHandler2);
   }

   void UnhookEvent(IEventSource* pSource) {
      __unhook(&IEvents::MyEvent, pSource, &CReceiver::MyHandler1);
      __unhook(&IEvents::MyEvent, pSource, &CReceiver::MyHandler2);
   }
};

int main() {
   // Create COM object
   CoInitialize(NULL);
   {
      IEventSource* pSource = 0;
      HRESULT hr = CoCreateInstance(__uuidof(CSource), NULL,         CLSCTX_ALL,
__uuidof(IEventSource), (void **) &pSource);
      if (FAILED(hr)) {
         return -1;
      }

      // Create receiver and fire event
      CReceiver receiver;
      receiver.HookEvent(pSource);
      pSource->FireEvent();
      receiver.UnhookEvent(pSource);
   }
   CoUninitialize();
```

```
    return 0;
}
```

输出

Output

```
MyHandler1 was called with value 123.
MyHandler2 was called with value 123.
```

依赖于布局的 COM 事件

布局依赖性只是 COM 编程中的一个问题。 在本机和托管事件处理中，处理程序的签名（返回类型、调用约定和参数）必须与其事件匹配，但处理程序的名称不必与其事件匹配。

但是，在 COM 事件处理中，如果将 `layout_dependent` 的 `event_receiver` 参数设置为 **`true`**，则将强制名称和签名匹配。 事件接收器和挂钩事件中处理程序的名称和签名必须完全匹配。

当 `layout_dependent` 设置为 **`false`** 时，触发事件方法与挂钩方法（其委托）之间的调用约定和存储类（虚拟、静态等）可以混合和匹配。 使用 `layout_dependent` = **`true`** 效率略高。

例如，假设 `IEventSource` 定义为具有下列方法：

C++

```
[id(1)] HRESULT MyEvent1([in] int value);
[id(2)] HRESULT MyEvent2([in] int value);
```

假定事件源具有以下形式：

C++

```
[coclass, event_source(com)]
class CSource : public IEventSource {
public:
    __event __interface IEvents;

    HRESULT FireEvent() {
        MyEvent1(123);
        MyEvent2(123);
        return S_OK;
    }
};
```

则在事件接收器中，挂钩到 `IEventSource` 中的方法的任何处理程序必须与其名称和签名匹配，如下所示：

C++

```
[coclass, event_receiver(com, true)]
class CReceiver {
public:
   HRESULT MyEvent1(int nValue) {  // name and signature matches MyEvent1
      ...
   }
   HRESULT MyEvent2(E c, char* pc) {  // signature doesn't match MyEvent2
      ...
   }
   HRESULT MyHandler1(int nValue) {  // name doesn't match MyEvent1 (or 2)
      ...
   }
   void HookEvent(IEventSource* pSource) {
      __hook(IFace, pSource);  // Hooks up all name-matched events
                               // under layout_dependent = true
      __hook(&IFace::MyEvent1, pSource, &CReceive::MyEvent1);   // valid
      __hook(&IFace::MyEvent2, pSource, &CSink::MyEvent2);   // not valid
      __hook(&IFace::MyEvent1, pSource, &CSink:: MyHandler1); // not valid
   }
};
```

另请参阅

事件处理

Last updated on 2025/11/07

Microsoft 专用的修饰符

本节从以下方面描述特定于 Microsoft 的 C++ 扩展：

- 基寻址，用一个指针充当其他指针可进行偏移的基准的做法
- 函数调用约定
- 使用 __declspec 关键字声明的扩展存储类特性
- __w64 关键字

Microsoft 特定关键字

许多特定于 Microsoft 的关键字可用于修改声明符以构成派生类型。 有关声明符的详细信息，请参阅声明符。

展开表

关键字	含义	是否已用于构成派生类型?
__based	后跟的名称将 32 位偏移量声明为包含在声明中的 32 位基。	是
__cdecl	后面的名称遵循 C 语言的命名和调用约定。	是
__declspec	紧随其后的名称指定 Microsoft 特定的存储类属性。	否
__fastcall	后续的名称用于声明一个函数，该函数在可用时使用寄存器进行自变量传递，而不是使用堆栈。	是
__restrict	类似于 __declspec（restrict），但用于变量。	否
__stdcall	后跟的名称指定遵循标准调用约定的函数。	是
__w64	将数据类型标记为 64 位编译器上较大数据类型。	否
__unaligned	指定指向类型或其他数据的指针未对齐。	否
__vectorcall	紧随其后的名称声明了一个函数，该函数在可能的情况下使用寄存器（包括 SSE 寄存器）来传递参数，而不是使用堆栈。	是

另请参阅

C++ 语言参考

Last updated on 2025/11/07

基于寻址

本节包括下列主题：

- __based 语法
- 基指针

另请参阅

Microsoft 专用的修饰符

Last updated on 2025/11/07

`__based` 语法

Microsoft 专用

需要精确控制分配对象（基于静态和动态的数据）的段时，基于地址的寻址非常有用。

32 位和 64 位编译中可接受的基于地址的唯一形式是“基于指针”，该类型定义一个类型，该类型包含 32 位或 64 位偏移量到 32 位或 64 位基或基于 `void`。

语法

based-range-modifier:

  __based(*base-expression* **)**

base-expression:

  based-variable *based-abstract-declarator* *segment-name* *segment-cast*

based-variable:

  identifier

based-abstract-declarator:

  abstract-declarator

base-type:

  type-name

结束 Microsoft 专用

另请参阅

基于指针

Last updated on 2025/11/07

基指针 (C++)

`__based` 关键字使你能够基于指针（作为现有指针的偏移量的指针）声明指针。 `__based` 关键字特定于 Microsoft。

语法

```
type __based( base ) declarator
```

备注

基于指针地址的指针是 32 位或 64 位编译中唯一有效的 `__based` 关键字形式。 对于 Microsoft 32 位 C/C++ 编译器，基指针是相对于 32 位指针基的 32 位偏移量。 一个针对 64 位环境的类似限制保留，其中基指针是相对于 64 位基的 64 位偏移量。

基于指针的指针的用途之一是用于包含指针的永久标识符。 可将包含基于指针的指针的链接列表保存到磁盘，然后重新加载到内存中的另一个位置，并且指针保持有效。 例如：

C++

```
// based_pointers1.cpp
// compile with: /c
void *vpBuffer;
struct llist_t {
   void __based( vpBuffer ) *vpData;
   struct llist_t __based( vpBuffer ) *llNext;
};
```

将指针 `vpBuffer` 分配给程序中后面某个时间点分配的内存地址。 相对于 `vpBuffer` 的值重新定位链接的列表。

> **注意**
>
> 还可以使用**内存映射文件**来保留包含指针的标识符。

当取消对基指针的引用时，必须显式指定基或通过声明隐式公开基。

为了与以前的版本兼容，_based 是 的同义词，除非指定了编译器选项 `__based`。

示例

下面的代码演示了通过更改其基来更改基指针。

C++

```
// based_pointers2.cpp
// compile with: /EHsc
#include <iostream>

int a1[] = { 1,2,3 };
int a2[] = { 10,11,12 };
int *pBased;

typedef int __based(pBased) * pBasedPtr;

using namespace std;
int main() {
   pBased = &a1[0];
   pBasedPtr pb = 0;

   cout << *pb << endl;
   cout << *(pb+1) << endl;

   pBased = &a2[0];

   cout << *pb << endl;
   cout << *(pb+1) << endl;
}
```

Output

```
1
2
10
11
```

另请参阅

关键字

alloc_text

Last updated on 2025/11/07

调用约定

Visual C/C++ 编译器提供了用于调用内部函数和外部函数的几个不同的约定。 了解这些不同的方法有助于调试程序以及将你的代码与汇编语言例程链接。

本主题中的各个主题说明了调用约定之间的差异、如何传递参数以及函数如何返回值。 它们也讨论了裸函数调用以及使你能够写入自己的 prolog 和 epilog 代码的高级功能。

若要了解 x64 处理器的调用约定，请参阅调用约定。

本部分中的主题

- 参数传递和命名约定（`__cdecl`、`__stdcall`、`__fastcall` 等）
- 调用示例：函数原型和调用
- 使用裸函数调用编写自定义 prolog/epilog 代码
- 浮点协处理器和调用约定
- 已过时调用约定

另请参阅

Microsoft 专用的修饰符

Last updated on 2025/11/07

自变量传递和命名约定

Microsoft 专用

利用 Microsoft C++ 编译器，你可以指定在函数和调用方之间传递参数和返回值的约定。 并非所有约定都在所有支持的平台上可用，某些约定使用平台特定的实现。 在大多数情况下，将忽略在特定平台上指定不支持的约定的关键字或编译器开关，并将使用平台默认约定。

在 x86 平台上，所有参数在传递时都将加宽到 32 位。 返回值也将加宽到 32 位，并将通过 EAX 寄存器返回，但在 EDX:EAX 寄存器对中返回的 8 字节结构除外。 更大的结构将在 EAX 寄存器中作为指向隐藏返回结构的指针返回。 参数将从右到左推送到堆栈中。 不是 POD 的结构不会在寄存器中返回。

编译器将生成 prolog 和 epilog 代码来保存并还原 ESI、EDI、EBX 和 EBP 寄存器（如果在函数中使用了它们）。

> **注意**
>
> 当结构、联合或类由值从函数返回时，类型的所有定义需要相同，否则程序可能在运行时失败。

有关如何定义自己的函数 prolog 和 epilog 代码的信息，请参阅裸函数调用。

有关面向 x64 平台的代码中的默认调用约定的信息，请参阅 x64 调用约定。 有关面向 ARM 平台的代码中的调用约定问题的信息，请参阅常见的 Visual C++ ARM 迁移问题。

Visual C/C++ 编译器支持下列调用约定。

展开表

关键字	堆栈清理	参数传递
__cdecl	调用方	在堆栈上按相反顺序推送参数（从右到左）
__clrcall	不适用	按顺序将参数加载到 CLR 表达式堆栈上（从左到右）。
__stdcall	被调用方	在堆栈上按相反顺序推送参数（从右到左）
__fastcall	被调用方	存储在寄存器中，然后在堆栈上推送
__thiscall	被调用方	在堆栈上推送；存储在 ECX 中的 `this` 指针
__vectorcall	被调用方	存储在寄存器中，然后按相反顺序在堆栈上推送（从右到左）

若要了解相关信息，请参阅已过时的调用约定。

另请参阅

调用约定

Last updated on 2025/11/07

__cdecl

`__cdecl` 是 C 和 C++ 程序的默认调用约定。 由于堆栈已由调用方清理，因此它可以执行 `vararg` 函数。 `__cdecl` 调用约定创建的可执行文件大于 __stdcall 调用约定创建的可执行文件，这是因为它要求每个函数调用包括堆栈清理代码。 以下列表显示此调用约定的实现。 `__cdecl` 修饰符是 Microsoft 专用的。

展开表

元素	实现
参数传递顺序	从右到左。
堆栈维护职责	调用函数从堆栈中弹出自变量。
名称修饰约定	下划线字符 (_) 作为名称的前缀，导出使用 C 链接的 __cdecl 函数时除外。
大小写转换约定	不执行任何大小写转换。

> **注意**
>
> 有关信息，请参阅**修饰名**。

将 `__cdecl` 修饰符放置在变量或者函数名称的前面。 由于 C 命名和调用约定为默认值，因此你只能在指定 `__cdecl` (vectorcall)、`/Gv` (stdcall) 或 `/Gz` (fastcall) 编译器选项时，在 x86 代码中使用 `/Gr`。 /Gd 编译器选项强制执行 `__cdecl` 调用约定。

在 ARM 和 x64 处理器上，接受 `__cdecl`，但编译器一般会忽略它。 按照 ARM 和 x64 上的约定，自变量将尽可能传入寄存器，后续自变量传递到堆栈中。 在 x64 代码中，使用 `__cdecl` 可重写 /Gv 编译器选项并使用默认 x64 调用约定。

对于非静态类函数，如果函数是超行定义的，则调用约定修饰符不必在超行定义中指定。 也就是说，对于类非静态成员方法，在定义时假定声明期间指定的调用约定。 给定此类定义：

C++

```
struct CMyClass {
   void __cdecl mymethod();
};
```

此：

C++

```
void CMyClass::mymethod() { return; }
```

等效于此：

```C++
void __cdecl CMyClass::mymethod() { return; }
```

为了与以前的版本兼容，cdecl 和 _cdecl 是 的同义词，除非指定了编译器选项 **/Za（禁用语言扩展）** `__cdecl`。

示例

在下面的示例中，将指示编译器对 `system` 函数使用 C 命名和调用约定。

```C++
// Example of the __cdecl keyword on function
int __cdecl system(const char *);
// Example of the __cdecl keyword on function pointer
typedef BOOL (__cdecl *funcname_ptr)(void * arg1, const char * arg2, DWORD flags,
...);
```

另请参阅

自变量传递和命名约定
关键字

Last updated on 2025/11/07

__clrcall

指定只能从托管代码调用的函数。 对所有只能从托管代码调用的虚函数使用 __clrcall。 但是，此调用约定不能用于从本机代码调用的函数。 __clrcall 修饰符是 Microsoft 专用的。

当通过指针从托管函数调用到虚拟托管函数或从托管函数调用到托管函数时，可使用 __clrcall 来提高性能。

入口点是编译器生成的单独函数。 如果函数同时具有本机和托管入口点，则其中一个将是具有函数实现的实际函数。 其他函数将是调用到实际函数的单独函数（形式转换 (thunk)）并允许公共语言运行时执行 PInvoke。 将函数标记为 __clrcall 时，可指示函数实现必须是 MSIL，并且不生成本机入口点函数。

当采用本机函数的地址时，如果未指定 __clrcall，编译器将使用本机入口点。 __clrcall 指示函数为托管函数，并且不需要经历从托管到本机的转换。 在这种情况下，编译器将使用托管入口点。

当使用了 `/clr`（不是 `/clr:pure` 或 `/clr:safe`）而未使用 __clrcall 时，采用函数的地址将始终返回本机入口点函数的地址。 当使用了 __clrcall 时，不会创建本机入口点函数，因此你将获得托管函数的地址，而不是入口点形式转换函数的地址。 有关详细信息，请参阅双重形式转换。 “/clr:pure”和“/clr:safe”编译器选项在 Visual Studio 2015 中已弃用，并且在 Visual Studio 2017 中不受支持。

/clr（公共语言运行时编译）表示所有函数和函数指针都是 __clrcall，并且编译器不允许将编译单位中的函数标记为 __clrcall 之外的任何内容。 使用 /clr:pure 时，只能在函数指针和外部声明上指定 __clrcall。

可直接从使用 /clr 编译的现有 C++ 代码调用 __clrcall 函数（只要该函数具有 MSIL 实现）。 例如，无法直接从具有内联 asm 的函数调用 __clrcall 函数，并且无法调用特定于 CPU 的内部函数，即使这些函数是使用 编译的。 `/clr`

__clrcall 函数指针仅能在创建它们的应用程序域中使用。 不要跨应用程序域传递 __clrcall 函数指针，而应使用 。CrossAppDomainDelegate 有关详细信息，请参阅应用程序域和 Visual C++。

示例

请注意，当使用 __clrcall 声明函数时，将根据需要生成代码；例如，当调用函数时。

C++

```
// clrcall2.cpp
// compile with: /clr
using namespace System;
int __clrcall Func1() {
```

```
   Console::WriteLine("in Func1");
   return 0;
}

// Func1 hasn't been used at this point (code has not been generated),
// so runtime returns the address of a stub to the function
int (__clrcall *pf)() = &Func1;

// code calls the function, code generated at difference address
int i = pf();   // comment this line and comparison will pass

int main() {
   if (&Func1 == pf)
      Console::WriteLine("&Func1 == pf, comparison succeeds");
   else
      Console::WriteLine("&Func1 != pf, comparison fails");

   // even though comparison fails, stub and function call are correct
   pf();
   Func1();
}
```

Output

```
in Func1
&Func1 != pf, comparison fails
in Func1
in Func1
```

以下示例显示，你可以定义函数指针以便将该函数指针声明为仅从托管代码调用。 这样，编译器便能直接调用托管函数并避免本机入口点（双形式转换 (thunk) 问题）。

C++

```
// clrcall3.cpp
// compile with: /clr
void Test() {
   System::Console::WriteLine("in Test");
}

int main() {
   void (*pTest)() = &Test;
   (*pTest)();

   void (__clrcall *pTest2)() = &Test;
   (*pTest2)();
}
```

另请参阅

自变量传递和命名约定

关键字

Last updated on 2025/11/07

__stdcall

`__stdcall` 调用约定用于调用 Win32 API 函数。 被调用方将清理堆栈，以便让编译器生成 `vararg` 函数 `__cdecl`。 使用此调用约定的函数需要一个函数原型。 `__stdcall` 修饰符是 Microsoft 专用的。

语法

> *return-type* `__stdcall` *function-name*[`(` *argument-list* `)`]
>
> `auto` `__stdcall` *函数名称*[`(` *参数列表* `)`] [`->` *return-type*]

注解

以下列表显示此调用约定的实现。

展开表

元素	实现
参数传递顺序	从右到左。
参数传递约定	按值，除非传递指针或引用类型。
堆栈维护职责	调用的函数从堆栈中弹出自己的参数。
名称修饰约定	下划线 (`_`) 是名称的前缀。 名称后跟后面是自变量列表中的字节数（采用十进制）的符号 (`@`)。 因此，声明为 `int func( int a, double b )` 的函数按如下所示进行修饰： `_func@12`
大小写转换约定	无

/Gz 编译器选项为未使用不同调用约定显式声明的所有函数指定 `__stdcall`。

为了与以前的版本兼容，除非指定了编译器选项 `_stdcall`，否则 `__stdcall` 是 `/Za` 的同义词。

使用 `__stdcall` 修饰符声明的函数返回值的方式与使用 __cdecl 声明的函数相同。

在 ARM 和 x64 处理器上， `__stdcall` 由编译器接受和忽略；在 ARM 和 x64 体系结构上，按照约定，自变量将传入寄存器（如果可能）且后续自变量将在堆栈上传递。

对于非静态类函数，如果函数是超行定义的，则调用约定修饰符不必在超行定义中指定。 也就是说，对于类非静态成员方法，在定义时假定声明期间指定的调用约定。 给定此类定义，

```
C++
struct CMyClass {
   void __stdcall mymethod();
};
```

this

```
C++
void CMyClass::mymethod() { return; }
```

等效于此

```
C++
void __stdcall CMyClass::mymethod() { return; }
```

示例

在以下示例中，使用 `__stdcall` 将生成所有作为标准调用处理的 `WINAPI` 函数类型：

```
C++
// Example of the __stdcall keyword
#define WINAPI __stdcall
// Example of the __stdcall keyword on function pointer
typedef BOOL (__stdcall *funcname_ptr)(void * arg1, const char * arg2, DWORD flags,
...);
```

另请参阅

自变量传递和命名约定
关键字

Last updated on 2025/11/07

__fastcall

Microsoft 专用

`__fastcall` 调用约定指定尽可能在寄存器中传递函数的自变量。 此调用约定仅适用于 x86 体系结构。 以下列表显示此调用约定的实现。

展开表

元素	实现
参数传递顺序	在自变量列表中按从左到右的顺序找到的前两个 `DWORD` 或更小自变量将在 ECX 和 EDX 寄存器中传递；所有其他自变量在堆栈上从右向左传递。
堆栈维护职责	已调用函数会弹出显示堆栈中的参数。
名称修饰约定	At 符号 (@) 是名称的前缀；参数列表中的字节数（在十进制中）前面的 at 符号是名称的后缀。
大小写转换约定	不执行任何大小写转换。
类、结构和并集	被视为“多字节”类型（无论大小）并在堆栈上传递。
枚举和枚举类	如果它们的基础类型是通过寄存器传递的，则通过寄存器传递。 例如，如果基础类型是大小为 8、16 或 32 位的 `int` 或 `unsigned int`。

> **注意**
>
> 将来版本的编译器可使用其他寄存器来存储参数。

使用 /Gr 编译器选项会导致将模块中的每个函数编译为 `__fastcall`，除非使用冲突特性来声明函数，或者函数的名称为 `main`。

`__fastcall` 关键字由面向 ARM 和 x64 体系结构的编译器接受和忽略；在 x64 芯片上，按照约定，前四个参数在寄存器中传递（如果可能），而其他参数在堆栈上传递。 有关详细信息，请参阅 x64 调用约定。 在 ARM 芯片上，寄存器中可以传递最多四个整数参数和八个浮点参数，而其他参数在堆栈上传递。

对于非静态类函数，如果函数是超行定义的，则调用约定修饰符不必在超行定义中指定。 也就是说，对于类非静态成员方法，在定义时假定声明期间指定的调用约定。 给定此类定义：

```
C++
```

```
struct CMyClass {
   void __fastcall mymethod();
};
```

此：

```C++
void CMyClass::mymethod() { return; }
```

等效于此：

```C++
void __fastcall CMyClass::mymethod() { return; }
```

为了与以前的版本兼容，除非指定了编译器选项 `_fastcall`，否则 `__fastcall` 是 的同义词。

示例

在下面的示例中，函数 `DeleteAggrWrapper` 是寄存器中传递的参数：

```C++
// Example of the __fastcall keyword
#define FASTCALL    __fastcall

void FASTCALL DeleteAggrWrapper(void* pWrapper);
// Example of the __ fastcall keyword on function pointer
typedef BOOL (__fastcall *funcname_ptr)(void * arg1, const char * arg2, DWORD flags,
...);
```

结束 Microsoft 专用

另请参阅

自变量传递和命名约定
关键字

Last updated on 2025/11/07

`__thiscall`

特定于 Microsoft `__thiscall` 的调用约定用于 x86 体系结构上的 C++ 类成员函数。 它是成员函数使用的默认调用约定，该约定不使用变量参数（`vararg` 函数）。

在 `__thiscall` 下，被调用方清理堆栈，这对于 `vararg` 函数是不可能的。 自变量将从右到左推送到堆栈中。 指针 `this` 通过注册 ECX 传递，而不是在堆栈上传递。

在 ARM、ARM64 和 x64 计算机上， `__thiscall` 由编译器接受和忽略。 这是因为它们默认使用基于寄存器的调用约定。

使用 `__thiscall` 的原因之一是在类中成员函数默认使用 `__clrcall`。 在这种情况下，可以使用 `__thiscall` 确保各个成员函数可以从本机代码调用。

采用 /clr:pure 进行编译时，除非另有规定，否则所有函数和函数指针都是 `__clrcall`。 `/clr:pure` 和 `/clr:safe` 编译器选项在 Visual Studio 2015 中已弃用，并且在 Visual Studio 2017 中不受支持。

`vararg` 成员函数使用 `__cdecl` 调用约定。 所有函数参数都推送在堆栈上， `this` 指针放在最后一个堆栈上。

由于此调用约定仅适用于 C++，因此它没有 C 名称修饰方案。

在非静态类成员函数外行定义时，仅在声明中指定调用约定修饰符。 无需在行外定义上再次指定它。 编译器使用在定义点声明期间指定的调用约定。

另请参阅

参数传递和命名约定

Last updated on 2025/11/07

__vectorcall

Microsoft 专用

`__vectorcall` 调用约定指定尽可能在寄存器中传递函数的自变量。 `__vectorcall` 对自变量使用的寄存器的数目多于 __fastcall 或默认的 x64 调用约定对自变量使用的寄存器的数目。 `__vectorcall` 调用约定仅受到包括流式处理 SIMD 扩展 2 (SSE2) 和更高版本的 x86 和 x64 处理器上的本机代码的支持。 使用 `__vectorcall` 可使传递多个浮点或 SIMD 矢量自变量的函数加速，并执行利用寄存器中加载的自变量的操作。 以下列表显示了 `__vectorcall` 的 x86 和 x64 实现所共有的功能。 本文的后面部分解释了这些差异。

展开表

元素	实现
C 名称修饰约定	函数名的前缀为两个“at”符号 (@@)，后接参数列表中的字节数（以十进制形式表示）。
大小写转换约定	不执行任何大小写转换。

使用 /Gv 编译器选项将导致模块中的每个函数编译为 `__vectorcall`，除非函数是成员函数、利用冲突调用约定特性进行声明、使用 `vararg` 变量自变量列表或具有名称 `main`。

你可通过寄存器在 `__vectorcall` 函数中传递三种自变量：整数类型值、矢量类型值和同类矢量聚合 (HVA) 值。

整数类型满足两个要求：它适合处理器的本机寄存器大小（例如，x86 计算机上的 4 个字节或 x64 计算机上的 8 个字节），而且它可以转换成与寄存器长度一样的整数，并且不用更改其位表示形式就能转换回来。 例如，所有可以提升到 x86 上的 `int`（x64 上的 `long long`）- 例如，`char` 或 `short`（或可转换为 `int` (x64 上的 `long long`)）是整数类型，它们无需更改即可返回其原始类型。 整数类型包括指针、引用和小于或等于 4 个字节（x64 上为 8 个字节）的 `struct` 或 `union` 类型。 在 x64 平台上，更大的 `struct` 和 `union` 类型通过引用传递给调用方所分配的内存；在 x86 平台上，通过堆栈上的值传递它们。

矢量类型是一个浮点类型（例如，`float` 或 `double`）或一个 SIMD 矢量类型（例如，`__m128` 或 `__m256`）。

HVA 类型是复合类型，包含四个具有相同矢量类型的数据成员。 HVA 类型具有和其成员的矢量类型相同的对齐需求。 这是有关 HVA `struct` 定义的示例，该定义包含三个相同的矢量类型且具有 32 字节对齐方式：

C++

```
typedef struct {
    __m256 x;
```

```
    __m256 y;
    __m256 z;
} hva3;    // 3 element HVA type on __m256
```

使用头文件中的 `__vectorcall` 关键字对函数进行显式声明，以使独立编译的代码在链接时不会出错。 函数必须为使用 `__vectorcall` 的原型，而不能使用 `vararg` 可变长度自变量列表。

成员函数可以通过使用 `__vectorcall` 说明符进行声明。 通过寄存器传递隐藏的 `this` 指针作为第一个整数类型参数。

在 ARM 计算机上，`__vectorcall` 由编译器接受和忽略。 在 ARM64EC 上，`__vectorcall` 不受编译器支持并被拒绝。

对于非静态类成员函数，如果函数是超行定义的，则调用约定修饰符不必在超行定义中指定。 也就是说，对于类非静态成员，在定义时假定声明期间指定的调用约定。 给定此类定义：

C++

```
struct MyClass {
    void __vectorcall mymethod();
};
```

此：

C++

```
void MyClass::mymethod() { return; }
```

等效于此：

C++

```
void __vectorcall MyClass::mymethod() { return; }
```

必须在创建指向 `__vectorcall` 函数的指针时，指定 `__vectorcall` 调用约定修饰符。 下一个示例针对指向 `typedef` 函数的指针创建 `__vectorcall`，该函数采用四个 `double` 自变量并返回一个 `__m256` 值：

C++

```
typedef __m256 (__vectorcall * vcfnptr)(double, double, double, double);
```

为了与以前的版本兼容，除非指定了编译器选项 `__vectorcall`，否则 `/Za` 是 的同义词。

x64 上的 __vectorcall 约定

x64 上的 `__vectorcall` 调用约定扩展标准 x64 调用约定以利用其他寄存器。 根据参数列表中的位置将整数类型参数和矢量类型参数映射到寄存器。 将 HVA 自变量分配给未使用的矢量寄存器。

当前四个自变量（按从左到右的顺序）都是整数类型参数时，它们将传递到对应该位置的寄存器，即 RCX、RDX、R8 或 R9。 隐藏的 `this` 指针被视为第一个整数类型参数。 当前四个自变量之一中的 HVA 自变量无法传入可用寄存器时，对调用方分配的内存的引用将传入相应的整数类型寄存器。 第四参数位置之后的整数类型参数在堆栈上传递。

当前六个参数（按从左到右的顺序）都是矢量类型参数时，将根据参数位置通过 0 到 5 的 SSE 矢量寄存器中的值传递它们。 浮点和 `__m128` 类型在 XMM 寄存器中传递，而 `__m256` 类型在 YMM 寄存器中传递。 这与标准 x64 调用约定不同，因为矢量类型是通过值而不是引用传递的，并且还使用了其他寄存器。 为矢量类型参数分配的影子堆栈空间固定为 8 个字节，而 /homeparams 选项不适用。 在第七个参数位置和后面的参数位置的矢量类型参数在堆栈上通过对由调用方分配的内存的引用进行传递。

在为矢量自变量分配寄存器后，只要整个 HVA 有足量的可用寄存器，HVA 自变量的数据成员就会以升序分配给未使用的矢量寄存器 XMM0 至 XMM5（或者，对于 `__m256` 类型，为 YMM0 至 YMM5）。 如果没有足够多的可用寄存器，HVA 自变量将通过对由调用方分配的内存的引用进行传递。 HVA 自变量的影子堆栈空间固定为 8 个字节，且带有未定义的内容。 在参数列表中按从左到右的顺序将 HVA 参数分配给寄存器，并且这些参数可处于任意位置。 位于未分配给矢量寄存器的前四个参数位置中的任一位置的 HVA 参数将通过由对应于该位置的整数寄存器中的引用进行传递。 在第四个参数位置后通过引用传递的 HVA 自变量将被推入堆栈。

如果可能，`__vectorcall` 函数的结果将由寄存器中的值返回。 整数类型的结果（包括小于或等于 8 字节的结构或联合）按 RAX 中的值返回。 根据大小，矢量类型结果在 XMM0 或 YMM0 中按值返回。 HVA 结果具有每个由 XMM0:XMM3 或 YMM0:YMM3 寄存器中的值返回的数据元素，取决于元素大小。 不适合对应的寄存器的结果类型将通过对由调用方分配的内存的引用返回。

堆栈在 `__vectorcall` 的 x64 实现中由调用方保持。 调用方 prolog 和 epilog 代码分配并清除被调用函数的堆栈。 参数从右向左推入堆栈；并且为传递到寄存器中的参数分配影子堆栈空间。

示例：

C++

```
// crt_vc64.c
// Build for amd64 with: cl /arch:AVX /W3 /FAs crt_vc64.c
// This example creates an annotated assembly listing in
// crt_vc64.asm.
```

```
#include <intrin.h>
#include <xmmintrin.h>

typedef struct {
   __m128 array[2];
} hva2;    // 2 element HVA type on __m128

typedef struct {
   __m256 array[4];
} hva4;    // 4 element HVA type on __m256

// Example 1: All vectors
// Passes a in XMM0, b in XMM1, c in YMM2, d in XMM3, e in YMM4.
// Return value in XMM0.
__m128 __vectorcall
example1(__m128 a, __m128 b, __m256 c, __m128 d, __m256 e) {
   return d;
}

// Example 2: Mixed int, float and vector parameters
// Passes a in RCX, b in XMM1, c in R8, d in XMM3, e in YMM4,
// f in XMM5, g pushed on stack.
// Return value in YMM0.
__m256 __vectorcall
example2(int a, __m128 b, int c, __m128 d, __m256 e, float f, int g) {
   return e;
}

// Example 3: Mixed int and HVA parameters
// Passes a in RCX, c in R8, d in R9, and e pushed on stack.
// Passes b by element in [XMM0:XMM1];
// b's stack shadow area is 8-bytes of undefined value.
// Return value in XMM0.
__m128 __vectorcall example3(int a, hva2 b, int c, int d, int e) {
   return b.array[0];
}

// Example 4: Discontiguous HVA
// Passes a in RCX, b in XMM1, d in XMM3, and e is pushed on stack.
// Passes c by element in [YMM0,YMM2,YMM4,YMM5], discontiguous because
// vector arguments b and d were allocated first.
// Shadow area for c is an 8-byte undefined value.
// Return value in XMM0.
float __vectorcall example4(int a, float b, hva4 c, __m128 d, int e) {
   return b;
}

// Example 5: Multiple HVA arguments
// Passes a in RCX, c in R8, e pushed on stack.
// Passes b in [XMM0:XMM1], d in [YMM2:YMM5], each with
// stack shadow areas of an 8-byte undefined value.
// Return value in RAX.
int __vectorcall example5(int a, hva2 b, int c, hva4 d, int e) {
   return c + e;
}
```

```
// Example 6: HVA argument passed by reference, returned by register
// Passes a in [XMM0:XMM1], b passed by reference in RDX, c in YMM2,
// d in [XMM3:XMM4].
// Register space was insufficient for b, but not for d.
// Return value in [YMM0:YMM3].
hva4 __vectorcall example6(hva2 a, hva4 b, __m256 c, hva2 d) {
   return b;
}

int __cdecl main( void )
{
   hva4 h4;
   hva2 h2;
   int i;
   float f;
   __m128 a, b, d;
   __m256 c, e;

   a = b = d = _mm_set1_ps(3.0f);
   c = e = _mm256_set1_ps(5.0f);
   h2.array[0] = _mm_set1_ps(6.0f);
   h4.array[0] = _mm256_set1_ps(7.0f);

   b = example1(a, b, c, d, e);
   e = example2(1, b, 3, d, e, 6.0f, 7);
   d = example3(1, h2, 3, 4, 5);
   f = example4(1, 2.0f, h4, d, 5);
   i = example5(1, h2, 3, h4, 5);
   h4 = example6(h2, h4, c, h2);
}
```

x86 上的 __vectorcall 约定

`__vectorcall` 调用约定遵循 32 位整数类型参数的 `__fastcall` 约定，并利用矢量类型和 HVA 自变量的 SSE 矢量寄存器。

将在参数列表中从左至右发现的前两个整数类型参数分别放入 ECX 和 EDX 中。 隐藏的 `this` 指针被视为第一个整数类型参数，并且被传入 ECX。 前六个矢量类型参数通过 XMM 或 YMM 寄存器中的 SSE 矢量寄存器 0 到 5（具体取决于参数大小）中的值传递。

前六个矢量类型参数（按从左至右的顺序）按 SSE 矢量寄存器 0 到 5 中的值传递。 浮点和 `__m128` 类型在 XMM 寄存器中传递，而 `__m256` 类型在 YMM 寄存器中传递。 不会为寄存器传入的矢量类型参数分配影子堆栈空间。 第七个矢量类型参数和后面的矢量类型参数在堆栈上通过对由调用方分配的内存的引用进行传递。 编译器错误 C2719 的限制不适用于这些自变量。

在为矢量自变量分配寄存器后，只要整个 HVA 有足量的可用寄存器，HVA 自变量的数据成员就会以升序分配给未使用的矢量寄存器 XMM0 至 XMM5（或者，对于 `__m256` 类型，为 YMM0 至 YMM5）。 如果没有足量的可用寄存器，则 HVA 自变量将在堆栈上通过对由调用方分配的内存

的引用进行传递。 不会为 HVA 参数分配堆栈影子空间。 在参数列表中按从左到右的顺序将 HVA 参数分配给寄存器，并且这些参数可处于任意位置。

如果可能，`__vectorcall` 函数的结果将由寄存器中的值返回。 整数类型的结果（包括小于或等于 4 字节的结构或联合）按 EAX 中的值返回。 小于或等于 8 字节的整数类型结构或联合由 EDX:EAX 中的值返回。 根据大小，矢量类型结果在 XMM0 或 YMM0 中按值返回。 HVA 结果具有每个由 XMM0:XMM3 或 YMM0:YMM3 寄存器中的值返回的数据元素，取决于元素大小。 其他结果类型通过对由调用方分配的内存的引用返回。

`__vectorcall` 的 x86 实现遵循由调用方从右向左推入堆栈的自变量的约定，而调用的函数可在其返回之前清除堆栈。 仅将未置于寄存器上的自变量推入堆栈。

示例：

C++

```
// crt_vc86.c
// Build for x86 with: cl /arch:AVX /W3 /FAs crt_vc86.c
// This example creates an annotated assembly listing in
// crt_vc86.asm.

#include <intrin.h>
#include <xmmintrin.h>

typedef struct {
   __m128 array[2];
} hva2;    // 2 element HVA type on __m128

typedef struct {
   __m256 array[4];
} hva4;    // 4 element HVA type on __m256

// Example 1: All vectors
// Passes a in XMM0, b in XMM1, c in YMM2, d in XMM3, e in YMM4.
// Return value in XMM0.
__m128 __vectorcall
example1(__m128 a, __m128 b, __m256 c, __m128 d, __m256 e) {
   return d;
}

// Example 2: Mixed int, float and vector parameters
// Passes a in ECX, b in XMM0, c in EDX, d in XMM1, e in YMM2,
// f in XMM3, g pushed on stack.
// Return value in YMM0.
__m256 __vectorcall
example2(int a, __m128 b, int c, __m128 d, __m256 e, float f, int g) {
   return e;
}

// Example 3: Mixed int and HVA parameters
// Passes a in ECX, c in EDX, d and e pushed on stack.
```

```
// Passes b by element in [XMM0:XMM1].
// Return value in XMM0.
__m128 __vectorcall example3(int a, hva2 b, int c, int d, int e) {
   return b.array[0];
}

// Example 4: HVA assigned after vector types
// Passes a in ECX, b in XMM0, d in XMM1, and e in EDX.
// Passes c by element in [YMM2:YMM5].
// Return value in XMM0.
float __vectorcall example4(int a, float b, hva4 c, __m128 d, int e) {
   return b;
}

// Example 5: Multiple HVA arguments
// Passes a in ECX, c in EDX, e pushed on stack.
// Passes b in [XMM0:XMM1], d in [YMM2:YMM5].
// Return value in EAX.
int __vectorcall example5(int a, hva2 b, int c, hva4 d, int e) {
   return c + e;
}

// Example 6: HVA argument passed by reference, returned by register
// Passes a in [XMM1:XMM2], b passed by reference in ECX, c in YMM0,
// d in [XMM3:XMM4].
// Register space was insufficient for b, but not for d.
// Return value in [YMM0:YMM3].
hva4 __vectorcall example6(hva2 a, hva4 b, __m256 c, hva2 d) {
   return b;
}

int __cdecl main( void )
{
   hva4 h4;
   hva2 h2;
   int i;
   float f;
   __m128 a, b, d;
   __m256 c, e;

   a = b = d = _mm_set1_ps(3.0f);
   c = e = _mm256_set1_ps(5.0f);
   h2.array[0] = _mm_set1_ps(6.0f);
   h4.array[0] = _mm256_set1_ps(7.0f);

   b = example1(a, b, c, d, e);
   e = example2(1, b, 3, d, e, 6.0f, 7);
   d = example3(1, h2, 3, 4, 5);
   f = example4(1, 2.0f, h4, d, 5);
   i = example5(1, h2, 3, h4, 5);
   h4 = example6(h2, h4, c, h2);
}
```

结束 Microsoft 专用

另请参阅

自变量传递和命名约定
关键字

Last updated on 2025/11/07

调用示例： 函数原型和调用

Microsoft 专用

以下示例显示了使用各种调用约定执行函数调用的结果。

此示例基于以下函数主干。 将 `calltype` 替换为适当的调用约定。

C++

```
void    calltype MyFunc( char c, short s, int i, double f );
.
.
.
void    MyFunc( char c, short s, int i, double f )
    {
    .
    .
    .
    }
.
.
.
MyFunc ('x', 12, 8192, 2.7183);
```

有关详细信息，请参阅调用结果示例。

结束 Microsoft 专用

另请参阅

调用约定

Last updated on 2025/11/07

调用示例的结果

Microsoft 专用

`__cdecl`

C 修饰函数名为“`_MyFunc`”。

`__cdecl` 调用约定

`__stdcall` 和 `thiscall`

C 修饰名（`__stdcall`）是“`_MyFunc@20`”。 C 修饰名是特定于实现的。

__stdcall 和 thiscall 调用约定

`__fastcall`

C 修饰名（`__fastcall`）是“`@MyFunc@20`”。 C 修饰名是特定于实现的。

`__fastcall` 调用约定

结束 Microsoft 专用

另请参阅

调用示例：函数原型和调用

Last updated on 2025/11/07

Naked 函数调用

Microsoft 专用

无需 prolog 或 epilog 代码即可发出使用 `naked` 特性声明的函数，这使你能够使用内联汇编程序编写自己的自定义 prolog/epilog 序列。 将裸函数作为高级功能提供。 利用这些函数，您可以声明从 C/C++ 之外的上下文中调用的函数，从而作出有关参数位置或保留哪些寄存器的各种假设。 示例包括例程（如中断处理程序）。 此功能对于虚拟设备驱动程序 (VxDs) 的编写器特别有用。

你想进一步了解什么?

- naked
- 裸函数的规则和限制
- 有关编写 Prolog/Epilog 代码的注意事项

结束 Microsoft 专用

另请参阅

调用约定

Last updated on 2025/11/07

裸函数的规则和限制

Microsoft 专用

以下规则和限制适用于裸函数：

- 不允许使用 `return` 语句。
- 不允许结构化异常处理和 C++ 异常处理构造，因为它们必须在堆栈帧中展开。
- 出于同一原因，禁止任何形式的 `setjmp`。
- 禁止使用 `_alloca` 函数。
- 若要确保局部变量的初始化代码不在 prolog 序列之前出现，函数范围中不允许存在初始化的局部变量。 具体而言，函数范围中不允许有 C++ 对象的声明。 但是，嵌套的范围中可能有初始化的数据。
- 不建议使用帧指针优化（/Oy 编译器选项），但会自动为裸函数将其取消。
- 不能在函数词法范围中声明 C++ 类对象。 但是，可以在嵌套的块中声明对象。
- 在使用 `naked` 进行编译时，将忽略 关键字。
- 对于 __fastcall 裸函数，只要 C/C++ 代码中存在对某个寄存器自变量的引用，prolog 代码就应将该寄存器的值存储到该变量的堆栈位置中。 例如：

C++

```
// nkdfastcl.cpp
// compile with: /c
// processor: x86
__declspec(naked) int __fastcall  power(int i, int j) {
   // calculates i^j, assumes that j >= 0

   // prolog
   __asm {
      push ebp
      mov ebp, esp
      sub esp, __LOCAL_SIZE
     // store ECX and EDX into stack locations allocated for i and j
     mov i, ecx
     mov j, edx
   }

   {
      int k = 1;   // return value
      while (j-- > 0)
         k *= i;
      __asm {
```

```
        mov eax, k
      };
   }

   // epilog
   __asm {
      mov esp, ebp
      pop ebp
      ret
   }
}
```

结束 Microsoft 专用

另请参阅

Naked 函数调用

Last updated on 2025/11/07

有关编写 Prolog/Epilog 代码的注意事项

Microsoft 专用

在编写你自己的 prolog 和 epilog 代码序列之前，请务必了解堆栈帧的布局方式。了解如何使用 `__LOCAL_SIZE` 符号也很有用。

堆栈帧布局

此示例显示了可能出现在 32 位函数中的标准 prolog 代码：

```
push        ebp                 ; Save ebp
mov         ebp, esp            ; Set stack frame pointer
sub         esp, localbytes     ; Allocate space for locals
push        <registers>         ; Save registers
```

`localbytes` 变量表示局部变量堆栈上所需的字节数， `<registers>` 变量是表示要保存在堆栈上的寄存器列表的占位符。 推入寄存器后，您可以将任何其他适当的数据放置在堆栈上。 下面是相应的 epilog 代码：

```
pop         <registers>   ; Restore registers
mov         esp, ebp      ; Restore stack pointer
pop         ebp           ; Restore ebp
ret                       ; Return from function
```

堆栈始终向下增长（从高内存地址到低内存地址）。 基指针（`ebp`）指向 `ebp` 的推入值。 本地区域开始于 `ebp-4`。 若要访问局部变量，可通过从 `ebp` 中减去适当的值来计算 `ebp` 的偏移量。

__LOCAL_SIZE

编译器提供 `__LOCAL_SIZE` 符号，以用于函数 prolog 代码的内联汇编程序块中。 此符号用于在自定义 prolog 代码中的堆栈帧上为局部变量分配空间。

编译器确定 `__LOCAL_SIZE` 的值。 其值是所有用户定义的局部变量和编译器生成的临时变量的总字节数。 `__LOCAL_SIZE` 只能用作即时操作数；它不能在表达式中使用。 不得更改或重新定义此符号的值。 例如：

```
mov         eax, __LOCAL_SIZE           ;Immediate operand--Okay
mov         eax, [ebp - __LOCAL_SIZE]   ;Error
```

包含自定义 prolog 和 epilog 序列的 naked 函数的以下示例在 prolog 序列中使用 `__LOCAL_SIZE` 符号：

C++

```
// the__local_size_symbol.cpp
// processor: x86
__declspec ( naked ) int main() {
   int i;
   int j;

   __asm {      /* prolog */
      push   ebp
      mov      ebp, esp
      sub      esp, __LOCAL_SIZE
      }

   /* Function body */
   __asm {   /* epilog */
      mov      esp, ebp
      pop      ebp
      ret
      }
}
```

结束 Microsoft 专用

另请参阅

Naked 函数调用

Last updated on 2025/11/07

浮点协处理器和调用约定

如果正在为浮点协处理器编写汇编例程，则必须保留浮点控制字并清理协处理器堆栈，除非返回 `float` 或 `double` 值（你的函数应在 ST(0) 中返回）。

另请参阅

调用约定

Last updated on 2025/11/07

已过时调用约定

Microsoft 专用

不再支持 __pascal、__fortran 和 __syscall 调用约定。 通过使用支持的调用约定之一和适当的链接器选项，可以模拟其功能。

<windows.h> 现在支持 WINAPI 宏，该宏可转换为目标的适当调用约定。 在之前使用 PASCAL 或 __far __pascal 的位置使用 WINAPI。

结束 Microsoft 专用

另请参阅

自变量传递和命名约定

Last updated on 2025/11/07

限制（C++ AMP）

可将限制说明符应用于函数和 lambda 声明。 它会强制对函数中的代码以及使用 C++ Accelerated Massive Parallelism (C++ AMP) 运行时的应用程序中的函数的行为实施限制。

> **注意**
>
> 有关作为 `restrict` 存储类特性的一部分的 `__declspec` 关键字的信息，请参阅**限制**。

`restrict` 子句采用以下形式：

展开表

子句	说明
`restrict(cpu)`	函数可使用完整的 C++ 语言。 只有使用 restrict(cpu) 函数声明的其他函数功能可调用函数。
`restrict(amp)`	函数只能使用 C++ AMP 可为其加速的一部分 C++ 语言。
一系列 `restrict(cpu)` 和 `restrict(amp)`。	函数必须遵循 `restrict(cpu)` 和 `restrict(amp)` 的限制。 函数可由使用 `restrict(cpu)`、`restrict(amp)`、`restrict(cpu, amp)` 或 `restrict(amp, cpu)` 声明的函数调用。 `restrict(A) restrict(B)` 形式可以编写为 `restrict(A,B)`。

备注

`restrict` 关键字是上下文关键字。 限制说明符、`cpu` 和 `amp` 不是保留字。 说明符列表不可扩展。 没有 `restrict` 子句的函数与具有 `restrict(cpu)` 子句的函数相同。

包含 `restrict(amp)` 子句的函数具有以下限制：

- 函数只能调用具有 `restrict(amp)` 子句的函数。
- 函数必须可内联。
- 函数只能声明 `int`、`unsigned int`、`float` 和 `double` 变量，以及只包含这些类型的类和结构。 也允许使用 `bool`，但如果在复合类型中使用它，则它必须是 4 字节对齐的。
- Lambda 函数无法通过引用捕获，并且无法捕获指针。
- 仅支持引用和单一间接指针作为局部变量、函数自变量和返回类型。

- 不允许使用以下项：
  - 递归。
  - 使用 volatile 关键字声明的变量。
  - 虚函数。
  - 指向函数的指针。
  - 指向成员函数的指针。
  - 结构中的指针。
  - 指向指针的指针。
  - `goto` 语句。
  - Labeled 语句。
  - `try`、`catch` 或 `throw` 语句。
  - 全局变量。
  - 静态变量。 请改用 tile_static 关键字。
  - `dynamic_cast` 强制转换。
  - `typeid` 运算符。
  - asm 声明。
  - Varargs。

有关函数限制的讨论，请参阅 restrict(amp) 限制。

示例

以下示例演示如何使用 `restrict(amp)` 子句。

C++

```
void functionAmp() restrict(amp) {}
void functionNonAmp() {}

void callFunctions() restrict(amp)
{
    // int is allowed.
```

```
    int x;
    // long long int is not allowed in an amp-restricted function. This generates a
compiler error.
    // long long int y;

    // Calling an amp-restricted function is allowed.
    functionAmp();

    // Calling a non-amp-restricted function is not allowed.
    // functionNonAmp();
}
```

另请参阅

C++ AMP（C++加速大规模并行度）

Last updated on 2025/11/07

tile_static 关键字

tile_static 关键字用于声明可由线程平铺中的所有线程访问的变量。 此变量的生存期在执行到达声明点时开始，在内核函数返回时结束。 有关使用平铺的详细信息，请参阅使用平铺。

tile_static 关键字具有下列限制：

- 它只能对具有 `restrict(amp)` 修饰符的函数中的变量使用。
- 它不能对作为指针或引用类型的变量使用。
- tile_static 变量不能有初始值设定项。 不会自动调用默认构造函数和析构函数。
- 未初始化的 tile_static 变量的值是不确定的。
- 如果在产生于对 的非平铺调用的调用关系图中声明 tile_static 变量，则将生成警告并且变量的行为是不确定的`parallel_for_each`。

示例

以下示例演示如何使用 tile_static 变量在一个平铺的多个线程中累积数据。

C++

```
// Sample data:
int sampledata[] = {
    2, 2, 9, 7, 1, 4,
    4, 4, 8, 8, 3, 4,
    1, 5, 1, 2, 5, 2,
    6, 8, 3, 2, 7, 2};

// The tiles:
// 2 2    9 7    1 4
// 4 4    8 8    3 4
//
// 1 5    1 2    5 2
// 6 8    3 2    7 2

// Averages:
int averagedata[] = {
    0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0,
};

array_view<int, 2> sample(4, 6, sampledata);
array_view<int, 2> average(4, 6, averagedata);
```

```
parallel_for_each(
    // Create threads for sample.extent and divide the extent into 2 x 2 tiles.
    sample.extent.tile<2,2>(),
    [=](tiled_index<2,2> idx) restrict(amp)
    {
        // Create a 2 x 2 array to hold the values in this tile.
        tile_static int nums[2][2];
        // Copy the values for the tile into the 2 x 2 array.
        nums[idx.local[1]][idx.local[0]] = sample[idx.global];
        // When all the threads have executed and the 2 x 2 array is complete, find
the average.
        idx.barrier.wait();
        int sum = nums[0][0] + nums[0][1] + nums[1][0] + nums[1][1];
        // Copy the average into the array_view.
        average[idx.global] = sum / 4;
      }
);

for (int i = 0; i < 4; i++) {
    for (int j = 0; j < 6; j++) {
        std::cout << average(i,j) << " ";
    }
    std::cout << "\n";
}

// Output:
// 3 3 8 8 3 3
// 3 3 8 8 3 3
// 5 5 2 2 4 4
// 5 5 2 2 4 4
// Sample data.
int sampledata[] = {
    2, 2, 9, 7, 1, 4,
    4, 4, 8, 8, 3, 4,
    1, 5, 1, 2, 5, 2,
    6, 8, 3, 2, 7, 2};

// The tiles are:
// 2 2    9 7    1 4
// 4 4    8 8    3 4
//
// 1 5    1 2    5 2
// 6 8    3 2    7 2

// Averages.
int averagedata[] = {
    0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0,
};

array_view<int, 2> sample(4, 6, sampledata);
array_view<int, 2> average(4, 6, averagedata);
```

```
parallel_for_each(
    // Create threads for sample.grid and divide the grid into 2 x 2 tiles.
    sample.extent.tile<2,2>(),
    [=](tiled_index<2,2> idx) restrict(amp)
    {
        // Create a 2 x 2 array to hold the values in this tile.
        tile_static int nums[2][2];
        // Copy the values for the tile into the 2 x 2 array.
        nums[idx.local[1]][idx.local[0]] = sample[idx.global];
        // When all the threads have executed and the 2 x 2 array is complete, find
the average.
        idx.barrier.wait();
        int sum = nums[0][0] + nums[0][1] + nums[1][0] + nums[1][1];
        // Copy the average into the array_view.
        average[idx.global] = sum / 4;
      }
);

for (int i = 0; i < 4; i++) {
    for (int j = 0; j < 6; j++) {
        std::cout << average(i,j) << " ";
    }
    std::cout << "\n";
}

// Output.
// 3 3 8 8 3 3
// 3 3 8 8 3 3
// 5 5 2 2 4 4
// 5 5 2 2 4 4
```

另请参阅

Microsoft 专用的修饰符
C++ AMP 概述
parallel_for_each 函数 (C++ AMP)
演练：矩阵乘法

Last updated on 2025/11/07

`__declspec`

Microsoft 专用

用于指定存储类信息的扩展属性语法采用`__declspec`关键字，该关键字指定将以Microsoft特定存储类属性存储给定类型的实例，具体属性如下所列。 其他存储类修饰符的示例包括 `static` 和 `extern` 关键字。 但是，这些关键字是 C 和 C++ 语言的 ANSI 规范的一部分，因此，扩展属性语法并未涵盖这些关键字。 扩展属性语法简化并标准化了 Microsoft 特定的 C 和 C++ 语言扩展。

语法

decl-specifier：

__declspec (*extended-decl-modifier-seq* **)**

extended-decl-modifier-seq：

extended-decl-modifier 选择

extended-decl-modifier *extended-decl-modifier-seq*

extended-decl-modifier：

align(number **)**

allocate(" *segname* **")**

allocator

appdomain

code_seg(" *segname* **")**

deprecated

dllimport

dllexport

empty_bases

hybrid_patchable

jitintrinsic

naked

noalias

noinline

noreturn

nothrow

novtable

no_sanitize_address

process

`property(` { `get=` *get-func-name* | `,put=` *put-func-name* } `)`

`restrict`

`safebuffers`

`selectany`

`spectre(nomitigation)`

`thread`

`uuid("` *ComObjectGUID* `")`

空格分隔声明修饰符序列。 示例显示在后面的部分中。

扩展属性语法支持这些特定于Microsoft存储类属性：align、allocate、allocator、appdomain、code_seg、deprecated、dllexport、dllimport、empty_bases、jitintrinsic、naked、noalias、noinline、noreturn、nothrow、novtable、no_sanitize_address、process、restrict、safebuffers、selectany、spectre和thread。 它还支持以下 COM 对象属性：property 和 uuid。

`code_seg`、`dllexport`、`dllimport`、`empty_bases`、`naked`、`noalias`、`nothrow`、`no_sanitize_address`、`property`、`restrict`、`selectany`、`thread`和 `uuid` 存储类属性仅是应用它们的对象的声明或函数的属性。 `thread` 属性仅影响数据和对象。 `naked` 和 `spectre` 属性仅影响函数。 `dllimport` 和 `dllexport` 属性会影响函数、数据和对象。 `property`、`selectany`和 `uuid` 属性会影响 COM 对象。

为了与以前的版本兼容，除非指定编译器选项 `_declspec`，否则 `__declspec` 是 `/Za` 的同义词。

应将 `__declspec` 关键字放在简单声明的开头。 编译器忽略在 * 或 & 之后以及声明中变量标识符前面放置的任何 `__declspec` 关键字，而不发出警告。

用户定义的类型声明开头指定的 `__declspec` 属性适用于该类型的变量。 例如：

```
C++
__declspec(dllimport) class X {} varX;
```

在这种情况下，该属性适用于 `varX`。 `__declspec` 或 `class` 关键字后放置的 `struct` 属性适用于用户定义的类型。 例如：

```
C++
class __declspec(dllimport) X {};
```

在这种情况下，该属性适用于 `X`。

对简单声明使用 `__declspec` 属性的一般准则如下：

`decl-specifier-seq` `init-declarator-list` `;`

此外，`decl-specifier-seq` 应包含基类型（例如 `int`、`float`、`typedef` 或类名）、存储类（例如 `static`、`extern`）或 `__declspec` 扩展。 另外，`init-declarator-list` 还应包含声明的指针部分。 例如：

C++

```
__declspec(selectany) int * pi1 = 0;   //Recommended, selectany & int both part of
decl-specifier
int __declspec(selectany) * pi2 = 0;   //OK, selectany & int both part of decl-
specifier
int * __declspec(selectany) pi3 = 0;   //ERROR, selectany is not part of a
declarator
```

以下代码声明整数线程局部变量，并使用值初始化它：

C++

```
// Example of the __declspec keyword
__declspec( thread ) int tls_i = 1;
```

结束 Microsoft 专用

另请参阅

关键字
C 扩展的存储类属性

Last updated on 2025/11/07

`align` (C++)

在 Visual Studio 2015 及更高版本中，使用 说明符`alignas` (C++ 11) 来控制对齐效果。 有关详细信息，请参阅对齐。

Microsoft 专用

使用 `__declspec(align(#))` 准确控制用户定义的数据的对齐方式（例如，函数中的静态分配或自动数据）。

语法

> **__declspec（align（#）） ）** *声明符*

注解

编写使用最新的处理器指令的应用程序会引入一些新的约束和问题。 许多新指令需要与 16 字节边界对齐的数据。 将常用数据与处理器的缓存行大小对齐，这可以提高缓存性能。 例如，如果你定义一个大小小于 32 字节的结构，则可能需要 32 字节对齐，以确保有效缓存该结构类型的对象。

是对齐值。 有效项为介于 1 和 8192（字节）之间的 2 的整数幂，如 2、4、8、16、32 或 64。 `declarator` 是你声明为对齐的数据。

有关如何返回该类型对齐要求的 `size_t` 类型的值的信息，请参阅 alignof。 有关如何在面向 64 位处理器时声明未对齐指针的信息，请参阅 __unaligned。

在定义 `__declspec(align(#))`、`struct` 或 `union` 时或声明变量时，可以使用 `class`。

编译器不保证或不尝试保留复制过程中或数据转换操作中数据的对齐特性。 例如，memcpy 可将使用 `__declspec(align(#))` 声明的结构复制到任何位置。 普通分配器（例如，malloc、C++ operator new 和 Win32 分配器）通常返回未对齐 `__declspec(align(#))` 结构或结构数组的内存。 若要保证复制或数据转换操作的目标正确对齐，请使用 _aligned_malloc。 你也可以编写自己的分配器。

不能指定函数参数的对齐方式。 当在堆栈上按值传递具有对齐属性的数据时，由调用约定控制其对齐方式。 如果数据对齐在所调用函数中很重要，请在使用前将参数复制到正确对齐的内存中。

如果没有 `__declspec(align(#))`，编译器将根据目标处理器和数据的大小将数据对齐到自然边界，在 32 位处理器上对齐到 4 字节边界，64 位处理器上则对齐到 8 字节边界。 类或结构中的

数据在类或结构中依据其自然对齐的最小值和当前包装设置（来自 `#pragma pack` 或 `/Zp` 编译器选项）进行对齐。

本示例演示 `__declspec(align(#))` 的使用：

C++

```
__declspec(align(32)) struct Str1{
   int a, b, c, d, e;
};
```

此类型现在具有 32 字节对齐特性。 这意味着所有静态和自动实例都在 32 字节边界上启动。 使用此类型声明为成员的其他结构类型保留此类型的对齐属性。 也就是说，任何以 `Str1` 作为元素的结构都具有至少 32 的对齐属性。

此处， `sizeof(struct Str1)` 等于 32。 这意味着，如果创建 `Str1` 对象的数组，并且该数组的基为对齐的 32 字节，则数组的每个成员也是对齐的 32 字节。 若要创建其基已在动态内存中正确对齐的数组，请使用 _aligned_malloc。 你也可以编写自己的分配器。

任何结构的 **`sizeof`** 值是最终成员的偏移量加上该成员的大小，向上舍入为最大成员对齐值的最接近倍数或整个结构的对齐值（以较大者为准）。

编译器使用以下结构对齐规则：

- 除非使用 `__declspec(align(#))` 进行重写，否则标量结构成员的对齐方式为其大小和当前包装的最小值。
- 除非使用 `__declspec(align(#))` 进行重写，否则结构的对齐方式是其成员单个对齐方式的最大值。
- 结构成员被放置在其父级结构的开头的偏移量位置，父级结构是大于或等于前一个成员末尾的偏移量的对齐的最小倍数。
- 结构的大小是大于或等于其最后一个成员末尾的偏移量的对齐的最小倍数。

`__declspec(align(#))` 只能增大对齐限制。

有关详细信息，请参阅：

- align 示例
- 使用 __declspec(align(#)) 定义新类型
- 在线程本地存储中对齐数据
- align 如何与数据打包一起工作
- x64 结构对齐示例

`align` 示例

以下示例说明 `__declspec(align(#))` 如何影响数据结构的大小和对齐方式。 这些示例假定下列定义：

C++

```
#define CACHE_LINE  32
#define CACHE_ALIGN __declspec(align(CACHE_LINE))
```

在此示例中，通过使用 `S1` 定义 `__declspec(align(32))` 结构。 针对一个变量定义或其他类型声明中的全部 `S1` 使用均为 32 字节对齐。 `sizeof(struct S1)` 返回 32，并且 `S1` 在包含四个整数所需的 16 字节之后有 16 个填充字节。 每个 **`int`** 成员要求为 4 字节对齐，但它自身结构的对齐声明为 32 字节。 那么，整体为 32 字节对齐。

C++

```
struct CACHE_ALIGN S1 { // cache align all instances of S1
   int a, b, c, d;
};
struct S1 s1;   // s1 is 32-byte cache aligned
```

在此示例中， `sizeof(struct S2)` 返回 16，它恰好为成员大小的总和，因为这是最大对齐要求的倍数（8 的倍数）。

C++

```
__declspec(align(8)) struct S2 {
   int a, b, c, d;
};
```

在下面的示例中， `sizeof(struct S3)` 返回 64。

C++

```
struct S3 {
   struct S1 s1;   // S3 inherits cache alignment requirement
                  // from S1 declaration
   int a;         // a is now cache aligned because of s1
                  // 28 bytes of trailing padding
};
```

在此示例中，注意 `a` 具有自然类型的对齐方式，在这种情况下为 4 字节。 但是， `S1` 必须是对齐的 32 字节。 填充的 28 个字节遵循 `a`，因此 `s1` 从偏移量 32 开始。 然后， `S4` 将继承 `S1` 的对齐要求，因为这是结构中的最大对齐要求。 `sizeof(struct S4)` 返回 64。

```C++
struct S4 {
   int a;
   // 28 bytes padding
   struct S1 s1;      // S4 inherits cache alignment requirement of S1
};
```

以下三个变量声明还使用 `__declspec(align(#))`。 在每种情况下，变量必须是对齐的 32 字节。 在数组中，数组的基址（而非每个数组成员）是对齐的 32 字节。 使用 **`sizeof`** 不会影响每个数组成员的 `__declspec(align(#))` 值。

```C++
CACHE_ALIGN int i;
CACHE_ALIGN int array[128];
CACHE_ALIGN struct s2 s;
```

若要对齐数组的每个成员，应使用如下代码：

```C++
typedef CACHE_ALIGN struct { int a; } S5;
S5 array[10];
```

在此示例中，注意对齐结构本身与对齐第一个元素具有相同的效果：

```C++
CACHE_ALIGN struct S6 {
   int a;
   int b;
};

struct S7 {
   CACHE_ALIGN int a;
               int b;
};
```

`S6` 和 `S7` 具有相同的对齐、分配和大小特性。

在此示例中， `a`、 `b`、 `c` 和 `d` 的起始地址的对齐方式分别为 4、1、4 和 1。

```C++
void fn() {
   int a;
   char b;
```

```
    long c;
    char d[10]
}
```

在堆上分配了内存时，对齐方式取决于所调用的分配函数。 例如，如果你使用 `malloc`，则该结果取决于操作数大小。 如果参数 = 8，返回的内存为 8 字节对齐>。 如果参数 8，所返回内存的对齐方式将为小于参数的 2 的一次幂<。 例如，如果你使用 `malloc(7)`，则对齐为 4 字节。

使用 `__declspec(align(#))` 定义新类型

可以使用对齐特性定义类型。

例如，你可以使用对齐值定义 `struct`，如下所示：

C++

```
struct aType {int a; int b;};
typedef __declspec(align(32)) struct aType bType;
```

现在， `aType` 和 `bType` 的大小相同（8 字节），但 `bType` 类型的变量将是对齐的 32 字节。

在线程本地存储中对齐数据

使用 `__declspec(thread)` 特性创建的置于映像中的 TLS 部分中的静态线程-本地存储 (TLS) 与常规静态数据一样适用于对齐。 若要创建 TLS 数据，操作系统需分配 TLS 部分大小的内存并遵循 TLS 部分对齐特性。

此示例说明用于将对齐的数据置于线程本地存储区中的各种方法。

C++

```
// put an aligned integer in TLS
__declspec(thread) __declspec(align(32)) int a;

// define an aligned structure and put a variable of the struct type
// into TLS
__declspec(thread) __declspec(align(32)) struct F1 { int a; int b; } a;

// create an aligned structure
struct CACHE_ALIGN S9 {
    int a;
    int b;
};
// put a variable of the structure type into TLS
__declspec(thread) struct S9 a;
```

`align` 如何与数据打包一起工作

`/Zp` 编译器选项和 `pack` pragma 对打包结构和联合成员的数据很有用。 此示例说明 `/Zp` 和 `__declspec(align(#))` 的协作方式：

C++

```
struct S {
   char a;
   short b;
   double c;
   CACHE_ALIGN double d;
   char e;
   double f;
};
```

下表列出了不同 `/Zp` （或 `#pragma pack`）值下的每个成员的偏移量，并演示了二者的交互方式。

展开表

变量	/Zp1	/Zp2	/Zp4	/Zp8
a	0	0	0	0
b	1	2	2	2
c	3	4	4	8
d	32	32	32	32
e	40	40	40	40
F	41	42	44	48
sizeof(S)	64	64	64	64

有关详细信息，请参阅 /Zp（结构成员对齐）。

对象的偏移量基于上一个对象的偏移量和当前打包设置，除非该对象具有 `__declspec(align(#))` 特性，在此情况下，对齐将基于上一个对象的偏移量和该对象的 `__declspec(align(#))` 值。

结束 Microsoft 专用

另请参阅

__declspec
ARM ABI 约定概述
x64 软件约定

Last updated on 2025/11/07

分配

Microsoft 专用

`allocate` 声明说明符命名将在其中分配数据项的数据段。

语法

> `__declspec(allocate("` *段名* `))` *声明符*

注解

必须使用下列 pragmas 之一来声明名称 segname：

- code_seg
- const_seg
- data_seg
- init_seg
- section

示例

C++

```
// allocate.cpp
#pragma section("mycode", read)
__declspec(allocate("mycode"))  int i = 0;

int main() {
}
```

结束 Microsoft 专用

另请参阅

__declspec
关键字

Last updated on 2025/11/07

`allocator`

Microsoft 专用

`allocator` 声明说明符可应用于自定义内存分配函数，以通过 Windows 事件跟踪 (ETW) 使分配可见。

语法

> `__declspec(allocator)`

备注

Visual Studio 中的本机内存探查器的工作原理是收集在运行时发出的分配 ETW 事件数据。 CRT 和 Windows SDK 中的分配器在源级别上注释，因此可以捕获其分配数据。 若要编写自己的分配器，对于返回的指针指向新分配的堆内存的任何函数，可使用 `__declspec(allocator)` 进行修饰，如以下 myMalloc 示例所示：

C++

```
__declspec(allocator) void* myMalloc(size_t size)
```

有关详细信息，请参阅度量 Visual Studio 中的内存使用量和自定义本机 ETW 堆事件。

结束 Microsoft 专用

Last updated on 2025/11/07

appdomain

指定托管应用程序的每个应用程序域应具有其自己的特定全局变量或静态成员变量的副本。 有关详细信息，请参阅应用程序域和 Visual C++。

每个应用程序域具有其自己的 per-appdomain 变量的副本。 在将程序集加载到应用程序域中时执行 appdomain 变量的构造函数，并在卸载应用程序域时执行析构函数。

如果希望公共语言运行时中进程内的所有应用程序域共享全局变量，请使用 `__declspec(process)` 修饰符。 `__declspec(process)` 默认情况下在 /clr 下有效。 “/clr:pure”和“/clr:safe”编译器选项在 Visual Studio 2015 中已弃用，并且在 Visual Studio 2017 中不受支持。

`__declspec(appdomain)` 仅在使用 **/clr** 编译器选项之一时有效。 只有全局变量、静态成员变量或静态局部变量可以使用 `__declspec(appdomain)` 进行标记。 将 `__declspec(appdomain)` 应用于托管类型的静态成员是错误的，因为它们始终具有此行为。

使用 `__declspec(appdomain)` 与使用线程本地存储 (TLS) 类似。 线程具有其自己的存储，就像应用程序域一样。 使用 `__declspec(appdomain)` 可确保全局变量在为此应用程序创建的每个应用程序域中都具有其自己的存储。

在将 per process 和 per appdomain 变量结合使用方面存在一些限制；有关详细信息，请参阅 process。

例如，在程序启动时，初始化所有 per-process 变量，然后初始化所有 per-appdomain 变量。 因此，当初始化 per-process 变量时，它不能依赖于任何 per-application 域变量的值。 混合使用（分配）per appdomain 和 per process 变量的做法不妥。

有关如何在特定应用程序域中调用函数的信息，请参阅 call_in_appdomain Function。

示例

C++

```
// declspec_appdomain.cpp
// compile with: /clr
#include <stdio.h>
using namespace System;

class CGlobal {
public:
   CGlobal(bool bProcess) {
      Counter = 10;
      m_bProcess = bProcess;
      Console::WriteLine("__declspec({0}) CGlobal::CGlobal constructor", m_bProcess
? (String^)"process" : (String^)"appdomain");
   }
```

```
   ~CGlobal() {
      Console::WriteLine("__declspec({0}) CGlobal::~CGlobal destructor", m_bProcess
? (String^)"process" : (String^)"appdomain");
   }

   int Counter;

private:
   bool m_bProcess;
};

__declspec(process) CGlobal process_global = CGlobal(true);
__declspec(appdomain) CGlobal appdomain_global = CGlobal(false);

value class Functions {
public:
   static void change() {
      ++appdomain_global.Counter;
   }

   static void display() {
      Console::WriteLine("process_global value in appdomain '{0}': {1}",
         AppDomain::CurrentDomain->FriendlyName,
         process_global.Counter);

      Console::WriteLine("appdomain_global value in appdomain '{0}': {1}",
         AppDomain::CurrentDomain->FriendlyName,
         appdomain_global.Counter);
   }
};

int main() {
   AppDomain^ defaultDomain = AppDomain::CurrentDomain;
   AppDomain^ domain = AppDomain::CreateDomain("Domain 1");
   AppDomain^ domain2 = AppDomain::CreateDomain("Domain 2");
   CrossAppDomainDelegate^ changeDelegate = gcnew
CrossAppDomainDelegate(&Functions::change);
   CrossAppDomainDelegate^ displayDelegate = gcnew
CrossAppDomainDelegate(&Functions::display);

   // Print the initial values of appdomain_global in all appdomains.
   Console::WriteLine("Initial value");
   defaultDomain->DoCallBack(displayDelegate);
   domain->DoCallBack(displayDelegate);
   domain2->DoCallBack(displayDelegate);

   // Changing the value of appdomain_global in the domain and domain2
   // appdomain_global value in "default" appdomain remain unchanged
   process_global.Counter = 20;
   domain->DoCallBack(changeDelegate);
   domain2->DoCallBack(changeDelegate);
   domain2->DoCallBack(changeDelegate);

   // Print values again
```

```
    Console::WriteLine("Changed value");
    defaultDomain->DoCallBack(displayDelegate);
    domain->DoCallBack(displayDelegate);
    domain2->DoCallBack(displayDelegate);

    AppDomain::Unload(domain);
    AppDomain::Unload(domain2);
}
```

Output

```
__declspec(process) CGlobal::CGlobal constructor
__declspec(appdomain) CGlobal::CGlobal constructor
Initial value
process_global value in appdomain 'declspec_appdomain.exe': 10
appdomain_global value in appdomain 'declspec_appdomain.exe': 10
__declspec(appdomain) CGlobal::CGlobal constructor
process_global value in appdomain 'Domain 1': 10
appdomain_global value in appdomain 'Domain 1': 10
__declspec(appdomain) CGlobal::CGlobal constructor
process_global value in appdomain 'Domain 2': 10
appdomain_global value in appdomain 'Domain 2': 10
Changed value
process_global value in appdomain 'declspec_appdomain.exe': 20
appdomain_global value in appdomain 'declspec_appdomain.exe': 10
process_global value in appdomain 'Domain 1': 20
appdomain_global value in appdomain 'Domain 1': 11
process_global value in appdomain 'Domain 2': 20
appdomain_global value in appdomain 'Domain 2': 12
__declspec(appdomain) CGlobal::~CGlobal destructor
__declspec(appdomain) CGlobal::~CGlobal destructor
__declspec(appdomain) CGlobal::~CGlobal destructor
__declspec(process) CGlobal::~CGlobal destructor
```

另请参阅

__declspec
关键字

Last updated on 2025/11/07

`__declspec(code_seg)`

Microsoft 专用

声明 `code_seg` 属性为文件中存储函数或类成员函数的对象代码的可执行文本段 `.obj` 命名。

语法

> `__declspec(code_seg("` *`segname`* `"))` *`declarator`*

注解

该 `__declspec(code_seg(...))` 属性允许将代码放置在可以单独分页或锁定在内存中的单独命名段中。 可以使用此属性来控制实例化模板和编译器生成的代码的位置。

*段*是文件中作为一个单元加载到内存中的`.obj`命名数据块。 *文本段*是包含可执行代码的段。 术语 *节* 通常与段互换使用。

定义时 `declarator` 生成的对象代码将放在指定的 `segname`文本段中，这是一个窄字符串文本。 `segname`名称不必在节杂注中指定，然后才能在声明中使用它。 默认情况下，如果未 指定，对象代码将放入名为 的段中。 属性 `code_seg` 将替代任何现有的 #pragma code_seg 指令。 `code_seg`应用于成员函数的属性将替代应用于封闭类的任何`code_seg`属性。

如果实体具有属性，则同一 `code_seg` 实体的所有声明和定义都必须具有相同 `code_seg` 的属性。 如果基类具有属性 `code_seg` ，则派生类必须具有相同的属性。

`code_seg`当属性应用于命名空间范围函数或成员函数时，该函数的对象代码将放入指定的文本段中。 当此属性应用于类时，类和嵌套类的所有成员函数（包括编译器生成的特殊成员函数）都放在指定的段中。 本地定义的类（例如成员函数正文中定义的类）不会继承 `code_seg` 封闭范围的属性。

`code_seg`当属性应用于类模板或函数模板时，模板的所有隐式专用化都放在指定的段中。 显式或部分专用化不会从主模板继承 `code_seg` 属性。 可以在专用化上指定相同或不同的 `code_seg` 属性。 `code_seg`无法将属性应用于显式模板实例化。

默认情况下，编译器生成的代码（如特殊成员函数）将放入 `.text` 段中。 该 `#pragma code_seg` 指令不会重写此默认值。 `code_seg`使用类、类模板或函数模板上的属性来控制编译器生成的代码的放置位置。

Lambda 从其封闭范围继承 `code_seg` 属性。 若要指定 lambda 的段，请在参数声明子句之后以及任何可变或异常规范、任何尾随返回类型规范和 lambda 正文之前应用 `code_seg` 属性。 有关

详细信息，请参阅 Lambda 表达式语法。 此示例在名为 PagedMem 的段中定义 lambda：

C++

```
auto Sqr = [](int t) __declspec(code_seg("PagedMem")) -> int { return t*t; };
```

在将特定成员函数（尤其是虚拟成员函数）置于不同的段中时，请小心。 假设在派生类中定义一个虚拟函数，当基类方法驻留在非分页段时，该函数驻留在分页段中。 其他基类方法或用户代码可能假定调用虚拟方法不会触发页面错误。

示例：

此示例演示如何 `code_seg` 在使用隐式和显式模板专用化时控制属性控制段放置：

C++

```
// code_seg.cpp
// Compile: cl /EHsc /W4 code_seg.cpp

// Base template places object code in Segment_1 segment
template<class T>
class __declspec(code_seg("Segment_1")) Example
{
public:
   virtual void VirtualMemberFunction(T /*arg*/) {}
};

// bool specialization places code in default .text segment
template<>
class Example<bool>
{
public:
   virtual void VirtualMemberFunction(bool /*arg*/) {}
};

// int specialization places code in Segment_2 segment
template<>
class __declspec(code_seg("Segment_2")) Example<int>
{
public:
   virtual void VirtualMemberFunction(int /*arg*/) {}
};

// Compiler warns and ignores __declspec(code_seg("Segment_3"))
// in this explicit specialization
__declspec(code_seg("Segment_3")) Example<short>; // C4071

int main()
{
   // implicit double specialization uses base template's
```

```
    // __declspec(code_seg("Segment_1")) to place object code
    Example<double> doubleExample{};
    doubleExample.VirtualMemberFunction(3.14L);

    // bool specialization places object code in default .text segment
    Example<bool> boolExample{};
    boolExample.VirtualMemberFunction(true);

    // int specialization uses __declspec(code_seg("Segment_2"))
    // to place object code
    Example<int> intExample{};
    intExample.VirtualMemberFunction(42);
}
```

结束 Microsoft 专用

另请参阅

__declspec
关键字

Last updated on 2025/11/07

已弃用 (C++)

本主题介绍 Microsoft 专用的 deprecated declspec 声明。 有关 C++14 `[[deprecated]]` 特性的信息，以及有关何时使用该特性与 Microsoft 专用的 declspec 或 pragma 的指导，请参阅 C++ 标准特性。

在下面所示的异常中，`deprecated` 声明提供了与 deprecated pragma 相同的功能：

- 利用 `deprecated` 声明，可以将函数重载的特殊形式指定为已弃用，而 pragma 形式适用于函数名称的所有重载形式。
- 利用 `deprecated` 声明，可以指定在编译时显示的消息。 该消息的文本可以来自宏。
- 只能使用 `deprecated` pragma 将宏标记为已弃用。

如果编译器遇到使用已弃用标识符或标准 [[deprecated]] 特性的情况，则会引发 C4996 警告。

示例

下面的示例演示在使用已弃用的函数时，如何将函数标记为已弃用以及如何指定在编译时将显示的消息。

C++

```
// deprecated.cpp
// compile with: /W3
#define MY_TEXT "function is deprecated"
void func1(void) {}
__declspec(deprecated) void func1(int) {}
__declspec(deprecated("** this is a deprecated function **")) void func2(int) {}
__declspec(deprecated(MY_TEXT)) void func3(int) {}

int main() {
   func1();
   func1(1);   // C4996
   func2(1);   // C4996
   func3(1);   // C4996
}
```

下面的示例演示在使用已弃用的类时，如何将类标记为已弃用以及如何指定在编译时将显示的消息。

C++

```
// deprecate_class.cpp
// compile with: /W3
struct __declspec(deprecated) X {
```

```
   void f(){}
};

struct __declspec(deprecated("** X2 is deprecated **")) X2 {
   void f(){}
};

int main() {
   X x;   // C4996
   X2 x2;   // C4996
}
```

另请参阅

__declspec
关键字

Last updated on 2025/11/07

`dllexport`、`dllimport`

Microsoft **专用**

`dllexport` `dllimport`存储类属性是 C 和 C++ 语言Microsoft特定的扩展。 可以使用它们来导出和导入 DLL 中的函数、数据和对象。

语法

> `__declspec( dllimport )` *`declarator`*
>
> `__declspec( dllexport )` *`declarator`*

注解

这些属性显式定义 DLL 的接口到其客户端，可以是可执行文件或其他 DLL。 将函数声明为 `dllexport` 不需要模块定义 （`.def`） 文件，至少与导出函数的规范有关。 该 `dllexport` 属性替换关键字 `__export` 。

如果标记 `__declspec(dllexport)`类，则类层次结构中类模板的任何专用化都隐式标记为 `__declspec(dllexport)`。 这意味着显式实例化类模板，并且必须定义类的成员。

`dllexport` 函数的 公开其修饰名称（有时称为“名称混乱”）的函数。 对于C++函数，修饰的名称包括编码类型和参数信息的额外字符。 声明为 `extern "C"` C 名称修饰规则的 C 函数或函数。 有关 C/C++ 代码中的名称修饰的详细信息，请参阅 修饰名称。

若要导出未编码的名称，可以使用定义节中`EXPORTS`未编码名称的模块定义 （`.def`） 文件进行链接。 有关详细信息，请参阅 EXPORTS。 导出未编码名称的另一 `#pragma comment(linker, "/export:alias=decorated_name")` 种方法是在源代码中使用指令。

声明或`dllimport`声明`dllexport`时，必须使用扩展属性语法和`__declspec`关键字。

示例：

C++

```cpp
// Example of the dllimport and dllexport class attributes
__declspec( dllimport ) int i;
__declspec( dllexport ) void func();
```

或者，若要使代码更易于阅读，可以使用宏定义：

C++

```
#define DllImport   __declspec( dllimport )
#define DllExport   __declspec( dllexport )

DllExport void func();
DllExport int i = 10;
DllImport int j;
DllExport int n;
```

有关详细信息，请参见:

- 定义和声明
- 定义内联C++函数以及dllexportdllimport
- 常规规则和限制
- 在C++类中使用 dllimport 和 dllexport 使用

结束 Microsoft 专用

另请参阅

__declspec
关键字

Last updated on 2025/11/07

定义和声明 (C++)

Microsoft 专用

DLL 接口引用已知由系统中的某程序导出的所有项（函数和数据）；即所有被声明为 `dllimport` 或 `dllexport` 的项。 DLL 接口中包含的所有声明都必须指定 `dllimport` 或 `dllexport` 特性。 但是，该定义必须仅指定 `dllexport` 特性。 例如，以下函数定义产生了一个编译器错误：

```
__declspec( dllimport ) int func() {   // Error; dllimport
                                       // prohibited on definition.
   return 1;
}
```

以下代码也会产生错误：

```
__declspec( dllimport ) int i = 10;  // Error; this is a definition.
```

但是，这是正确的语法：

```
__declspec( dllexport ) int i = 10;  // Okay--export definition
```

使用 `dllexport` 意味着定义，而使用 `dllimport` 则意味着声明。 必须使用带 `extern` 的 `dllexport` 关键字来强制进行声明；否则，会进行隐式定义。 因此，以下示例是正确的：

```
#define DllImport   __declspec( dllimport )
#define DllExport   __declspec( dllexport )

extern DllExport int k; // These are both correct and imply a
DllImport int j;        // declaration.
```

以下示例阐明了前面的示例：

```
static __declspec( dllimport ) int l; // Error; not declared extern.

void func() {
    static __declspec( dllimport ) int s;  // Error; not declared
                                           // extern.
```

```
    __declspec( dllimport ) int m;         // Okay; this is a
                                           // declaration.
    __declspec( dllexport ) int n;         // Error; implies external
                                           // definition in local scope.
    extern __declspec( dllimport ) int i;  // Okay; this is a
                                           // declaration.
    extern __declspec( dllexport ) int k;  // Okay; extern implies
                                           // declaration.
    __declspec( dllexport ) int x = 5;     // Error; implies external
                                           // definition in local scope.
}
```

结束 Microsoft 专用

另请参阅

dllexport、dllimport

Last updated on 2025/11/07

使用 dllexport 和 dllimport 定义内联 C++ 函数

Microsoft 专用

可以定义为将函数与 `dllexport` 特性内联。 在这种情况下，将始终实例化并导出该函数，无论程序中是否有模块引用该函数。 假定该函数由另一个程序导入。

还可以定义为内联使用 `dllimport` 特性声明的函数。 在这种情况下，该函数可以展开（遵从 /Ob 规范），但决不实例化。 具体而言，如果采用内联导入函数的地址，则返回驻留在 DLL 中的函数地址。 此行为与采用非内联导入函数的地址相同。

这些规则适用于其定义出现在类定义中的内联函数。 此外，内联函数中的静态本地数据和字符串在 DLL 和客户端之间保持的标识与它们在单一程序（即，没有 DLL 接口的可执行文件）中保持的一样。

在提供导入的内联函数时谨慎操作。 例如，如果更新 DLL，请不要假定该客户端将使用更改后的 DLL 版本。 若要确保加载 DLL 的适当版本，请重新生成 DLL 的客户端。

结束 Microsoft 专用

另请参阅

dllexport、dllimport

Last updated on 2025/11/07

一般规则和限制

Microsoft 专用

- 如果在不使用 `dllimport` 或 `dllexport` 特性的情况下声明函数或对象，则该函数或对象不会被视为 DLL 接口的一部分。 因此，函数或对象的定义必须存在于该模块或同一程序的另一个模块中。 若要使函数或对象成为 DLL 接口的一部分，必须将其他模块中函数或对象的定义声明为 `dllexport`。 否则，将生成链接器错误。

  如果使用 `dllexport` 特性声明函数或对象，则其定义必须出现在同一程序的某个模块中。 否则，将生成链接器错误。

- 如果程序中的单个模块包含对同一函数或对象的 `dllimport` 和 `dllexport` 声明，则 `dllexport` 特性优先于 `dllimport` 特性。 但是，会生成编译器警告。 例如：

  C++

  ```
  __declspec( dllimport ) int i;
  __declspec( dllexport ) int i;   // Warning; inconsistent;
                                   // dllexport takes precedence.
  ```

- 在 C++ 中，可初始化全局声明的局部数据指针或静态局部数据指针，或利用使用 `dllimport` 特性声明的数据对象的地址进行初始化，后一种初始化方法在 C 中会生成错误。此外，还可利用使用 `dllimport` 特性声明的函数的地址初始化静态局部函数指针。 在 C 中，此类赋值会将指针设置为指向 DLL 导入形式转换 (thunk)（将控制权转交给函数的代码存根）的地址而不是函数的地址。 在 C++ 中，此类赋值会将指针设置为指向函数的地址。 例如：

  C++

  ```
  __declspec( dllimport ) void func1( void );
  __declspec( dllimport ) int i;

  int *pi = &i;                              // Error in C
  static void ( *pf )( void ) = &func1;      // Address of thunk in C,
                                             // function in C++

  void func2()
  {
     static int *pi = &i;                   // Error in C
     static void ( *pf )( void ) = &func1;  // Address of thunk in C,
                                            // function in C++
  }
  ```

  但是，由于包含对象声明中的 `dllexport` 特性的程序必须在程序中的某个位置为对象提供定义，因此可以利用 `dllexport` 函数的地址初始化全局或局部静态函数指针。 同样，您可

以利用 `dllexport` 数据对象的地址初始化全局或局部静态数据指针。 例如，以下代码在 C 或 C++ 中不会生成错误：

C++

```
__declspec( dllexport ) void func1( void );
__declspec( dllexport ) int i;

int *pi = &i;                              // Okay
static void ( *pf )( void ) = &func1;      // Okay

void func2()
{
    static int *pi = &i;                   // Okay
    static void ( *pf )( void ) = &func1;  // Okay
}
```

- 如果将 `dllexport` 应用于具有未标记为 `dllexport` 的基类的常规类，编译器将生成 C4275。

  如果基类是类模板的专用化，则编译器将生成相同的警告。 若要解决此问题，请将基类标记为 `dllexport`。 类模板专用化的问题在于在何处放置 `__declspec(dllexport)`；不允许标记类模板。 相反，应显式实例化类模板并将此显式实例化标记为 `dllexport`。 例如：

  C++

  ```
  template class __declspec(dllexport) B<int>;
  class __declspec(dllexport) D : public B<int> {
  // ...
  ```

  如果模板自变量是派生类，则此解决方法将失败。 例如：

  C++

  ```
  class __declspec(dllexport) D : public B<D> {
  // ...
  ```

  由于这是模板的常见模式，因此当 `dllexport` 应用于具有一个或多个基类的类时，以及一个或多个基类是类模板的专用化时，编译器更改了该特性的语义。 在这种情况下，编译器会将 `dllexport` 隐式应用于类模板的专用化。 可执行以下代码而不会收到警告：

  C++

  ```
  class __declspec(dllexport) D : public B<D> {
  // ...
  ```

结束 Microsoft 专用

另请参阅

dllexport、dllimport

Last updated on 2025/11/07

在 C++ 类中使用 dllimport 和 dllexport

Microsoft 专用

可以使用 `dllimport` 或 `dllexport` 特性来声明 C++ 类。 这些形式表示已导入或导出整个类。 以这种方式导出的类称为可导出类。

以下示例定义了可导出类。 将导出其所有成员函数和静态数据：

C++

```
#define DllExport   __declspec( dllexport )

class DllExport C {
   int i;
   virtual int func( void ) { return 1; }
};
```

请注意，禁止对可导出类的成员显式使用 `dllimport` 和 `dllexport` 特性。

dllexport 类

当你声明 `dllexport` 类时，它的所有成员函数和静态数据成员都会导出。 您必须在同一程序中提供所有此类成员的定义。 否则，将生成链接器错误。 此规则有一个例外情况，即对于纯虚函数，您无需为其提供显式定义。 但是，由于基类的析构函数始终在调用抽象类的析构函数，因此纯虚拟析构函数必须始终提供定义。 请注意，这些规则对不可导出的类是相同的。

如果导出类类型的数据或返回类的函数，请务必导出类。

dllimport 类

当你声明 `dllimport` 类时，它的所有成员函数和静态数据成员都会导入。 与非类类型上的 `dllimport` 和 `dllexport` 的行为不同，静态数据成员无法在定义 `dllimport` 类的同一程序中指定定义。

继承和可导出类

可导出类的所有基类都必须是可导出的。 否则，会生成编译器警告。 此外，同样是类的所有可访问成员必须是可导出的。 此规则只允许 `dllexport` 类从 `dllimport` 类继承， `dllimport` 类从 `dllexport` 类继承（但不建议后者）。 通常来说，对 DLL 客户端可访问的所有内容（根据 C++ 访问规则）都应该是可导出接口的一部分。 这包括在内联函数中引用的私有数据成员。

选择性成员导入/导出

由于类中的成员函数和静态数据隐式包含了外部链接，因此可以使用 `dllimport` 或 `dllexport` 特性声明它们，除非整个类都已导出。 如果整个类都已导入或导出，则禁止将成员函数和数据显式声明为 `dllimport` 或 `dllexport`。 如果在类定义中将静态数据成员声明为 `dllexport`，定义一定会在同一程序中的某处出现（就像非类外部链接一样）。

同样，可以使用 `dllimport` 或 `dllexport` 特性声明成员函数。 在这种情况下，必须在同一程序中的某处提供 `dllexport` 定义。

有关选择性成员导入和导出的某些要点值得注意：

- 选择性成员导入/导出最适合用于提供具有更强限制的导出类接口版本；即，您可以为该版本设计一个 DLL，该 DLL 公开的公用和专用功能比本应允许的语言公开的更少。 这对于优化可导出接口也很有用：当通过定义知道客户端无法访问某些私有数据时，您不需要导出整个类。
- 如果导出了某个类中的一个虚函数，则必须导出其中的所有虚函数，或者至少必须提供客户端可直接使用的版本。
- 如果有在其中将选择性成员导入/导出用于虚函数的类，则这些函数必须在可导出接口或已定义内联中（对客户端可见）。
- 如果将某个成员定义为 `dllexport`，但不在类定义中包含它，则会产生编译器错误。 必须在类头中定义成员。
- 尽管允许将类成员定义为 `dllimport` 或 `dllexport`，但无法重写在类定义中指定的接口。
- 如果在声明成员函数的类定义的主体以外的地方定义成员函数，并且将函数定义为 `dllexport` 或 `dllimport`（如果此定义不同于类声明中指定的定义），则会生成警告。

结束 Microsoft 专用

另请参阅

dllexport、dllimport

Last updated on 2025/11/07

`empty_bases`

Microsoft 专用

C++ Standard 要求大多数情况下为派生的对象的大小不得为零，并且该对象必须占用一个或多个存储字节。 由于此要求仅扩展到大多数情况下为派生的对象，因此基类子对象不受此约束的约束。 空基类优化 (EBCO) 充分利用了这种自由。 这会导致内存消耗减少，因此可提高性能。 Microsoft Visual C++ 编译器在历史上对 EBCO 的支持有限。 在 Visual Studio 2015 Update 3 及更高版本中，我们添加了一个新的 `__declspec(empty_bases)` 属性，用于充分利用此优化的类类型。

> ⓘ **重要**
>
> 使用 `__declspec(empty_bases)` 可能会导致应用它的结构和类布局中发生 ABI 中断性变更。 使用此存储类属性时，请确保所有客户端代码对结构和类使用与你的代码相同的定义。

语法

```
__declspec( empty_bases )
```

备注

在缺少任何 `__declspec(align())` 或 `alignas()` 规范的 Visual Studio 中，空类的大小为 1 字节：

C++

```
struct Empty1 {};
static_assert(sizeof(Empty1) == 1, "Empty1 should be 1 byte");
```

包含类型为 `char` 的单个非静态数据成员的类的大小也为 1 字节：

C++

```
struct Struct1
{
  char c;
};
static_assert(sizeof(Struct1) == 1, "Struct1 should be 1 byte");
```

在类层次结构中合并这些类也会生成大小为 1 字节的类：

C++

```
struct Derived1 : Empty1
{
  char c;
};
static_assert(sizeof(Derived1) == 1, "Derived1 should be 1 byte");
```

此结果是空基类优化生效导致的，因为如果没有它，`Derived1` 的大小就会是 2 字节：1 字节用于 `Empty1`，1 字节用于 `Derived1::c`。 当存在一系列空类时，类布局也是最佳的：

C++

```
struct Empty2 : Empty1 {};
struct Derived2 : Empty2
{
  char c;
};
static_assert(sizeof(Derived2) == 1, "Derived2 should be 1 byte");
```

但是，Visual Studio 中的默认类布局不会在多个继承方案中利用 EBCO：

C++

```
struct Empty3 {};
struct Derived3 : Empty2, Empty3
{
  char c;
};
static_assert(sizeof(Derived3) == 1, "Derived3 should be 1 byte"); // Error
```

尽管 `Derived3` 的大小可以是 1 字节，但默认类布局会导致它的大小为 2 字节。 类布局算法是在任意两个连续的空基类之间添加 1 字节的填充内容，实际上导致 `Empty2` 在 `Derived3` 中消耗一个额外的字节：

```
class Derived3  size(2):
   +---
0  | +--- (base class Empty2)
0  | | +--- (base class Empty1)
   | | +---
   | +---
1  | +--- (base class Empty3)
   | +---
1  | c
   +---
```

当后续的基类或成员子对象的对齐要求强制进行额外填充时，此次优布局的效果会复杂化：

C++

```
struct Derived4 : Empty2, Empty3
{
  int i;
};
static_assert(sizeof(Derived4) == 4, "Derived4 should be 4 bytes"); // Error
```

类型为 `int` 的对象的自然对齐方式为 4 字节，因此必须在 `Empty3` 后添加 3 字节的额外填充内容才能将 `Derived4::i` 正确对齐：

```
class Derived4 size(8):
   +---
0  | +--- (base class Empty2)
0  | | +--- (base class Empty1)
   | | +---
   | +---
1  | +--- (base class Empty3)
   | +---
   | <alignment member> (size=3)
4  | i
   +---
```

默认类布局的另一个问题是，空基类布局时存在的偏移量可能超出类的末尾：

C++

```
struct Struct2 : Struct1, Empty1
{
};
static_assert(sizeof(Struct2) == 1, "Struct2 should be 1 byte");
```

```
class Struct2 size(1):
   +---
0  | +--- (base class Struct1)
0  | | c
   | +---
1  | +--- (base class Empty1)
   | +---
   +---
```

虽然 `Struct2` 是最佳大小， `Empty1` 布局时在 `Struct2` 中的偏移量为 1，但 `Struct2` 的大小的增加并未考虑到此因素。 因此，对于 `A` 对象的数组 `Struct2`， `Empty1` 的 `A[0]` 子对象的地址将与

`A[1]` 的地址相同，这种情况是不应该出现的。 如果在 `Empty1` 中偏移量为 0 处对 `Struct2` 进行布局，导致 `Struct1` 子对象重叠，则不会发生此问题。

未修改默认布局算法来解决这些限制并充分利用 EBCO。 这种更改会破坏二进制兼容性。 如果类的默认布局因 EBCO 而更改，则需要重新编译包含类定义的每个对象文件和库，使它们都同意类布局。 此要求还将扩展到从外部源获取的库。 此类库的开发人员将必须同时提供使用和不使用 EBCO 布局编译的独立版本，以支持使用不同版本的编译器的客户。 我们无法更改默认布局，但可以通过添加 `__declspec(empty_bases)` 类属性来提供一种按类更改布局的方法。 使用此属性定义的类可以充分利用 EBCO。

C++

```
struct __declspec(empty_bases) Derived3 : Empty2, Empty3
{
  char c;
};
static_assert(sizeof(Derived3) == 1, "Derived3 should be 1 byte"); // No Error
```

```
class Derived3  size(1):
   +---
0  | +--- (base class Empty2)
0  | | +--- (base class Empty1)
   | | +---
   | +---
0  | +--- (base class Empty3)
   | +---
0  | c
   +---
```

`Derived3` 的所有子对象布局时的偏移量都为 0，其大小是最佳的 1 字节。 需要记住的一个要点是， `__declspec(empty_bases)` 只影响它所应用到的类的布局。 它不会以递归方式应用到基类：

C++

```
struct __declspec(empty_bases) Derived5 : Derived4
{
};
static_assert(sizeof(Derived5) == 4, "Derived5 should be 4 bytes"); // Error
```

```
class Derived5  size(8):
   +---
0  | +--- (base class Derived4)
0  | | +--- (base class Empty2)
0  | | | +--- (base class Empty1)
```

```
   | | | +---
   | | +---
1  | | +--- (base class Empty3)
   | | +---
   | | <alignment member> (size=3)
4  | | i
   | +---
   +---
```

虽然 `__declspec(empty_bases)` 应用到 `Derived5`，但它不符合 EBCO 的条件，因为它没有任何直接的空基类，因此不起作用。 但是，如果改将其应用于符合 EBCO 条件的 `Derived4` 基类，则 `Derived4` 和 `Derived5` 都将具有最佳布局：

C++

```cpp
struct __declspec(empty_bases) Derived4 : Empty2, Empty3
{
  int i;
};
static_assert(sizeof(Derived4) == 4, "Derived4 should be 4 bytes"); // No Error

struct Derived5 : Derived4
{
};
static_assert(sizeof(Derived5) == 4, "Derived5 should be 4 bytes"); // No Error
```

```
class Derived5  size(4):
   +---
0  | +--- (base class Derived4)
0  | | +--- (base class Empty2)
0  | | | +--- (base class Empty1)
   | | | +---
   | | +---
0  | | +--- (base class Empty3)
   | | +---
0  | | i
   | +---
   +---
```

由于要求所有对象文件和库都同意类布局，因此 `__declspec(empty_bases)` 只能应用于你所控制的类。 它不能应用于标准库中的类，也不能应用于也未使用 EBCO 布局重新编译的库中包含的类。

结束 Microsoft 专用

另请参阅

__declspec
关键字

Last updated on 2025/11/07

`hybrid_patchable` (C++)

Microsoft 专用

用于 `__declspec(hybrid_patchable)` 将函数标记为混合可修补函数。 此属性生成快速转发序列。 快进序列是不包含实际逻辑的小型 x64 函数，并尾调用真正的 Arm64EC 函数。 由于快进序列主要用于挂钩，因此如果它们未更改，则执行将直接传输到 Arm64EC 函数。

语法

> `__declspec(hybrid_patchable)`

注解

`__declspec(hybrid_patchable)` 是仅ARM64EC功能。

示例：

C++

```
__declspec(hybrid_patchable) int Example()
{
    return 1;
}
```

生成以下快进序列：

```
EXP+#Example:
  00000001400CE000: 48 8B C4          mov         rax,rsp
  00000001400CE003: 48 89 58 20       mov         qword ptr [rax+20h],rbx
  00000001400CE007: 55                push        rbp
  00000001400CE008: 5D                pop         rbp
  00000001400CE009: E9 BA 7A F3 FF    jmp         #Example
  00000001400CE00E: CC                int         3
  00000001400CE00F: CC                int         3
```

结束 Microsoft 专用

另请参阅

__declspec
Fast-Forward 序列

Last updated on 2025/11/07

jitintrinsic

将函数标记为对 64 位公共语言运行时很有用。 这用于 Microsoft 提供的库中的某些函数。

语法

```
__declspec(jitintrinsic)
```

备注

`jitintrinsic` 将 MODOPT (IsJitIntrinsic) 添加到函数签名。

不建议用户使用该 `__declspec` 修饰符，因为可能出现意外结果。

另请参阅

__declspec
关键字

Last updated on 2025/11/07

裸体 （C++）

Microsoft 专用

对于使用 `naked` 特性声明的函数，编译器生成不带 prolog 和 epilog 代码的代码。 利用此功能，可以使用内联汇编程序代码编写您自己的 prolog/epilog 代码序列。 裸函数对于编写虚拟设备驱动程序特别有用。 请注意，`naked` 特性仅在 x86 和 ARM 上有效，且不可用于 x64 上。

语法

```
__declspec(naked) declarator
```

备注

由于 `naked` 特性仅与函数定义相关且不是类型修饰符，因此 naked 函数使用扩展的特性语法和 __declspec 关键词。

即使函数也标有 __forceinline 关键字，编译器也不能为用裸属性标记的函数生成内联函数。

如果 `naked` 属性应用于非成员方法的定义以外的任何内容，则编译器会发出错误。

示例

此代码使用 `naked` 属性定义函数：

```
__declspec( naked ) int func( formal_parameters ) {}
```

或者：

```
#define Naked __declspec( naked )
Naked int func( formal_parameters ) {}
```

`naked` 特性仅影响函数的 prolog 和 epilog 序列的编译器代码生成的性质。 它不影响为调用这些函数而生成的代码。 因此，`naked` 特性不被视为函数的类型的一部分，并且函数指针不能具有 `naked` 特性。 此外，`naked` 特性不能应用于数据定义。 例如，此代码示例生成错误：

```
__declspec( naked ) int i;
// Error--naked attribute not permitted on data declarations.
```

`naked` 特性仅与函数的定义相关，且无法在函数原型中指定。 例如，此声明生成编译器错误：

```
__declspec( naked ) int func();  // Error--naked attribute not permitted on function
declarations
```

结束 Microsoft 专用

另请参阅

__declspec
关键字
Naked 函数调用

Last updated on 2025/11/07

`noalias`

Microsoft 专用

`noalias` 意味着函数调用不会修改或引用可见全局状态且只会修改指针参数（第一级间接寻址）直接指向的内存。

如果函数被批注为 `noalias`，则优化器可假定只在函数内引用或修改参数本身和指针参数的第一级间接寻址。

`noalias` 批注仅适用于已批注的函数的正文。 将函数标记为 `__declspec(noalias)` 不会影响该函数返回的指针的别名。

如需可能影响别名的其他批注，请参阅 __declspec(restrict)。

示例

以下示例演示了 `__declspec(noalias)` 的用法。

访问内存的 `multiply` 函数用 `__declspec(noalias)` 批注时，会通知编译器：此函数不会修改全局状态，除非通过其参数列表中的指针。

C

```c
// declspec_noalias.c
#include <stdio.h>
#include <stdlib.h>

#define M 800
#define N 600
#define P 700

float * mempool, * memptr;

float * ma(int size)
{
    float * retval;
    retval = memptr;
    memptr += size;
    return retval;
}

float * init(int m, int n)
{
    float * a;
    int i, j;
    int k=1;

    a = ma(m * n);
```

```
    if (!a) exit(1);
    for (i=0; i<m; i++)
        for (j=0; j<n; j++)
            a[i*n+j] = 0.1/k++;
    return a;
}

__declspec(noalias) void multiply(float * a, float * b, float * c)
{
    int i, j, k;

    for (j=0; j<P; j++)
        for (i=0; i<M; i++)
            for (k=0; k<N; k++)
                c[i * P + j] =
                          a[i * N + k] *
                          b[k * P + j];
}

int main()
{
    float * a, * b, * c;

    mempool = (float *) malloc(sizeof(float) * (M*N + N*P + M*P));

    if (!mempool)
    {
        puts("ERROR: Malloc returned null");
        exit(1);
    }

    memptr = mempool;
    a = init(M, N);
    b = init(N, P);
    c = init(M, P);

    multiply(a, b, c);
}
```

另请参阅

__declspec
关键字
__declspec(restrict)

Last updated on 2025/11/07

noinline

Microsoft 专用

`__declspec(noinline)` 告知编译器永远不要内联特定的成员函数（类中的函数）。

如果某个函数很小，并且对代码性能的影响不大，则不值得内联它。 即，如果函数很小并且不太可能经常调用（如处理错误条件的函数）。

请记住，如果某个函数标记为 `noinline`，则调用函数更小，因此它本身就是编译器内联的候选项。

C++

```
class X {
   __declspec(noinline) int mbrfunc() {
      return 0;
   }   // will not inline
};
```

结束 Microsoft 专用

另请参阅

__declspec
关键字
inline、__inline、__forceinline

Last updated on 2025/11/07

noreturn

Microsoft 专用

`__declspec` 属性告知编译器函数不会返回。 然后，编译器就会知晓调用 `__declspec(noreturn)` 函数后的代码是不可访问的。

如果编译器找到带有不返回值的控制路径的函数，则它会生成警告 (C4715) 或错误消息 (C2202)。 如果控制路径由于永不返回的函数而无法访问，则请使用 `__declspec(noreturn)` 阻止警告或错误。

> **注意**
>
> 将 `__declspec(noreturn)` 添加到到预期返回的函数可能会导致未定义的行为。

示例

在以下示例中，当 `isZeroOrPositive` 的参数为负数时，将调用 `fatal`。 该控制路径中没有 return 语句，这会生成警告 C4715，指示并非所有控制路径都返回值。 将 `fatal` 声明为 `__declspec(noreturn)` 可以缓解该警告，这是可取的，因为它没有任何意义（因为 `fatal()` 会终止程序）。

C++

```
// noreturn2.cpp
#include <exception>

__declspec(noreturn) void fatal()
{
   std::terminate();
}

int isZeroOrPositive(int val)
{
   if (val == 0)
   {
      return 0;
   }
   else if (val > 0)
   {
      return 1;
   }
   // this function terminates if val is negative
   fatal();
}
```

```
int main()
{
   isZeroOrPositive(123);
}
```

结束 Microsoft 专用

另请参阅

__declspec
关键字

Last updated on 2025/11/07

`no_sanitize_address`

Microsoft 专用

`__declspec(no_sanitize_address)` 说明符告知编译器针对函数、局部变量或全局变量上禁用地址净化器。 此说明符与 AddressSanitizer 结合使用。

> **注意**
>
> `__declspec(no_sanitize_address)` 禁用 compiler 行为，而不是 runtime 行为。

示例

有关示例，请参阅 AddressSanitizer 生成参考。

结束 Microsoft 专用

另请参阅

__declspec
关键字
AddressSanitizer

Last updated on 2025/11/07

`nothrow` (C++)

Microsoft 专用

`__declspec` 扩展特性，可在函数声明中使用。

语法

> *return-type* `__declspec(nothrow)` [*call-convention*] *function-name* ([*argument-list*])

备注

建议所有新代码都使用 noexcept 运算符而不是 `__declspec(nothrow)`。

此特性告知编译器，声明的函数及其调用的函数从不引发异常。 但是，它不强制实施该指令。 换句话说，它从不导致调用 std::terminate，不像 **`noexcept`**，或 **`/std:c++17`** 模式下（Visual Studio 2017 版本 15.5 及更高版本）的 `throw()`。

利用当前默认的同步异常处理模式，编译器可以消除跟踪此类函数中的某些不可展开的对象的生命周期的机制，从而显著减小代码大小。 在使用以下预处理器指令的情况下，下面的三个函数声明在 **`/std:c++14`** 模式下是等效的：

C++

```
#define WINAPI __declspec(nothrow) __stdcall

void WINAPI f1();
void __declspec(nothrow) __stdcall f2();
void __stdcall f3() throw();
```

在 **`/std:c++17`** 模式下， `throw()` 与使用 `__declspec(nothrow)` 的其他函数不等效，因为当从函数引发异常时，它将导致 `std::terminate` 调用。

`void __stdcall f3() throw();` 声明使用由 C++ 标准定义的语法。 在 C++17 中， `throw()` 关键字已弃用。

结束 Microsoft 专用

另请参阅

__declspec
noexcept

关键字

Last updated on 2025/11/07

novtable

Microsoft 专用

这是 `__declspec` 扩展属性。

此形式的 `__declspec` 可应用于任何类声明，但只应该应用于纯接口类，即永远不会自行实例化的类。 `__declspec` 可阻止编译器生成用来初始化类的构造函数和析构函数中的 vfptr 的代码。 在许多情况下，这将移除对与类关联的 vtable 的唯一引用，因此链接器将移除它。 使用此形式的 `__declspec` 可使代码大小显著显减小。

如果尝试实例化标记为 `novtable` 的类，然后访问类成员，将收到访问冲突 (AV)。

示例

C++

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

struct __declspec(novtable) X {
   virtual void mf();
};

struct Y : public X {
   void mf() {
      printf_s("In Y\n");
   }
};

int main() {
   // X *pX = new X();
   // pX->mf();   // Causes a runtime access violation.

   Y *pY = new Y();
   pY->mf();
}
```

Output

```
In Y
```

结束 Microsoft 专用

另请参阅

__declspec
关键字

Last updated on 2025/11/07

`process`

指定托管应用程序进程在该过程中应具有特定全局变量、静态成员变量或在任何应用程序域之间共享的静态局部变量的单一副本。 这主要是为了在用 `/clr:pure` 编译时使用，它在 Visual Studio 2015 中已弃用，在 Visual Studio 2017 中不受支持。 在使用 `/clr` 编译时，默认情况下，全局变量和静态变量是按进程进行的，并且不需要使用 `__declspec(process)`。

只有本机类型的全局变量、静态成员变量或静态局部变量可以使用 `__declspec(process)` 标记。

`process` 仅在使用 /clr 编译时有效。

如果希望每个应用程序域都有自己的全局变量副本，请使用 appdomain。

有关详细信息，请参阅应用程序域和 Visual C++。

另请参阅

__declspec
关键字

Last updated on 2025/11/07

属性 (C++)

Microsoft 专用

此特性可应用于类或结构定义中的非静态“虚拟数据成员”。 编译器通过将这些“虚拟数据成员”的引用更改为函数调用来将其作为数据成员处理。

语法

```
__declspec( property( get=get_func_name ) ) declarator
__declspec( property( put=put_func_name ) ) declarator
__declspec( property( get=get_func_name, put=put_func_name ) ) declarator
```

备注

当编译器在成员选择运算符（“.”或“-”）的右侧看到使用此特性声明的数据成员时，它会根据此类表达式是左值还是右值，将运算转换为 > 或 函数`get` `put`。 在更复杂的上下文中（如“`+=`”），可通过同时执行 `get` 和 `put` 来执行重写。

此特性还可用于类或结构定义中的空数组的声明。 例如：

C++

```
__declspec(property(get=GetX, put=PutX)) int x[];
```

上面的语句指示 `x[]` 可用于一个或多个数组索引。 在这种情况下，`i=p->x[a][b]` 将变为 `i=p->GetX(a, b)`，`p->x[a][b] = i` 将变为 `p->PutX(a, b, i);`

结束 Microsoft 专用

示例

C++

```
// declspec_property.cpp
struct S {
   int i;
   void putprop(int j) {
      i = j;
   }

   int getprop() {
```

```
        return i;
    }

    __declspec(property(get = getprop, put = putprop)) int the_prop;
};

int main() {
    S s;
    s.the_prop = 5;
    return s.the_prop;
}
```

另请参阅

__declspec
关键字

Last updated on 2025/11/07

restrict

Microsoft 专用

当应用于返回指针类型的函数声明或定义时，`restrict` 通知编译器该函数将返回没有*别名*（即被任何其他指针引用）的对象。 这允许编译器执行其他优化。

语法

> `__declspec(restrict)` *pointer_return_type函数*（）；

注解

编译器传播 `__declspec(restrict)`。 例如，CRT `malloc` 函数具有 `__declspec(restrict)` 修饰，因此，编译器假定由 `malloc` 初始化到内存位置的指针也不会被之前存在的指针指定别名。

编译器不检查指针到底有没有使用别名。 开发人员负责确保程序没有对使用 **restrict __declspec** 修饰符标记的指针使用别名。

有关变量的类似语义，请参阅 __restrict。

有关应用于函数内别名的另一个注释，请参阅 __declspec(noalias)。

有关作为 C++ AMP 的一部分的 `restrict` 关键字的信息，请参阅 restrict (C++ AMP)。

示例

以下示例演示了 `__declspec(restrict)` 的用法。

当 `__declspec(restrict)` 应用于返回指针的函数时，这会告知编译器返回值指向的内存没有别名。 在此示例中，指针 `mempool` 和 `memptr` 为全局指针，因此编译器无法确定所引用的内存没有别名。 但是，它们以返回程序未引用的内存的方式在 `ma` 及其调用方 `init` 内部使用，因此，**__decslpec(restrict)** 用于帮助优化器。 这类似于 CRT 标头如何修饰分配函数，例如 `malloc`，方法是使用 `__declspec(restrict)` 指示它们始终返回无法由现有指针别名的内存。

C

```c
// declspec_restrict.c
// Compile with: cl /W4 declspec_restrict.c
#include <stdio.h>
#include <stdlib.h>

#define M 800
```

```c
#define N 600
#define P 700

float * mempool, * memptr;

__declspec(restrict) float * ma(int size)
{
    float * retval;
    retval = memptr;
    memptr += size;
    return retval;
}

__declspec(restrict) float * init(int m, int n)
{
    float * a;
    int i, j;
    int k=1;

    a = ma(m * n);
    if (!a) exit(1);
    for (i=0; i<m; i++)
        for (j=0; j<n; j++)
            a[i*n+j] = 0.1f/k++;
    return a;
}

void multiply(float * a, float * b, float * c)
{
    int i, j, k;

    for (j=0; j<P; j++)
        for (i=0; i<M; i++)
            for (k=0; k<N; k++)
                c[i * P + j] =
                          a[i * N + k] *
                          b[k * P + j];
}

int main()
{
    float * a, * b, * c;

    mempool = (float *) malloc(sizeof(float) * (M*N + N*P + M*P));

    if (!mempool)
    {
        puts("ERROR: Malloc returned null");
        exit(1);
    }

    memptr = mempool;
    a = init(M, N);
    b = init(N, P);
    c = init(M, P);
```

```
    multiply(a, b, c);
}
```

结束 Microsoft 专用

另请参阅

关键字
__declspec
__declspec(noalias)

Last updated on 2025/11/07

safebuffers

Microsoft 专用

告知编译器不要插入针对函数的缓冲区溢出安全检查。

语法

```
__declspec( safebuffers )
```

备注

/GS 编译器选项将使编译器通过插入对堆栈的安全检查来测试缓冲区溢出。 符合安全检查条件的数据结构的类型在 /GS（缓冲区安全检查）中介绍。 有关缓冲区溢出检测的详细信息，请参阅 MSVC 中的安全功能。

专家手动代码评审或外部分析可能确定函数不会出现缓冲区溢出。 在这种情况下，可以通过对声明函数应用 `__declspec(safebuffers)` 关键字来取消对函数的安全检查。

> **注意**
>
> 缓冲区安全检查提供了重要的安全保护，并且对性能的影响微不足道。 因此，我们建议您不要取消它们，但在以下罕见的情况下除外：函数的性能至关重要，并且已经知道函数是安全的。

内联函数

*主函数*可以使用 inlining 关键字插入*辅助函数*的副本。 如果将 `__declspec(safebuffers)` 关键字应用于函数，则会取消对该函数的缓冲区溢出检测。 但是，内联将通过以下几种方式影响 `__declspec(safebuffers)` 关键字。

假设为这两种函数都指定了 /GS 编译器选项，但主函数指定了 `__declspec(safebuffers)` 关键字。 辅助函数中的数据结构将使它符合安全检查的条件，因此该函数不会取消这些检查。 在这种情况下：

- 在辅助函数上指定 __forceinline 关键字以强制编译器内联到该函数（无论编译器优化情况如何）。

- 由于辅助函数符合安全检查的条件，因此即使主函数指定了 `__declspec(safebuffers)` 关键字，也会对主函数应用安全检查。

示例

以下代码演示如何使用 `__declspec(safebuffers)` 关键字。

C++

```
// compile with: /c /GS
typedef struct {
    int x[20];
} BUFFER;
static int checkBuffers() {
    BUFFER cb;
    // Use the buffer...
    return 0;
}
static __declspec(safebuffers)
    int noCheckBuffers() {
    BUFFER ncb;
    // Use the buffer...
    return 0;
}
int wmain() {
    checkBuffers();
    noCheckBuffers();
    return 0;
}
```

结束 Microsoft 专用

另请参阅

__declspec
关键字
inline、__inline、__forceinline
strict_gs_check

Last updated on 2025/11/07

`selectany`

Microsoft 专用

告知编译器声明的全局数据项（变量或对象）是“任一拣选”COMDAT（已包装函数）。

语法

> `__declspec( selectany )` 声明符

注解

在链接时，如果显示了 COMDAT 的多个定义，则链接器会选取一个定义并丢弃其余的定义。 如果链接器选项 /OPT:REF（优化）处于选中状态，则将进行 COMDAT 清除以移除链接器输出中所有未引用的数据项。

声明中的构造函数和全局函数或静态方法的赋值不创建引用，并且不会阻止 /OPT: REF 清除。 此类代码的副作用不应依赖于不存在对数据的其他引用的情况。

对于动态初始化的全局对象，`selectany` 也将放弃一个未引用对象的初始化代码。

全局数据项一般只能在 EXE 或 DLL 项目中初始化一次。 当相同的标头出现在多个源文件中时，`selectany` 可用于初始化由标头定义的全局数据。 `selectany` 可用于 C 和 C++ 编译器。

> **注意**
>
> `selectany` 只能应用于外部可见的全局数据项的实际初始化。

示例：`selectany` 特性

此代码演示如何使用 `selectany` 特性：

C++

```
//Correct - x1 is initialized and externally visible
__declspec(selectany) int x1=1;

//Incorrect - const is by default static in C++, so
//x2 is not visible externally (This is OK in C, since
//const is not by default static in C)
const __declspec(selectany) int x2 =2;

//Correct - x3 is extern const, so externally visible
```

```
extern const __declspec(selectany) int x3=3;

//Correct - x4 is extern const, so it is externally visible
extern const int x4;
const __declspec(selectany) int x4=4;

//Incorrect - __declspec(selectany) is applied to the uninitialized
//declaration of x5
extern __declspec(selectany) int x5;

// OK: dynamic initialization of global object
class X {
public:
X(int i){i++;}
int i;
};

__declspec(selectany) X x(1);
```

示例：使用 `selectany` 特性来确保数据 COMDAT 折叠

此代码说明了如何在也使用 `selectany` 链接器选项时使用 /OPT:ICF 特性来确保数据 COMDAT 折叠。 请注意，数据必须有 `selectany` 标记并放置在 `const` （只读）部分中。 您必须显式指定只读部分。

C++

```
// selectany2.cpp
// in the following lines, const marks the variables as read only
__declspec(selectany) extern const int ix = 5;
__declspec(selectany) extern const int jx = 5;
int main() {
   int ij;
   ij = ix + jx;
}
```

结束 Microsoft 专用

另请参阅

__declspec
关键字

Last updated on 2025/11/07

spectre

Microsoft 专用

告知编译器不要为函数插入 Spectre 变体 1 推理执行屏障指令。

语法

> __declspec(spectre(nomitigation))

注解

/Qspectre 编译器选项会导致编译器插入推理执行屏障指令。 它们已插入到分析表明存在 Spectre 变体 1 安全漏洞的位置。 发出的特定指令取决于处理器。 虽然这些指令对代码大小或性能的影响应该是最小的，但在某些情况下，代码不受漏洞影响，并且需要最佳性能。

专家分析可以确定函数不会受到 Spectre 变体 1 边界检查绕过缺陷的影响。 在这种情况下，可以通过将 `__declspec(spectre(nomitigation))` 应用于函数声明来禁止函数中缓解代码的生成。

> **⊗ 注意**
>
> /Qspectre 推理执行屏障指令提供重要的安全保护，对性能的影响可以忽略不计。 因此，我们建议您不要取消它们，但在以下罕见的情况下除外：函数的性能至关重要，并且已经知道函数是安全的。

示例

下面的代码演示如何使用 `__declspec(spectre(nomitigation))`。

C++

```
// compile with: /c /Qspectre
static __declspec(spectre(nomitigation))
int noSpectreIssues() {
    // No Spectre variant 1 vulnerability here
    // ...
    return 0;
}

int main() {
    noSpectreIssues();
    return 0;
}
```

结束 Microsoft 专用

另请参阅

__declspec
关键字
/Qspectre

Last updated on 2025/11/07

会话

Microsoft 专用

`thread` 扩展存储类修饰符用于声明线程本地变量。 对于 C++11 及更高版本中的可移植等效，请将 thread_local 存储类说明符用于可移植代码。 在 Windows 上，`thread_local` 是使用 `__declspec(thread)` 实现的。

语法

`__declspec(thread)` 声明符

注解

线程本地存储 (TLS) 是多线程进程中的每个线程为线程特定的数据分配存储时所采用的机制。 在标准多线程程序中，数据在给定进程的所有线程间共享，而线程本地存储是用于分配每个线程数据的机制。 有关线程的完整讨论，请参阅多线程。

线程本地变量的声明必须将扩展的特性语法与 `__declspec` 关键字和 `thread` 关键字一起使用。 例如，以下代码声明了一个整数线程局部变量，并用一个值对其进行初始化：

C++

```
__declspec( thread ) int tls_i = 1;
```

在动态加载的库中使用线程局部变量时，需注意可能导致线程局部变量无法正确初始化的因素：

1. 如果变量是通过函数调用（包括构造函数）初始化的，则只会为导致二进制文件/DLL 加载到进程中的线程，以及在加载二进制文件/DLL 之后启动的线程调用此函数。 对于加载 DLL 时已经在运行的其他任何线程，不会调用初始化函数。 动态初始化在 DLL_THREAD_ATTACH 的 DllMain 调用时进行，但如果线程启动时 DLL 不在进程中，DLL 永远不会收到该消息。

2. 使用常量值静态初始化的线程局部变量通常在所有线程上正确初始化。 但截至 2017 年 12 月，Microsoft C++ 编译器有一个已知的一致性问题，即 `constexpr` 变量接收动态而不是静态初始化。

   注意：预计这两个问题都将在编译器的未来更新中得到修复。

另外，在声明线程本地对象和变量时必须遵守下列准则：

- 可以只将 `thread` 特性应用于类和数据声明和定义；`thread` 不能用于函数声明或定义。

- 只能在具有静态存储持续时间的数据项上指定 `thread` 特性。 这包括全局数据对象（`static` 和 `extern`）、本地静态对象和类的静态数据成员。 不能声明带 `thread` 特性的自动数据对象。
- 必须为线程本地对象的声明和定义使用 `thread` 特性，无论声明和定义是在同一文件中还是单独的文件中发生。
- 无法将 `thread` 特性用作类型修饰符。
- 由于允许使用 `thread` 特性的对象的声明，因此下面的两个示例在语义上是等效的：

  C++

  ```
  // declspec_thread_2.cpp
  // compile with: /LD
  __declspec( thread ) class B {
  public:
     int data;
  } BObject;   // BObject declared thread local.

  class B2 {
  public:
     int data;
  };
  __declspec( thread ) B2 BObject2;   // BObject2 declared thread local.
  ```

- 标准 C 允许使用涉及引用自身的表达式初始化对象或变量，但只适用于非静态对象。 虽然 C++ 通常允许使用涉及引用自身的表达式动态初始化对象，但是不允许将这种类型的初始化用于线程本地对象。 例如：

  C++

  ```
  // declspec_thread_3.cpp
  // compile with: /LD
  #define Thread __declspec( thread )
  int j = j;   // Okay in C++; C error
  Thread int tls_i = sizeof( tls_i );   // Okay in C and C++
  ```

  包含将初始化的对象的 `sizeof` 表达式不构成对自身的引用，允许在 C 和 C++ 中使用该表达式。

结束 Microsoft 专用

另请参阅

__declspec
关键字
线程本地存储 (TLS)

Last updated on 2025/11/07

uuid （C++）

Microsoft 专用

编译器将 GUID 附加到使用 `uuid` 特性声明或定义的（仅完整的 COM 对象定义）类或结构中。

语法

```
__declspec( uuid("ComObjectGUID") ) declarator
```

备注

`uuid` 特性采用字符串作为其自变量。 此字符串采用带有或不带 { } 分隔符的普通注册表格式命名 GUID。 例如：

```C++
struct __declspec(uuid("00000000-0000-0000-c000-000000000046")) IUnknown;
struct __declspec(uuid("{00020400-0000-0000-c000-000000000046}")) IDispatch;
```

此特性可应用于重新声明。 这允许系统标头提供接口定义（如 `IUnknown`），并允许其他标头（如 <comdef.h>）中的重新声明提供 GUID。

可以应用关键字 __uuidof 来检索附加到用户定义的类型的常量 GUID。

结束 Microsoft 专用

另请参阅

__declspec
关键字

Last updated on 2025/11/07

`__restrict`

与 `__declspec` (restrict) 修饰符一样， 关键字（两个前导下划线"_"）指示符号在当前范围中未使用别名 `__restrict`。 `__restrict` 关键字与 `__declspec (restrict)` 修饰符在下列方面不同：

- `__restrict` 关键字仅对变量有效，而 `__declspec (restrict)` 仅对函数声明和函数定义有效。
- 对于从 C99 开始的 C， `__restrict` 与 restrict 类似，并且可在 /std:c11 或 /std:c17 模式下使用，但 `__restrict` 在 C++ 和 C 程序中均可使用。
- 使用 `__restrict` 时，编译器将不会传播变量的非别名属性。 即，如果你向非 `__restrict` 变量分配 `__restrict` 变量，则编译器仍允许非 __restrict 变量使用别名。 这与 C99 C 语言 `restrict` 关键字的行为不同。

通常，如果想要影响整个函数的行为，请使用 `__declspec (restrict)` 而不是关键字。

为了与以前的版本兼容，除非指定了编译器选项 `__restrict`，否则 `/Za` 是 的同义词。

在 Visual Studio 2015 及更高版本中，可以对 C++ 引用使用 `__restrict`。

> ⓘ **注意**
>
> 在同时包含 volatile 关键字的变量上使用时， `volatile` 将优先。

示例

C++

```
// __restrict_keyword.c
// compile with: /LD
// In the following function, declare a and b as disjoint arrays
// but do not have same assurance for c and d.
void sum2(int n, int * __restrict a, int * __restrict b,
          int * c, int * d) {
   int i;
   for (i = 0; i < n; i++) {
      a[i] = b[i] + c[i];
      c[i] = b[i] + d[i];
    }
}

// By marking union members as __restrict, tell compiler that
// only z.x or z.y will be accessed in any given scope.
union z {
```

```
    int * __restrict x;
    double * __restrict y;
};
```

另请参阅

关键字

Last updated on 2025/11/07

__sptr、__uptr

Microsoft 专用

在 32 位指针声明上使用 `__sptr` 或 `__uptr` 修饰符，以指定编译器如何将 32 位指针转换为 64 位指针。 例如，在将 32 位指针分配给 64 位指针变量或在 64 位平台上取消引用 64 位指针时，将转换该指针。

为 64 位平台提供支持的 Microsoft 文档有时将 32 位指针的最高有效位用作符号位。 默认情况下，编译器使用符号扩展来将 32 位指针转换为 64 位指针。 即，将 64 位指针的最低有效 32 位设置为 32 位指针的值，并将最高有效 32 位设置为 32 位指针的符号位的值。 如果符号位为 0，则此转换会生成正确的结果；如果符号位为 1，则此转换不会生成正确的结果。 例如，32 位地址 0x7FFFFFFF 生成等效的 64 位地址 0x000000007FFFFFFF，但 32 位地址 0x80000000 未正确地更改为 0xFFFFFFFF80000000。

`__sptr` 或带符号的指针修饰符指定指针转换将 64 位指针的最高有效位设置为 32 位指针的符号位。 `__uptr` 或不带符号的指针修饰符指定转换将最高有效位设置为零。 下面的声明展示了用于两个非限定指针的 `__sptr` 和 `__uptr` 修饰符、两个使用 __ptr32 类型限定的指针和一个函数参数。

C++

```
int * __sptr psp;
int * __uptr pup;
int * __ptr32 __sptr psp32;
int * __ptr32 __uptr pup32;
void MyFunction(char * __uptr __ptr32 myValue);
```

将 `__sptr` 和 `__uptr` 修饰符用于指针声明。 在指针类型限定符的位置使用修饰符，这意味着修饰符必须在星号的后面。 不能将修饰符用于指向成员的指针。 修饰符不影响非指针声明。

为了与以前的版本兼容，除非指定了编译器选项 **/Za（禁用语言扩展）**，否则 _sptr 和 _uptr 是和 `__sptr` 的同义词`__uptr`。

示例

下面的示例声明了使用 `__sptr` 和 `__uptr` 修饰符的 32 位指针，将每个 32 位指针分配给 64 位指针变量，然后显示每个 64 位指针的十六进制值。 该示例利用本机 64 位编译器进行编译并在 64 位平台上执行。

C++

```
// sptr_uptr.cpp
// processor: x64
```

```
#include "stdio.h"

int main()
{
    void *        __ptr64 p64;
    void *        __ptr32 p32d; //default signed pointer
    void * __sptr __ptr32 p32s; //explicit signed pointer
    void * __uptr __ptr32 p32u; //explicit unsigned pointer

// Set the 32-bit pointers to a value whose sign bit is 1.
    p32d = reinterpret_cast<void *>(0x87654321);
    p32s = p32d;
    p32u = p32d;

// The printf() function automatically displays leading zeroes with each 32-bit
pointer. These are unrelated
// to the __sptr and __uptr modifiers.
    printf("Display each 32-bit pointer (as an unsigned 64-bit pointer):\n");
    printf("p32d:       %p\n", p32d);
    printf("p32s:       %p\n", p32s);
    printf("p32u:       %p\n", p32u);

    printf("\nDisplay the 64-bit pointer created from each 32-bit pointer:\n");
    p64 = p32d;
    printf("p32d: p64 = %p\n", p64);
    p64 = p32s;
    printf("p32s: p64 = %p\n", p64);
    p64 = p32u;
    printf("p32u: p64 = %p\n", p64);
    return 0;
}
```

Output

```
Display each 32-bit pointer (as an unsigned 64-bit pointer):
p32d:       0000000087654321
p32s:       0000000087654321
p32u:       0000000087654321

Display the 64-bit pointer created from each 32-bit pointer:
p32d: p64 = FFFFFFFF87654321
p32s: p64 = FFFFFFFF87654321
p32u: p64 = 0000000087654321
```

结束 Microsoft 专用

另请参阅

Microsoft 专用的修饰符

Last updated on 2025/11/07

`__unaligned`

Microsoft 专用。 声明带有 `__unaligned` 修饰符的指针时，编译器将假定该指针指向未对齐的数据。 因此，会生成平台适用的代码来通过指针处理未对齐的读取和写入。

备注

此修饰符描述指针指向的数据的对齐。 假定指针本身已对齐。

`__unaligned` 关键字的必要性因平台和环境而异。 无法适当标记数据可能会导致从性能损失到硬件故障等问题。 `__unaligned` 修饰符对 x86 平台无效。

为了与以前的版本兼容，除非指定了编译器选项 `__unaligned`，否则 `/Za` 是 的同义词。

有关对齐的详细信息，请参阅：

- align
- alignof 运算符
- pack
- /Zp（结构成员对齐）
- x64 结构对齐示例

另请参阅

关键字

Last updated on 2025/11/07

__w64

此 Microsoft 专用关键字已过时。 在 Visual Studio 2013 之前的版本中，允许标记变量，以确保在使用 /Wp64 进行编译时，编译器将报告使用 64 位编译器编译时可能报告的任何警告。

语法

> 类型 `__w64` 标识符

参数

type
可导致在从 32 位编译器移植到 64 位编译器时出现问题的三种类型之一： `int`、 `long` 或指针。

identifier
要创建变量的标识符。

注解

> ⓘ **重要**
>
> /Wp64 编译器选项和 `__w64` 关键字在 Visual Studio 2010 和 Visual Studio 2013 中已弃用，并从 Visual Studio 2013 起删除。 如果在命令行中使用 `/Wp64` 编译器选项，编译器将发出命令行警告 D9002。 `__w64` 关键字会自动忽略。 请勿使用此选项和关键字来检测 64 位可移植性问题，请使用面向 64 位平台的 Visual C++ 编译器。 有关详细信息，请参阅**为 64 位配置 Visual C++，面向 x64**。

任何具有 `__w64` 的 typedef 都必须在 x86 上是 32 位，在 x64 上是 64 位。

若要通过使用早于 Visual Studio 2010 的 Visual C++ 编译器版本来检测可移植性问题，应在任何可跨 32 位平台与 64 位平台变换大小的 typedef 上指定 `__w64` 关键字。 对于任何此类类型，`__w64` 必须仅在 typedef 的 32 位定义上出现。

为了与以前的版本兼容，_w64 是 的同义词，除非指定了编译器选项 `__w64`。

如果编译不使用 `__w64`，则将忽略 `/Wp64` 关键字。

有关移植到 64 位编译器的详细信息，请参阅以下主题：

- MSVC 编译器选项

- 将 32 位代码移植到 64 位代码
- 针对 64 位 x64 目标配置 Visual C++

示例

C++

```
// __w64.cpp
// compile with: /W3 /Wp64
typedef int Int_32;
#ifdef _WIN64
typedef __int64 Int_Native;
#else
typedef int __w64 Int_Native;
#endif

int main() {
   Int_32 i0 = 5;
   Int_Native i1 = 10;
   i0 = i1;   // C4244 64-bit int assigned to 32-bit int

   // char __w64 c;  error, cannot use __w64 on char
}
```

另请参阅

关键字

Last updated on 2025/11/07

`__func__`

(C++11) 预定义标识符 隐式定义为包含封闭函数的未限定和未修饰名称的字符串。`__func__` `__func__` 由 C++ 标准规定，且不是 Microsoft 扩展。

语法

C++

```
__func__
```

返回值

返回 null 终止的 const char 字符数组，该数组包含函数名称。

示例

C++

```
#include <string>
#include <iostream>

namespace Test
{
    struct Foo
    {
        static void DoSomething(int i, std::string s)
        {
           std::cout << __func__ << std::endl; // Output: DoSomething
        }
    };
}

int main()
{
    Test::Foo::DoSomething(42, "Hello");

    return 0;
}
```

要求

C++11

Last updated on 2025/11/07

编译器 COM 支持

Microsoft 专用

Microsoft C++ 编译器可以直接读取组件对象模型 (COM) 类型库，并将内容转换为可包含在编译中的 C++ 源代码。 提供了语言扩展来帮助在适用于桌面应用的客户端上进行 COM 编程。

通过使用 #import 预处理器指令，编译器可以读取类型库，并将其转换为将 COM 接口描述为类的 C++ 头文件。 提供了一组 `#import` 特性来实现对生成的类型库头文件的内容的用户控制。

可以使用 __declspec 扩展特性 uuid 将全局唯一标识符 (GUID) 分配给 COM 对象。 关键字 __uuidof 可用于提取与 COM 对象关联的 GUID。 另一个 **`__declspec`** 特性（属性）可用于为 COM 对象的数据成员指定 `get` 和 `set` 方法。

提供了一组 COM 支持全局函数和类来支持 `VARIANT` 和 `BSTR` 类型、实现智能指针以及封装 `_com_raise_error` 引发的错误对象：

- 编译器 COM 全局函数
- _bstr_t
- _com_error
- _com_ptr_t
- _variant_t

结束 Microsoft 专用

另请参阅

编译器 COM 支持类
编译器 COM 全局函数

Last updated on 2025/11/07

编译器 COM 全局函数

Microsoft 专用

下列例程可用：

展开表

Function	说明
_com_raise_error	引发 _com_error 来响应失败。
_set_com_error_handler	替换用于 COM 错误处理的默认函数。
ConvertBSTRToString	将 `BSTR` 值转换为 `char *`。
ConvertStringToBSTR	将 `char *` 值转换为 `BSTR`。

结束 Microsoft 专用

另请参阅

编译器 COM 支持类
编译器 COM 支持

Last updated on 2025/11/07

_com_raise_error

Microsoft 专用

引发 _com_error 来响应失败。

语法

C++

```
void __stdcall _com_raise_error(
   HRESULT hr,
   IErrorInfo* perrinfo = 0
);
```

参数

hr
HRESULT 信息。

perrinfo
`IErrorInfo` 对象。

备注

comdef.h< 中定义的 _com_raise_error 可以替换为具有相同的名称和原型的用户编写的版本>。 若要使用 `#import` 但不使用 C++ 异常处理，则可以执行此操作。 在这种情况下，_com_raise_error 的用户版本可能决定执行 或显示消息框并暂停`longjmp`。 但不应返回用户版本，因为编译器 COM 支持代码不希望返回它。

你也可以使用 _set_com_error_handler 来替换默认的错误处理函数。

默认情况下，_com_raise_error 的定义如下：

C++

```
void __stdcall _com_raise_error(HRESULT hr, IErrorInfo* perrinfo) {
   throw _com_error(hr, perrinfo);
}
```

结束 Microsoft 专用

要求

标头： <comdef.h>

库：如果启用了“wchar_t 是本机类型”编译器选项，请使用 comsuppw.lib 或 comsuppwd.lib。 如果关闭了“wchar_t 是本机类型”，请使用 comsupp.lib。 有关详细信息，请参阅 /Zc:wchar_t（wchar_t 是本机类型）。

另请参阅

编译器 COM 全局函数
_set_com_error_handler

Last updated on 2025/11/07

ConvertStringToBSTR

项目 • 2023/04/03

Microsoft 专用

将 `char *` 值转换为 `BSTR`。

语法

```
BSTR __stdcall ConvertStringToBSTR(const char* pSrc)
```

参数

pSrc
变量 `char *`。

示例

C++

```cpp
// ConvertStringToBSTR.cpp
#include <comutil.h>
#include <stdio.h>

#pragma comment(lib, "comsuppw.lib")
#pragma comment(lib, "kernel32.lib")

int main() {
   char* lpszText = "Test";
   printf_s("char * text: %s\n", lpszText);

   BSTR bstrText = _com_util::ConvertStringToBSTR(lpszText);
   wprintf_s(L"BSTR text: %s\n", bstrText);

   SysFreeString(bstrText);
}
```

Output

```
char * text: Test
BSTR text: Test
```

结束 Microsoft 专用

要求

标头：<comutil.h>

Lib：comsuppw.lib 或 comsuppwd.lib（有关详细信息，请参阅 /Zc:wchar_t（wchar_t 为本机类型））

另请参阅

编译器 COM 全局函数

ConvertBSTRToString

Microsoft 专用

将 `BSTR` 值转换为 `char *`。

语法

```
char* __stdcall ConvertBSTRToString(BSTR pSrc);
```

参数

pSrc
BSTR 变量。

注解

ConvertBSTRToString 分配必须删除的字符串。

示例

C++

```cpp
// ConvertBSTRToString.cpp
#include <comutil.h>
#include <stdio.h>

#pragma comment(lib, "comsuppw.lib")

int main() {
   BSTR bstrText = ::SysAllocString(L"Test");
   wprintf_s(L"BSTR text: %s\n", bstrText);

   char* lpszText2 = _com_util::ConvertBSTRToString(bstrText);
   printf_s("char * text: %s\n", lpszText2);

   SysFreeString(bstrText);
   delete[] lpszText2;
}
```

Output

```
BSTR text: Test
char * text: Test
```

结束 Microsoft 专用

要求

标头：<comutil.h>

Lib：comsuppw.lib 或 comsuppwd.lib（有关详细信息，请参阅 /Zc:wchar_t（wchar_t 为本机类型））

另请参阅

编译器 COM 全局函数

Last updated on 2025/11/07

_set_com_error_handler

替换用于 COM 错误处理的默认函数。 _set_com_error_handler 特定于 Microsoft。

语法

C++

```
void __stdcall _set_com_error_handler(
   void (__stdcall *pHandler)(
      HRESULT hr,
      IErrorInfo* perrinfo
   )
);
```

参数

pHandler
指向替换函数的指针。

hr
HRESULT 信息。

perrinfo
`IErrorInfo` 对象。

注解

默认情况下，_com_raise_error 处理所有 COM 错误。 可以使用 _set_com_error_handler 调用自己的错误处理函数来更改此行为。

替换函数必须具有与 `_com_raise_error` 的签名等效的签名。

示例

C++

```
// _set_com_error_handler.cpp
// compile with /EHsc
#include <stdio.h>
#include <comdef.h>
#include <comutil.h>

// Importing ado dll to attempt to establish an ado connection.
```

```
// Not related to _set_com_error_handler
#import "C:\Program Files\Common Files\System\ado\msado15.dll" no_namespace
rename("EOF", "adoEOF")

void __stdcall _My_com_raise_error(HRESULT hr, IErrorInfo* perrinfo)
{
   throw "Unable to establish the connection!";
}

int main()
{
   _set_com_error_handler(_My_com_raise_error);
   _bstr_t bstrEmpty(L"");
   _ConnectionPtr Connection = NULL;
   try
   {
      Connection.CreateInstance(__uuidof(Connection));
      Connection->Open(bstrEmpty, bstrEmpty, bstrEmpty, 0);
   }
   catch(char* errorMessage)
   {
      printf("Exception raised: %s\n", errorMessage);
   }

   return 0;
}
```

Output

```
Exception raised: Unable to establish the connection!
```

要求

标头： <comdef.h>

库： 如果指定了 /Zc:wchar_t 编译器选项（默认值），请使用 comsuppw.lib 或 comsuppwd.lib。 如果指定了 /Zc:wchar_t- 编译器选项，请使用 comsupp.lib。 有关详细信息（包括如何在 IDE 中设置此选项），请参阅 /Zc:wchar_t（wchar_t 本机类型）。

另请参阅

编译器 COM 全局函数

Last updated on 2025/11/07

编译器 COM 支持类

Microsoft 专用

标准类用于支持某些 COM 类型。 这些类是在从类型库生成的 <comdef.h> 文件和头文件中定义的。

展开表

类	目的
_bstr_t	包装 `BSTR` 类型，以提供有用的运算符和方法。
_com_error	定义在大多数失败中 _com_raise_error 引发的错误对象。
_com_ptr_t	封装 COM 接口指针，并且自动执行对 `AddRef`、`Release` 和 `QueryInterface` 的必需调用。
_variant_t	包装 `VARIANT` 类型，以提供有用的运算符和方法。

结束 Microsoft 专用

另请参阅

编译器 COM 支持
编译器 COM 全局函数
C++ 语言参考

Last updated on 2025/11/07

`_bstr_t` 类

Microsoft 专用

`_bstr_t` 对象可封装 BSTR 数据类型。 该类通过在适当时对 SysAllocString、SysFreeString 和其他 `BSTR` API 进行函数调用来管理资源分配和解除分配。 `_bstr_t` 类使用引用计数来避免开销过大。

成员

建筑

展开表

构造函数	说明
_bstr_t	构造 `_bstr_t` 对象。

Operations

展开表

函数	说明
Assign	将 `BSTR` 复制到 `BSTR` 包装的 `_bstr_t` 中。
Attach	将 `_bstr_t` 包装器链接到 `BSTR`。
copy	构造封装的 `BSTR` 的副本。
Detach	返回 `BSTR` 包装的 `_bstr_t` 并从 `BSTR` 中分离 `_bstr_t`。
GetAddress	指向 `BSTR` 包装的 `_bstr_t`。
GetBSTR	指向 `BSTR` 包装的 `_bstr_t` 的开头。
length	返回 `_bstr_t` 中的字符数。

运算符

展开表

运算符	说明
operator =	将新值赋给现有 `_bstr_t` 对象。
operator +=	将字符附加到 `_bstr_t` 对象的结尾。
operator +	串联两个字符串。
operator !	检查封装的 `BSTR` 是否为 NULL 字符串。
operator == operator != operator < operator > operator <= operator >=	比较两个 `_bstr_t` 对象。
operator wchar_t* operator char*	提取指向封装的 Unicode 或多字节 `BSTR` 对象的指针。

结束 Microsoft 专用

要求

标头：<comutil.h>

Lib：*`comsuppw.lib`* 或 *`comsuppwd.lib`*（有关详细信息，请参阅 /Zc:wchar_t（wchar_t 是本机类型））

另请参阅

编译器 COM 支持类

Last updated on 2025/11/07

`_bstr_t::_bstr_t`

Microsoft 专用

构造 `_bstr_t` 对象。

语法

C++

```
_bstr_t( ) throw( );
_bstr_t(
   const _bstr_t& s1
) throw( );
_bstr_t(
   const char* s2
);
_bstr_t(
   const wchar_t* s3
);
_bstr_t(
   const _variant_t& var
);
_bstr_t(
   BSTR bstr,
   bool fCopy
);
```

参数

`s1`
要复制的 `_bstr_t` 对象。

`s2`
多字节字符串。

`s3`
Unicode 字符串

`var`
_variant_t 对象。

`bstr`
一个现有的 `BSTR` 对象。

`fCopy`
如果为 **`false`**，则 *`bstr`* 参数将附加到新对象，而无需通过调用 `SysAllocString` 创建副本。

注解

`_bstr_t` 类提供了多个构造函数：

`_bstr_t( )`
构造封装空 `_bstr_t` 对象的默认 `BSTR` 对象。

`_bstr_t( _bstr_t& s1 )`
构造一个 `_bstr_t` 对象作为另一个对象的副本。 此构造函数创建一个浅表副本，它增加封装的对象的引用计数，而不是创建新对象`BSTR`。

`_bstr_t( char* s2 )`
通过调用 `_bstr_t` 来创建一个新的 `SysAllocString` 对象并封装该对象，从而构造一个 `BSTR` 对象。 此构造函数首先执行多字节到 Unicode 的转换。

`_bstr_t( wchar_t* s3 )`
通过调用 `_bstr_t` 来创建一个新的 `SysAllocString` 对象并封装该对象，从而构造一个 `BSTR` 对象。

`_bstr_t( _variant_t& var )`
通过先从封装的 `_bstr_t` 对象检查 `_variant_t` 对象来从 `BSTR` 对象构造一个 `VARIANT` 对象。

`_bstr_t( BSTR bstr, bool fCopy )`
从现有 `_bstr_t` 构造一个 `BSTR` 对象（与 `wchar_t*` 字符串相对）。 如果 *`fCopy`* 为 **`false`**，则提供的 `BSTR` 将附加到新对象，而无需通过使用 `SysAllocString` 创建新副本。 此构造函数由类型库标头中的包装器函数用于封装接口方法所返回的 `BSTR` 并获取其所有权。

结束 Microsoft 专用

另请参阅

_bstr_t 类
_variant_t 类

Last updated on 2025/11/07

`_bstr_t::Assign`

Microsoft 专用

将 `BSTR` 复制到 `BSTR` 包装的 `_bstr_t` 中。

语法

C++

```
void Assign(
   BSTR s
);
```

参数

`s`
要复制到 `BSTR` 包装的 `BSTR` 中的 `_bstr_t`。

备注

无论内容是什么，`Assign` 都会对 `BSTR` 的整个长度进行二进制复制。

示例

C++

```
// _bstr_t_Assign.cpp

#include <comdef.h>
#include <stdio.h>

int main()
{
    // creates a _bstr_t wrapper
    _bstr_t bstrWrapper;

    // creates BSTR and attaches to it
    bstrWrapper = "some text";
    wprintf_s(L"bstrWrapper = %s\n",
              static_cast<wchar_t*>(bstrWrapper));

    // bstrWrapper releases its BSTR
    BSTR bstr = bstrWrapper.Detach();
    wprintf_s(L"bstrWrapper = %s\n",
```

```
            static_cast<wchar_t*>(bstrWrapper));
    // "some text"
    wprintf_s(L"bstr = %s\n", bstr);

    bstrWrapper.Attach(SysAllocString(OLESTR("SysAllocedString")));
    wprintf_s(L"bstrWrapper = %s\n",
            static_cast<wchar_t*>(bstrWrapper));

    // assign a BSTR to our _bstr_t
    bstrWrapper.Assign(bstr);
    wprintf_s(L"bstrWrapper = %s\n",
            static_cast<wchar_t*>(bstrWrapper));

    // done with BSTR, do manual cleanup
    SysFreeString(bstr);

    // resuse bstr
    bstr= SysAllocString(OLESTR("Yet another string"));
    // two wrappers, one BSTR
    _bstr_t bstrWrapper2 = bstrWrapper;

    *bstrWrapper.GetAddress() = bstr;

    // bstrWrapper and bstrWrapper2 do still point to BSTR
    bstr = 0;
    wprintf_s(L"bstrWrapper = %s\n",
            static_cast<wchar_t*>(bstrWrapper));
    wprintf_s(L"bstrWrapper2 = %s\n",
            static_cast<wchar_t*>(bstrWrapper2));

    // new value into BSTR
    _snwprintf_s(bstrWrapper.GetBSTR(), 100, bstrWrapper.length(),
                L"changing BSTR");
    wprintf_s(L"bstrWrapper = %s\n",
            static_cast<wchar_t*>(bstrWrapper));
    wprintf_s(L"bstrWrapper2 = %s\n",
            static_cast<wchar_t*>(bstrWrapper2));
}
```

Output

```
bstrWrapper = some text
bstrWrapper = (null)
bstr = some text
bstrWrapper = SysAllocedString
bstrWrapper = some text
bstrWrapper = Yet another string
bstrWrapper2 = some text
bstrWrapper = changing BSTR
bstrWrapper2 = some text
```

结束 Microsoft 专用

另请参阅

_bstr_t 类

Last updated on 2025/11/07

`_bstr_t::Attach`

Microsoft 专用

将 `_bstr_t` 包装器链接到 `BSTR`。

语法

C++

```
void Attach(
   BSTR s
);
```

参数

`s`
要与之关联或分配到的 `BSTR`，即 `_bstr_t` 变量。

注解

如果 `_bstr_t` 以前被附加到了另一个 `BSTR`，并且没有其他 `_bstr_t` 变量使用 `BSTR`，`_bstr_t` 将清理 `BSTR` 资源。

示例

有关使用 _bstr_t::Assign 的示例，请参阅 **`Attach`**。

结束 Microsoft 专用

另请参阅

_bstr_t 类

Last updated on 2025/11/07

`_bstr_t::copy`

Microsoft 专用

构造封装的 `BSTR` 的副本。

语法

```
C++
BSTR copy( bool fCopy = true ) const;
```

参数

`fCopy`
如果为 **`true`**，则 **`copy`** 将返回包含的 `BSTR` 的副本；否则，**`copy`** 将返回实际的 `BSTR`。

备注

返回封装的 `BSTR` 对象的新分配的副本，或封装的对象本身，具体取决于该参数。

示例

```
C++
STDMETHODIMP CAlertMsg::get_ConnectionStr(BSTR *pVal){ //  m_bsConStr is _bstr_t
   *pVal = m_bsConStr.copy();
}
```

结束 Microsoft 专用

另请参阅

_bstr_t 类

Last updated on 2025/11/07

`_bstr_t::Detach`

Microsoft 专用

返回 `BSTR` 包装的 `_bstr_t` 并从 `BSTR` 中分离 `_bstr_t`。

语法

```
C++
BSTR Detach( ) throw;
```

返回值

返回 `BSTR` 封装的 `_bstr_t`。

示例

有关使用 _bstr_t::Assign 的示例，请参阅 **`Detach`**。

结束 Microsoft 专用

另请参阅

_bstr_t 类

Last updated on 2025/11/07

`_bstr_t::GetAddress`

Microsoft 专用

释放所有现有字符串并返回一个新分配的字符串的地址。

语法

```
C++
BSTR* GetAddress( );
```

返回值

指向由 `BSTR` 包装的 `_bstr_t` 的指针。

备注

`GetAddress` 影响共享 `_bstr_t` 的所有 `BSTR` 对象。 多个 `_bstr_t` 可以通过使用复制构造函数和 `BSTR` 共享一个 `operator=`。

示例

有关使用 _bstr_t::Assign 的示例，请参阅 `GetAddress`。

结束 Microsoft 专用

另请参阅

_bstr_t 类

Last updated on 2025/11/07

`_bstr_t::GetBSTR`

Microsoft 专用

指向 `BSTR` 包装的 `_bstr_t` 的开头。

语法

```
C++
BSTR& GetBSTR( );
```

返回值

`BSTR` 包装的 `_bstr_t` 的开头。

注解

`GetBSTR` 影响共享 `_bstr_t` 的所有 `BSTR` 对象。 多个 `_bstr_t` 可以通过使用复制构造函数和 `BSTR` 共享一个 `operator=`。

示例

有关使用 _bstr_t::Assign 的示例，请参阅 **`GetBSTR`**。

结束 Microsoft 专用

另请参阅

_bstr_t 类

Last updated on 2025/11/07

`_bstr_t::length`

Microsoft 专用

返回 `_bstr_t` 中的字符数，但不包括封装的 `BSTR` 的终止 null 字符。

语法

C++

```
unsigned int length ( ) const throw( );
```

备注

结束 Microsoft 专用

另请参阅

_bstr_t 类

Last updated on 2025/11/07

`_bstr_t::operator =`

Microsoft 专用

将新值赋给现有 `_bstr_t` 对象。

语法

C++

```
_bstr_t& operator=(const _bstr_t& s1) throw ( );
_bstr_t& operator=(const char* s2);
_bstr_t& operator=(const wchar_t* s3);
_bstr_t& operator=(const _variant_t& var);
```

参数

`s1`
将分配给现有 `_bstr_t` 对象的 `_bstr_t` 对象。

`s2`
将分配给现有 `_bstr_t` 对象的多字节字符串。

`s3`
将分配给现有 `_bstr_t` 对象的 Unicode 字符串。

`var`
将分配给现有 `_variant_t` 对象的 `_bstr_t` 对象。

结束 Microsoft 专用

示例

有关使用 _bstr_t::Assign 的示例，请参阅 **`operator=`**。

另请参阅

_bstr_t 类

Last updated on 2025/11/07

%>

Microsoft 专用

将字符追加到 `_bstr_t` 对象的末尾，或连接两个字符串。

语法

C++

```
_bstr_t& operator+=( const _bstr_t& s1 );
_bstr_t operator+( const _bstr_t& s1 );
friend _bstr_t operator+( const char* s2, const _bstr_t& s1);
friend _bstr_t operator+( const wchar_t* s3, const _bstr_t& s1);
```

参数

s1
`_bstr_t` 对象。

s2
多字节字符串。

s3
一个 Unicode 字符串。

备注

以下运算符将执行字符串串联：

- `operator+=( s1 )` 将 `BSTR` 的已封装 *s1* 中的字符追加到此对象的已封装 `BSTR` 的末尾。
- `operator+( s1 )` 返回通过将此对象的 `_bstr_t` 与 `BSTR` 中的对应项进行连接构成的新 *s1*。
- `operator+( s2, s1 )` 返回通过将多字节字符串 `_bstr_t`（已转换为 Unicode）与 *s2* 中封装的 `BSTR` 进行连接构成的新 *s1*。
- `operator+( s3, s1 )` 返回通过将 Unicode 字符串 `_bstr_t` 与 *s3* 中封装的 `BSTR` 进行连接构成的新 *s1*。

结束 Microsoft 专用

另请参阅

_bstr_t 类

Last updated on 2025/11/07

`_bstr_t::operator !`

Microsoft 专用

检查封装的 `BSTR` 是否为 NULL 字符串。

语法

```
C++
bool operator!( ) const throw( );
```

返回值

如果封装的 **`true`** 为 NULL 字符串，则返回 `BSTR`，否则返回 **`false`**。

结束 Microsoft 专用

另请参阅

_bstr_t 类

Last updated on 2025/11/07

`_bstr_t` 关系运算符

Microsoft 专用

比较两个 `_bstr_t` 对象。

语法

C++

```
bool operator==(const _bstr_t& str) const throw( );
bool operator!=(const _bstr_t& str) const throw( );
bool operator<(const _bstr_t& str) const throw( );
bool operator>(const _bstr_t& str) const throw( );
bool operator<=(const _bstr_t& str) const throw( );
bool operator>=(const _bstr_t& str) const throw( );
```

备注

这些运算符按字典顺序比较两个 `_bstr_t` 对象。 如果该比较保留，运算符将返回 `true`；否则返回 `false`。

结束 Microsoft 专用

另请参阅

_bstr_t 类

Last updated on 2025/11/07

%>

Microsoft 专用

将 `BSTR` 字符作为一个窄字符数组或宽字符数组返回。

语法

C++

```
operator const wchar_t*( ) const throw( );
operator wchar_t*( ) const throw( );
operator const char*( ) const;
operator char*( ) const;
```

备注

这些运算符可用于提取由 `BSTR` 对象封装的字符数据。 为返回的指针赋予新值不会修改原始 `BSTR` 数据。

结束 Microsoft 专用

另请参阅

_bstr_t 类

Last updated on 2025/11/07

`_com_error` 类

Microsoft 专用

`_com_error` 对象表示由类型库生成的头文件中错误处理包装器函数或由 COM 支持类之一检测到的异常条件。 `_com_error` 类封装 `HRESULT` 错误代码和任何相关的 `IErrorInfo Interface` 对象。

建筑

展开表

名称	描述
_com_error	构造 `_com_error` 对象。

运算符

展开表

名称	描述
operator =	将现有 `_com_error` 对象赋给另一个对象。

提取程序函数

展开表

名称	描述
Error	检索传递给构造函数的 `HRESULT`。
ErrorInfo	检索传递给构造函数的 `IErrorInfo` 对象。
WCode	检索映射到封装的 `HRESULT` 中的 16 位错误代码。

`IErrorInfo` 函数

展开表

名称	描述
Description	调用 `IErrorInfo::GetDescription` 函数。
HelpContext	调用 `IErrorInfo::GetHelpContext` 函数。
HelpFile	调用 `IErrorInfo::GetHelpFile` 函数
Source	调用 `IErrorInfo::GetSource` 函数。
GUID	调用 `IErrorInfo::GetGUID` 函数。

设置消息提取程序的格式

展开表

名称	描述
ErrorMessage	检索 `HRESULT` 对象中存储的 `_com_error` 的字符串消息。

`ExepInfo.wCode` 到 `HRESULT` 映射器

展开表

名称	描述
HRESULTToWCode	将 32 位 `HRESULT` 映射到 16 位 `wCode`。
WCodeToHRESULT	将 16 位 `wCode` 映射到 32 位 `HRESULT`。

结束 Microsoft 专用

要求

标头： <comdef.h>

Library： *`comsuppw.lib`* 或 *`comsuppwd.lib`*（有关详细信息，请参阅 /Zc:wchar_t（wchar_t 是本机类型））

另请参阅

编译器 COM 支持类
IErrorInfo 接口

Last updated on 2025/11/07

`_com_error` 成员函数

有关 `_com_error` 成员函数的信息，请参阅 _com_error 类。

另请参阅

_com_error 类

Last updated on 2025/11/07

`_com_error::_com_error`

Microsoft 专用

构造 `_com_error` 对象。

语法

```
C++
_com_error(
   HRESULT hr,
   IErrorInfo* perrinfo = NULL,
   bool fAddRef = false) throw();

_com_error( const _com_error& that ) throw();
```

参数

`hr`
`HRESULT` 信息。

`perrinfo`
`IErrorInfo` 对象。

`fAddRef`
默认值会导致构造函数不在非空 `IErrorInfo` 接口上调用 AddRef。 此行为可以在接口的所有权传入 `_com_error` 对象的常见情况下提供正确的引用计数，例如：

```
C++
throw _com_error(hr, perrinfo);
```

如果不想让代码将所有权转交到 `_com_error` 对象，并且在 `AddRef` 析构函数中对 `Release` 进行偏移需要 `_com_error`，则按以下方式构造对象：

```
C++
_com_error err(hr, perrinfo, true);
```

`that`
一个现有的 `_com_error` 对象。

备注

第一个构造函数基于 `HRESULT` 和可选 `IErrorInfo` 对象创建一个新对象。 第二个构造函数创建现有 `_com_error` 对象的副本。

结束 Microsoft 专用

另请参阅

_com_error 类

Last updated on 2025/11/07

`_com_error::Description`

Microsoft 专用

调用 `IErrorInfo::GetDescription` 函数。

语法

```
C++
_bstr_t Description() const;
```

返回值

返回 `IErrorInfo::GetDescription` 对象中记录的 `IErrorInfo` 对象的 `_com_error` 结果。 生成的 `BSTR` 封装在 `_bstr_t` 对象中。 如果未记录 `IErrorInfo`，它将返回空 `_bstr_t`。

备注

调用 `IErrorInfo::GetDescription` 函数并检索 `IErrorInfo` 对象中记录的 `_com_error`。 调用 `IErrorInfo::GetDescription` 方法时的任何失败都将被忽略。

结束 Microsoft 专用

另请参阅

_com_error 类

Last updated on 2025/11/07

`_com_error::Error`

Microsoft 专用

检索传递给构造函数的 `HRESULT`。

语法

```
C++
HRESULT Error() const throw();
```

返回值

已传入构造函数的原始 `HRESULT` 项。

备注

检索 `HRESULT` 对象中封装的 `_com_error` 项。

结束 Microsoft 专用

另请参阅

_com_error 类

Last updated on 2025/11/07

_com_error::ErrorInfo

Microsoft 专用

检索传递给构造函数的 `IErrorInfo` 对象。

语法

```
C++
IErrorInfo * ErrorInfo( ) const throw( );
```

返回值

已传入构造函数的原始 `IErrorInfo` 项。

备注

检索 `IErrorInfo` 对象中的封装 `_com_error` 项；如果未记录 `NULL` 项，则返回 `IErrorInfo`。 使用完返回的对象后，调用方必须对该对象调用 `Release`。

结束 Microsoft 专用

另请参阅

_com_error 类

Last updated on 2025/11/07

`_com_error::ErrorMessage`

Microsoft 专用

检索 `HRESULT` 对象中存储的 `_com_error` 的字符串消息。

语法

```
C++
const TCHAR * ErrorMessage() const throw();
```

返回值

返回 `HRESULT` 对象中记录的 `_com_error` 的字符串消息。 如果 `HRESULT` 是映射的 16 位 wCode，则返回一般消息“`IDispatch error #<wCode>`”。 如果未找到消息，则返回一般消息“`Unknown error #<hresult>`”。 返回的字符串是 Unicode 或多字节字符串，具体取决于 `_UNICODE` 宏的状态。

备注

检索在 `HRESULT` 对象中记录的 `_com_error` 的对应系统消息文本。 系统消息文本通过调用 Win32 FormatMessage 函数获取。 返回的字符串由 `FormatMessage` API 分配，并在 `_com_error` 对象被销毁后释放。

结束 Microsoft 专用

另请参阅

_com_error 类

Last updated on 2025/11/07

`_com_error::GUID`

Microsoft 专用

调用 `IErrorInfo::GetGUID` 函数。

语法

```
C++
GUID GUID() const throw();
```

返回值

返回 `IErrorInfo::GetGUID` 对象中记录的 `IErrorInfo` 对象的 `_com_error` 结果。 如果未记录 `IErrorInfo` 对象，则返回 `GUID_NULL`。

备注

调用 `IErrorInfo::GetGUID` 方法时的任何失败都将被忽略。

结束 Microsoft 专用

另请参阅

_com_error 类

Last updated on 2025/11/07

`_com_error::HelpContext`

Microsoft 专用

调用 `IErrorInfo::GetHelpContext` 函数。

语法

C++

```
DWORD HelpContext() const throw();
```

返回值

返回 `IErrorInfo::GetHelpContext` 对象中记录的 `IErrorInfo` 对象的 `_com_error` 结果。 如果未记录 `IErrorInfo` 对象，则返回 0。

注解

调用 `IErrorInfo::GetHelpContext` 方法时的任何失败都将被忽略。

结束 Microsoft 专用

另请参阅

_com_error 类

Last updated on 2025/11/07

`_com_error::HelpFile`

Microsoft 专用

调用 `IErrorInfo::GetHelpFile` 函数。

语法

```
C++
_bstr_t HelpFile() const;
```

返回值

返回 `IErrorInfo::GetHelpFile` 对象中记录的 `IErrorInfo` 对象的 `_com_error` 结果。 生成的 BSTR 封装在 `_bstr_t` 对象中。 如果未记录 `IErrorInfo`，它将返回空 `_bstr_t`。

注解

调用 `IErrorInfo::GetHelpFile` 方法时的任何失败都将被忽略。

结束 Microsoft 专用

另请参阅

_com_error 类

Last updated on 2025/11/07

`_com_error::HRESULTToWCode`

Microsoft 专用

将 32 位 `HRESULT` 映射到 16 位 `wCode`。

语法

C++

```
static WORD HRESULTToWCode(
   HRESULT hr
) throw();
```

参数

`hr`
要映射为 16 位 `HRESULT` 的 32 位 `wCode`。

返回值

已从 32 位 `wCode` 映射的 16 位 `HRESULT`。

备注

有关详细信息，请参阅 _com_error::WCode。

结束 Microsoft 专用

另请参阅

_com_error::WCode
_com_error::WCodeToHRESULT
_com_error 类

Last updated on 2025/11/07

`_com_error::Source`

Microsoft 专用

调用 `IErrorInfo::GetSource` 函数。

语法

```
C++
_bstr_t Source() const;
```

返回值

返回 `IErrorInfo::GetSource` 对象中记录的 `IErrorInfo` 对象的 `_com_error` 结果。 生成的 `BSTR` 封装在 `_bstr_t` 对象中。 如果未记录 `IErrorInfo`，它将返回空 `_bstr_t`。

注解

调用 `IErrorInfo::GetSource` 方法时的任何失败都将被忽略。

结束 Microsoft 专用

另请参阅

_com_error 类

Last updated on 2025/11/07

`_com_error::WCode`

Microsoft 专用

检索映射到封装的 `HRESULT` 中的 16 位错误代码。

语法

```
C++
WORD WCode ( ) const throw();
```

返回值

如果 `HRESULT` 在 0x80040200 到 0x8004FFFF 范围内， `WCode` 方法会返回 `HRESULT` 减去 0x80040200；否则，该方法返回零。

备注

`WCode` 方法用于撤消发生在 COM 支持代码中的映射。 `dispinterface` 属性或方法的包装器将调用打包自变量的支持例程并调用 `IDispatch::Invoke`。 返回时，如果返回 `HRESULT` 的失败 `DISP_E_EXCEPTION`，则从传递给 `EXCEPINFO` 的 `IDispatch::Invoke` 结构中检索错误信息。 错误代码可以是存储在 `wCode` 结构的 `EXCEPINFO` 成员中的 16 位值，也可以是存储在 `scode` 结构的 `EXCEPINFO` 成员中的完整 32 位值。 如果返回了 16 位 `wCode`，必须先将其映射到 32 位失败 `HRESULT`。

结束 Microsoft 专用

另请参阅

_com_error::HRESULTToWCode
_com_error::WCodeToHRESULT
_com_error 类

Last updated on 2025/11/07

`_com_error::WCodeToHRESULT`

Microsoft 专用

将 16 位 `wCode` 映射到 32 位 `HRESULT`。

语法

C++

```
static HRESULT WCodeToHRESULT(
   WORD wCode
) throw();
```

参数

`wCode`
要映射为 32 位 `wCode` 的 16 位 `HRESULT`。

返回值

已从 16 位 `HRESULT` 映射的 32 位 `wCode`。

注解

请参阅 WCode 成员函数。

结束 Microsoft 专用

另请参阅

_com_error::WCode
_com_error::HRESULTToWCode
_com_error 类

Last updated on 2025/11/07

`_com_error` 运算符

有关 `_com_error` 运算符的信息，请参阅 _com_error 类。

另请参阅

_com_error 类

Last updated on 2025/11/07

`_com_error::operator=`

Microsoft 专用

将现有 `_com_error` 对象赋给另一个对象。

语法

C++

```
_com_error& operator=(
   const _com_error& that
) throw ();
```

参数

`that`

`_com_error` 对象。

结束 Microsoft 专用

另请参阅

_com_error 类

Last updated on 2025/11/07

_com_ptr_t 类

Microsoft 专用

_com_ptr_t 对象封装 COM 接口指针，被称为“智能”指针。 此模板类通过对 `IUnknown` 成员函数的函数调用来管理资源分配和解除分配：`QueryInterface`、`AddRef` 和 `Release`。

智能指针通常由 _COM_SMARTPTR_TYPEDEF 宏提供的 typedef 定义引用。 此宏采用接口名称和 IID，并利用接口名称与后缀 声明 _com_ptr_t`Ptr` 的专用化。 例如：

C++

```
_COM_SMARTPTR_TYPEDEF(IMyInterface, __uuidof(IMyInterface));
```

声明 _com_ptr_t 专用化 `IMyInterfacePtr`。

函数模板集（而非此模板类的成员），支持与比较运算符右侧的智能指针进行比较。

建筑

展开表

名称	描述
_com_ptr_t	构造 _com_ptr_t 对象。

低级别运算

展开表

名称	描述
AddRef	调用封装的接口指针上的 `AddRef` 的 `IUnknown` 成员函数。
附加	封装此智能指针的类型的原始接口指针。
CreateInstance	创建给定了 `CLSID` 或 `ProgID` 的对象的新实例。
分离	提取并返回封装的接口指针。
GetActiveObject	附加到给定 `CLSID` 或 `ProgID` 的对象的现有实例。
GetInterfacePtr	返回封装的接口指针。
QueryInterface	调用封装的接口指针上的 `QueryInterface` 的 `IUnknown` 成员函数。

名称	描述
版本	调用封装的接口指针上的 `Release` 的 `IUnknown` 成员函数。

运算符

展开表

名称	描述
operator =	将新值赋给现有 _com_ptr_t 对象。
运算符 ==、！ =、<、、<>=、>=	将智能指针对象与另一个智能指针、原始接口指针或 NULL 进行比较。
提取器	提取封装的 COM 接口指针。

结束 Microsoft 专用

要求

标头：comip.h<>

Lib：comsuppw.lib 或 comsuppwd.lib（有关详细信息，请参阅 /Zc:wchar_t（wchar_t 为本机类型））

另请参阅

编译器 COM 支持类

Last updated on 2025/11/07

_com_ptr_t 成员函数

有关 _com_ptr_t 成员函数的信息，请参阅 _com_ptr_t 类。

另请参阅

_com_ptr_t 类

Last updated on 2025/11/07

_com_ptr_t::_com_ptr_t

Microsoft 专用

构造 _com_ptr_t 对象。

语法

C++

```
// Default constructor.
// Constructs a NULL smart pointer.
_com_ptr_t() throw();

// Constructs a NULL smart pointer. The NULL argument must be zero.
_com_ptr_t(
   int null
);

// Constructs a smart pointer as a copy of another instance of the
// same smart pointer. AddRef is called to increment the reference
// count for the encapsulated interface pointer.
_com_ptr_t(
   const _com_ptr_t& cp
) throw();

// Move constructor (Visual Studio 2015 Update 3 and later)
_com_ptr_t(_com_ptr_t&& cp) throw();

// Constructs a smart pointer from a raw interface pointer of this
// smart pointer's type. If fAddRef is true, AddRef is called
// to increment the reference count for the encapsulated
// interface pointer. If fAddRef is false, this constructor
// takes ownership of the raw interface pointer without calling AddRef.
_com_ptr_t(
   Interface* pInterface,
   bool fAddRef
) throw();

// Construct pointer for a _variant_t object.
// Constructs a smart pointer from a _variant_t object. The
// encapsulated VARIANT must be of type VT_DISPATCH or VT_UNKNOWN, or
// it can be converted into one of these two types. If QueryInterface
// fails with an E_NOINTERFACE error, a NULL smart pointer is
// constructed.
_com_ptr_t(
   const _variant_t& varSrc
);

// Constructs a smart pointer given the CLSID of a coclass. This
// function calls CoCreateInstance, by the member function
```

```
//  CreateInstance, to create a new COM object and then queries for
// this smart pointer's interface type. If QueryInterface fails with
// an E_NOINTERFACE error, a NULL smart pointer is constructed.
explicit _com_ptr_t(
   const CLSID& clsid,
   IUnknown* pOuter = NULL,
   DWORD dwClsContext = CLSCTX_ALL
);

// Calls CoCreateClass with provided CLSID retrieved from string.
explicit _com_ptr_t(
   LPCWSTR str,
   IUnknown* pOuter = NULL,
   DWORD dwClsContext = CLSCTX_ALL
);

// Constructs a smart pointer given a multibyte character string that
// holds either a CLSID (starting with "{") or a ProgID. This function
// calls CoCreateInstance, by the member function CreateInstance, to
// create a new COM object and then queries for this smart pointer's
// interface type. If QueryInterface fails with an E_NOINTERFACE error,
// a NULL smart pointer is constructed.
explicit _com_ptr_t(
   LPCSTR str,
   IUnknown* pOuter = NULL,
   DWORD dwClsContext = CLSCTX_ALL
);

// Saves the interface.
template<>
_com_ptr_t(
   Interface* pInterface
) throw();

// Make sure correct ctor is called
template<>
_com_ptr_t(
   LPSTR str
);

// Make sure correct ctor is called
template<>
_com_ptr_t(
   LPWSTR str
);

// Constructs a smart pointer from a different smart pointer type or
// from a different raw interface pointer. QueryInterface is called to
// find an interface pointer of this smart pointer's type. If
// QueryInterface fails with an E_NOINTERFACE error, a NULL smart
// pointer is constructed.
template<typename _OtherIID>
_com_ptr_t(
   const _com_ptr_t<_OtherIID>& p
);
```

```
// Constructs a smart-pointer from any IUnknown-based interface pointer.
template<typename _InterfaceType>
_com_ptr_t(
   _InterfaceType* p
);

// Disable conversion using _com_ptr_t* specialization of
// template<typename _InterfaceType> _com_ptr_t(_InterfaceType* p)
template<>
explicit _com_ptr_t(
   _com_ptr_t* p
);
```

参数

pInterface
原始接口指针。

fAddRef
如果为 `true`，则调用 `AddRef` 来增加封装的接口指针的引用计数。

cp
_com_ptr_t 对象。

p
一个原始接口指针，其类型与此 _com_ptr_t 对象的智能指针类型不同。

varSrc
`_variant_t` 对象。

clsid
组件类的 `CLSID`。

dwClsContext
运行可执行代码的上下文。

lpcStr
一个包含 `CLSID` （以“{”开头）或 的多字节字符串`ProgID`。

pOuter
聚合未知的外部对象。

结束 Microsoft 专用

另请参阅

_com_ptr_t 类

Last updated on 2025/11/07

_com_ptr_t::AddRef

Microsoft 专用

调用封装的接口指针上的 `AddRef` 的 `IUnknown` 成员函数。

语法

C++

```
void AddRef( );
```

备注

如果封装的接口指针为 NULL，则在该指针上调用 `IUnknown::AddRef` 会引发 `E_POINTER` 错误。

结束 Microsoft 专用

另请参阅

_com_ptr_t 类

Last updated on 2025/11/07

_com_ptr_t::Attach

Microsoft 专用

封装此智能指针的类型的原始接口指针。

语法

C++

```
void Attach( Interface* pInterface ) throw( );
void Attach( Interface* pInterface, bool fAddRef ) throw( );
```

参数

pInterface
原始接口指针。

fAddRef
如果是 `true`，则调用 `AddRef`。 如果是 `false`，则 `_com_ptr_t` 对象将拥有原始接口指针的所有权，而不调用 `AddRef`。

注解

- Attach(pInterface) 未被调用`AddRef`。 将接口的所有权传递给此 `_com_ptr_t` 对象。 调用 `Release` 以减少前面封装的指针的引用计数。
- Attach(pInterface, fAddRef) 如果 fAddRef 为 ， 则调用 来增加封装的接口指针的引用计数 `true` `AddRef`。 如果 fAddRef 为 ， 则 `false` 对象将拥有原始接口指针的所有权，而不调用 `_com_ptr_t` `AddRef`。 调用 `Release` 以减少前面封装的指针的引用计数。

结束 Microsoft 专用

另请参阅

_com_ptr_t 类

Last updated on 2025/11/07

_com_ptr_t::CreateInstance

Microsoft 专用

创建给定了 `CLSID` 或 `ProgID` 的对象的新实例。

语法

```
HRESULT CreateInstance(
   const CLSID& rclsid,
   IUnknown* pOuter=NULL,
   DWORD dwClsContext = CLSCTX_ALL
) throw( );
HRESULT CreateInstance(
   LPCWSTR clsidString,
   IUnknown* pOuter=NULL,
   DWORD dwClsContext = CLSCTX_ALL
) throw( );
HRESULT CreateInstance(
   LPCSTR clsidStringA,
   IUnknown* pOuter=NULL,
   DWORD dwClsContext = CLSCTX_ALL
) throw( );
```

参数

rclsid
对象的 `CLSID`。

clsidString
一个包含（以“{`CLSID`”开头的） 或 `ProgID` 的 Unicode 字符串。

clsidStringA
使用 ANSI 代码页并包括（以 "`CLSID`" 开头）的 或 `ProgID` 的多字节字符串。

dwClsContext
运行可执行代码的上下文。

pOuter
聚合未知的外部对象。

备注

这些成员函数调用 `CoCreateInstance` 来创建新的 COM 对象，然后查询此智能指针的接口类型。 生成的指针随后将封装在此 `_com_ptr_t` 对象内。 调用 `Release` 以减少前面封装的指针的引用计数。 此例程返回 HRESULT 以指示成功或失败。

- **CreateInstance**(*rclsid*,*dwClsContext*) 根据给定的 `CLSID` 创建对象的新运行实例。
- **CreateInstance**(*clsidString*,*dwClsContext*) 根据给定的包含 `CLSID` （以“{”开头）或 `ProgID` 的 Unicode 字符串来创建对象的新运行实例。
- **CreateInstance**(*clsidStringA*,*dwClsContext*) 根据给定的包含 `CLSID` （以“{”开头）或 `ProgID` 的多字节字符串来创建对象的新运行实例。 调用 MultiByteToWideChar，假定字符串是在 ANSI 代码页中而不是 OEM 代码页中。

结束 Microsoft 专用

另请参阅

_com_ptr_t 类

Last updated on 2025/11/07

_com_ptr_t::Detach

Microsoft 专用

提取并返回封装的接口指针。

语法

```
Interface* Detach( ) throw( );
```

备注

提取并返回封装的接口指针，然后将封装的指针存储清除为 NULL。 这将从封装中移除接口指针。 由你在返回的接口指针上调用 `Release`。

结束 Microsoft 专用

另请参阅

_com_ptr_t 类

Last updated on 2025/11/07

_com_ptr_t::GetActiveObject

Microsoft 专用

附加到给定 `CLSID` 或 `ProgID` 的对象的现有实例。

语法

```
HRESULT GetActiveObject(
   const CLSID& rclsid
) throw( );
HRESULT GetActiveObject(
   LPCWSTR clsidString
) throw( );
HRESULT GetActiveObject(
   LPCSTR clsidStringA
) throw( );
```

参数

rclsid
对象的 `CLSID`。

clsidString
一个包含（以"{`CLSID`"开头的） 或 `ProgID` 的 Unicode 字符串。

clsidStringA
使用 ANSI 代码页并包括（以 "`CLSID`" 开头）的 或 `ProgID` 的多字节字符串。

备注

这些成员函数调用 **GetActiveObject** 来检索指向已向 OLE 注册的正在运行对象的指针，然后查询此智能指针的接口类型。 生成的指针随后将封装在此 `_com_ptr_t` 对象内。 调用 `Release` 以减少前面封装的指针的引用计数。 此例程返回 HRESULT 以指示成功或失败。

- GetActiveObject(`rclsid`)附加到给定 `CLSID` 的对象的现有实例。
- GetActiveObject(`clsidString`) 附加到包含 `CLSID` （以 "{" 开头）或 `ProgID` 的给定 Unicode 字符串的对象的现有实例。

- GetActiveObject(`clsidStringA`) 附加到包含（以 "{`CLSID`" 开头）的 或 `ProgID` 的给定多字节字符串的对象的现有实例。 调用 MultiByteToWideChar，假定字符串是在 ANSI 代码页中而不是 OEM 代码页中。

结束 Microsoft 专用

另请参阅

_com_ptr_t 类

Last updated on 2025/11/07

_com_ptr_t::GetInterfacePtr

Microsoft 专用

返回封装的接口指针。

语法

```
Interface* GetInterfacePtr( ) const throw( );
Interface*& GetInterfacePtr() throw();
```

备注

返回封装的接口指针，这可能是 NULL。

结束 Microsoft 专用

另请参阅

_com_ptr_t 类

Last updated on 2025/11/07

_com_ptr_t::QueryInterface

Microsoft 专用

调用封装的接口指针上的 的 QueryInterface 成员函数`IUnknown`。

语法

```
template<typename _InterfaceType> HRESULT QueryInterface (
   const IID& iid,
   _InterfaceType*& p
) throw ( );
template<typename _InterfaceType> HRESULT QueryInterface (
   const IID& iid,
   _InterfaceType** p
) throw( );
```

参数

iid
接口指针的 `IID`。

p
原始接口指针。

备注

对具有指定的 `IUnknown::QueryInterface` 的已封装接口指针调用 `IID` 并返回 p 中生成的原始接口指针。 此例程返回 HRESULT 以指示成功或失败。

结束 Microsoft 专用

另请参阅

_com_ptr_t 类

Last updated on 2025/11/07

_com_ptr_t::Release

Microsoft 专用

调用封装的接口指针上的 的 Release 成员函数`IUnknown`。

语法

C++

```
void Release( );
```

备注

对封装的接口指针调用 `IUnknown::Release`，如果此接口指针为 NULL，则将引发 `E_POINTER` 错误。

结束 Microsoft 专用

另请参阅

_com_ptr_t 类

Last updated on 2025/11/07

_com_ptr_t 运算符

有关 `_com_ptr_t` 运算符的信息，请参阅 _com_ptr_t 类。

另请参阅

_com_ptr_t 类

Last updated on 2025/11/07

_com_ptr_t：： operator =

Microsoft 专用

将新值赋给现有 `_com_ptr_t` 对象。

语法

```
template<typename _OtherIID>
_com_ptr_t& operator=( const _com_ptr_t<_OtherIID>& p );

// Sets a smart pointer to be a different smart pointer of a different
// type or a different raw interface pointer. QueryInterface is called
// to find an interface pointer of this smart pointer's type, and
// Release is called to decrement the reference count for the previously
// encapsulated pointer. If QueryInterface fails with an E_NOINTERFACE,
// a NULL smart pointer results.
template<typename _InterfaceType>
_com_ptr_t& operator=(_InterfaceType* p );

// Encapsulates a raw interface pointer of this smart pointer's type.
// AddRef is called to increment the reference count for the encapsulated
// interface pointer, and Release is called to decrement the reference
// count for the previously encapsulated pointer.
template<> _com_ptr_t&
operator=( Interface* pInterface ) throw();

// Sets a smart pointer to be a copy of another instance of the same
// smart pointer of the same type. AddRef is called to increment the
// reference count for the encapsulated interface pointer, and Release
// is called to decrement the reference count for the previously
// encapsulated pointer.
_com_ptr_t& operator=( const _com_ptr_t& cp ) throw();

// Sets a smart pointer to NULL. The NULL argument must be a zero.
_com_ptr_t& operator=( int null );

// Sets a smart pointer to be a _variant_t object. The encapsulated
// VARIANT must be of type VT_DISPATCH or VT_UNKNOWN, or it can be
// converted to one of these two types. If QueryInterface fails with an
// E_NOINTERFACE error, a NULL smart pointer results.
_com_ptr_t& operator=( const _variant_t& varSrc );
```

备注

将接口指针分配给此 `_com_ptr_t` 对象。

结束 Microsoft 专用

另请参阅

_com_ptr_t 类

Last updated on 2025/11/07

_com_ptr_t 关系运算符

Microsoft 专用

将智能指针对象与另一个智能指针、原始接口指针或 NULL 进行比较。

语法

```
template<typename _OtherIID>
bool operator==( const _com_ptr_t<_OtherIID>& p );

template<typename _OtherIID>
bool operator==( _com_ptr_t<_OtherIID>& p );

template<typename _InterfaceType>
bool operator==( _InterfaceType* p );

template<>
bool operator==( Interface* p );

template<>
bool operator==( const _com_ptr_t& p ) throw();

template<>
bool operator==( _com_ptr_t& p ) throw();

bool operator==( int null );

template<typename _OtherIID>
bool operator!=( const _com_ptr_t<_OtherIID>& p );

template<typename _OtherIID>
bool operator!=( _com_ptr_t<_OtherIID>& p );

template<typename _InterfaceType>
bool operator!=( _InterfaceType* p );

bool operator!=( int null );

template<typename _OtherIID>
bool operator<( const _com_ptr_t<_OtherIID>& p );

template<typename _OtherIID>
bool operator<( _com_ptr_t<_OtherIID>& p );

template<typename _InterfaceType>
bool operator<( _InterfaceType* p );

template<typename _OtherIID>
```

```
bool operator>( const _com_ptr_t<_OtherIID>& p );

template<typename _OtherIID>
bool operator>(_com_ptr_t< _OtherIID>& p );

template<typename _InterfaceType>
bool operator>( _InterfaceType* p );

template<typename _OtherIID>
bool operator<=( const _com_ptr_t<_OtherIID>& p );

template<typename _OtherIID>
bool operator<=( _com_ptr_t<_OtherIID>& p );

template<typename _InterfaceType>
bool operator<=( _InterfaceType* p );

template<typename _OtherIID>
bool operator>=( const _com_ptr_t<_OtherIID>& p );

template<typename _OtherIID>
bool operator>=( _com_ptr_t<_OtherIID>& p );

template<typename _InterfaceType>
bool operator>=( _InterfaceType* p );
```

备注

将智能指针对象与另一个智能指针、原始接口指针或 NULL 比较。 除 NULL 指针测试外，这些运算符首先查询 `IUnknown` 的两个指针，然后比较其结果。

结束 Microsoft 专用

另请参阅

_com_ptr_t 类

Last updated on 2025/11/07

`_com_ptr_t` 提取器

Microsoft 专用

提取封装的 COM 接口指针。

语法

C++

```
operator Interface*( ) const throw( );
operator Interface&( ) const;
Interface& operator*( ) const;
Interface* operator->( ) const;
Interface** operator&( ) throw( );
operator bool( ) const throw( );
```

备注

- `operator Interface*`：返回封装的接口指针，这可能是 NULL。
- `operator Interface&` 返回对封装接口指针的引用，如果指针为 NULL，则抛出错误。
- `operator*`：允许智能指针对象在取消引用时充当实际封装的接口。
- `operator->`：允许智能指针对象在取消引用时充当实际封装的接口。
- `operator&`：释放任何封装的接口指针（将其替换为 NULL），并返回封装的指针的地址。此运算符允许您通过地址将智能指针传递给一个具有 out 参数的函数，该函数通过该参数返回接口指针。
- `operator bool`：支持在条件表达式中使用智能指针对象。 如果该指针不为 NULL，则此运算符返回 `true`。

> ⓘ **注意**
>
> 由于 `operator bool` 未声明为 `explicit`，因此 `_com_ptr_t` 可隐式转换为 `bool`，它可转换为任何标量类型。 这可能会在代码中产生意外的后果。 请启用**编译器警告（级别 4）C4800** 来防止无意中使用此转换。

另请参阅

_com_ptr_t 类

Last updated on 2025/11/07

关系函数模板

Microsoft 专用

语法

```
template<typename _InterfaceType> bool operator==(
   int NULL,
   _com_ptr_t<_InterfaceType>& p
);
template<typename _Interface,
   typename _InterfacePtr> bool operator==(
   _Interface* i,
   _com_ptr_t<_InterfacePtr>& p
);
template<typename _Interface> bool operator!=(
   int NULL,
   _com_ptr_t<_Interface>& p
);
template<typename _Interface,
   typename _InterfacePtr> bool operator!=(
   _Interface* i,
   _com_ptr_t<_InterfacePtr>& p
);
template<typename _Interface> bool operator<(
   int NULL,
   _com_ptr_t<_Interface>& p
);
template<typename _Interface,
   typename _InterfacePtr> bool operator<(
   _Interface* i,
   _com_ptr_t<_InterfacePtr>& p
);
template<typename _Interface> bool operator>(
   int NULL,
   _com_ptr_t<_Interface>& p
);
template<typename _Interface,
   typename _InterfacePtr> bool operator>(
   _Interface* i,
   _com_ptr_t<_InterfacePtr>& p
);
template<typename _Interface> bool operator<=(
   int NULL,
   _com_ptr_t<_Interface>& p
);
template<typename _Interface,
   typename _InterfacePtr> bool operator<=(
   _Interface* i,
   _com_ptr_t<_InterfacePtr>& p
```

```
);
template<typename _Interface> bool operator>=(
   int NULL,
   _com_ptr_t<_Interface>& p
);
template<typename _Interface,
   typename _InterfacePtr> bool operator>=(
   _Interface* i,
   _com_ptr_t<_InterfacePtr>& p
);
```

参数

i
原始接口指针。

p
智能指针。

备注

这些函数模板允许与比较运算符右侧的智能指针的比较。 这些函数不是 `_com_ptr_t` 的成员函数。

结束 Microsoft 专用

另请参阅

_com_ptr_t 类

Last updated on 2025/11/07

_variant_t 类

Microsoft 专用

_variant_t 对象封装 `VARIANT` 数据类型。 该类管理资源的分配和解除分配，并根据需要对 `VariantInit` 和 `VariantClear` 进行函数调用。

建筑

展开表

名称	描述
_variant_t	构造 _variant_t 对象。

Operations

展开表

名称	描述
附加	将 `VARIANT` 对象附加到 _variant_t 对象。
清除	清除封装的 `VARIANT` 对象。
ChangeType	将 _variant_t 对象的类型更改为指示的 `VARTYPE`。
分离	从该 _variant_t `VARIANT` 对象中分离已封装的 对象。
SetString	将字符串分配给此 _variant_t 对象。

运算符

展开表

名称	描述
运算符 =	将新值赋给现有 _variant_t 对象。
operator ==， ！ =	比较两个 _variant_t 对象是否相等。
提取器	从封装的 `VARIANT` 对象中提取数据。

结束 Microsoft 专用

要求

标头：<comutil.h>

Lib：comsuppw.lib 或 comsuppwd.lib（有关详细信息，请参阅 /Zc:wchar_t（wchar_t 为本机类型））

另请参阅

编译器 COM 支持类

Last updated on 2025/11/07

_variant_t 成员函数

有关 _variant_t 成员函数的信息，请参阅 _variant_t 类。

另请参阅

_variant_t 类

Last updated on 2025/11/07

`_variant_t::_variant_t`

Microsoft 专用

构造 `_variant_t` 对象。

语法

C++

```
_variant_t( ) throw( );

_variant_t(
   const VARIANT& varSrc
);

_variant_t(
   const VARIANT* pVarSrc
);

_variant_t(
   const _variant_t& var_t_Src
);

_variant_t(
   VARIANT& varSrc,
   bool fCopy
);

_variant_t(
   short sSrc,
   VARTYPE vtSrc = VT_I2
);

_variant_t(
   long lSrc,
   VARTYPE vtSrc = VT_I4
);

_variant_t(
   float fltSrc
) throw( );

_variant_t(
   double dblSrc,
   VARTYPE vtSrc = VT_R8
);

_variant_t(
   const CY& cySrc
) throw( );
```

```
_variant_t(
   const _bstr_t& bstrSrc
);

_variant_t(
   const wchar_t *wstrSrc
);

_variant_t(
   const char* strSrc
);

_variant_t(
   IDispatch* pDispSrc,
   bool fAddRef = true
) throw( );

_variant_t(
   bool bSrc
) throw( );

_variant_t(
   IUnknown* pIUknownSrc,
   bool fAddRef = true
) throw( );

_variant_t(
   const DECIMAL& decSrc
) throw( );

_variant_t(
   BYTE bSrc
) throw( );

variant_t(
   char cSrc
) throw();

_variant_t(
   unsigned short usSrc
) throw();

_variant_t(
   unsigned long ulSrc
) throw();

_variant_t(
   int iSrc
) throw();

_variant_t(
   unsigned int uiSrc
) throw();
```

```
_variant_t(
    __int64 i8Src
) throw();

_variant_t(
    unsigned __int64 ui8Src
) throw();
```

参数

varSrc

要复制到新的 `VARIANT` 对象中的 `_variant_t` 对象。

pVarSrc

指示要复制到新的 `VARIANT` 对象中的 `_variant_t` 对象的指针。

var_t_Src

要复制到新的 `_variant_t` 对象中的 `_variant_t` 对象。

fCopy

如果为 **`false`**，则将提供的 `VARIANT` 对象附加到新的 `_variant_t` 对象，而不使用 `VariantCopy` 创建新的副本。

%>

要复制到新的 `_variant_t` 对象中的整数值。

vtSrc

新 `VARTYPE` 对象的 `_variant_t`。

%>

要复制到新的 `_variant_t` 对象中的数值。

cySrc

要复制到新的 `CY` 对象中的 `_variant_t` 对象。

bstrSrc

要复制到新的 `_bstr_t` 对象中的 `_variant_t` 对象。

%>

要复制到新的 `_variant_t` 对象中的字符串。

bSrc

要复制到新的 **`bool`** 对象中的 `_variant_t` 值。

pIUknownSrc
指向要封装到新的 `_variant_t` 对象中的 VT_UNKNOWN 对象的 COM 接口指针。

pDispSrc
指向要封装到新的 `_variant_t` 对象中的 VT_DISPATCH 对象的 COM 接口指针。

decSrc
要复制到新的 `DECIMAL` 对象中的 `_variant_t` 值。

bSrc
要复制到新的 `BYTE` 对象中的 `_variant_t` 值。

cSrc
要复制到新的 **`char`** 对象中的 `_variant_t` 值。

usSrc
要复制到新的 **`unsigned short`** 对象中的 `_variant_t` 值。

ulSrc
要复制到新的 **`unsigned long`** 对象中的 `_variant_t` 值。

iSrc
要复制到新的 **`int`** 对象中的 `_variant_t` 值。

uiSrc
要复制到新的 **`unsigned int`** 对象中的 `_variant_t` 值。

i8Src
要复制到新的 **`__int64`** 对象中的 `_variant_t` 值。

ui8Src
要复制到新的 **`unsigned __int64`** 对象中的 `_variant_t` 值。

备注

- **`_variant_t()`** 构造一个空的 `_variant_t` 对象 `VT_EMPTY`。
- **`_variant_t( VARIANT& varSrc )`** 基于 `_variant_t` 对象的副本构造一个 `VARIANT` 对象。 变体类型将保留。
- **`_variant_t( VARIANT* pVarSrc )`** 基于 `_variant_t` 对象的副本构造一个 `VARIANT` 对象。 变体类型将保留。

- `_variant_t( _variant_t& var_t_Src )` 基于另一个 `_variant_t` 对象构造一个 `_variant_t` 对象。 变体类型将保留。

- `_variant_t( VARIANT& varSrc, bool fCopy )` 基于现有 `_variant_t` 对象构造一个 `VARIANT` 对象。 如果 *`fCopy`* 为 `false`，则 `VARIANT` 对象将附加到新对象中，而无需创建副本。

- `_variant_t( short sSrc, VARTYPE vtSrc = VT_I2 )` 基于 `_variant_t` 整数值构造一个 `VT_I2` 或 `VT_BOOL` 类型的 `short` 对象。 任何其他 `VARTYPE` 会导致 `E_INVALIDARG` 错误。

- `_variant_t( long lSrc, VARTYPE vtSrc = VT_I4 )` 基于 `_variant_t` 整数值构造一个 `VT_I4`、 `VT_BOOL` 或 `VT_ERROR` 类型的 `long` 对象。 任何其他 `VARTYPE` 会导致 `E_INVALIDARG` 错误。

- `_variant_t( float fltSrc )` 基于 `_variant_t` 数值构造一个 `VT_R4` 类型的 `float` 对象。

- `_variant_t( double dblSrc, VARTYPE vtSrc = VT_R8 )` 基于 `_variant_t` 数值构造一个 `VT_R8` 或 `VT_DATE` 类型的 `double` 对象。 任何其他 `VARTYPE` 会导致 `E_INVALIDARG` 错误。

- `_variant_t( CY& cySrc )` 基于 `_variant_t` 对象构造一个 `VT_CY` 类型的 `CY` 对象。

- *`_variant_t( _bstr_t& bstrSrc )`* 基于 `_variant_t` 对象构造一个 `VT_BSTR` 类型的 `_bstr_t` 对象。 分配新的 `BSTR`。

- `_variant_t( wchar_t* wstrSrc )` 基于 Unicode 字符串构造一个 `_variant_t` 类型的 `VT_BSTR` 对象。 分配新的 `BSTR`。

- `_variant_t( char* strSrc )` 基于字符串构造一个 `_variant_t` 类型的 `VT_BSTR` 对象。 分配新的 `BSTR`。

- `_variant_t( bool bSrc )` 基于 `_variant_t` 值构造一个 `VT_BOOL` 类型的 `bool` 对象。

- `_variant_t( IUnknown* pIUknownSrc, bool fAddRef = true )` 从 COM 接口指针构造 `_variant_t` 类型的 `VT_UNKNOWN` 对象。 如果 `fAddRef` 为 `true`，则在提供的接口指针上调用 `AddRef` 以匹配将在 `Release` 对象被销毁时发生的对 `_variant_t` 的调用。 由你来在提供的接口指针上调用 `Release`。 如果 `fAddRef` 为 `false`，则此构造函数将获得提供的接口指针的所有权；不在提供的接口指针上调用 `Release`。

- `_variant_t( IDispatch* pDispSrc, bool fAddRef = true )` 从 COM 接口指针构造 `_variant_t` 类型的 `VT_DISPATCH` 对象。 如果 `fAddRef` 为 `true`，则在提供的接口指针上调用 `AddRef` 以匹配将在 `Release` 对象被销毁时发生的对 `_variant_t` 的调用。 由你来在提供的接口指针上调用 `Release`。 如果 `fAddRef` 为 `false`，则此构造函数将获得提供的接口指针的所有权；不在提供的接口指针上调用 `Release`。

- `_variant_t( DECIMAL& decSrc )` 基于 `_variant_t` 值构造一个 `VT_DECIMAL` 类型的 `DECIMAL` 对象。
- `_variant_t( BYTE bSrc )` 基于 `_variant_t` 值构造一个 `VT_UI1` 类型的 `BYTE` 对象。

结束 Microsoft 专用

另请参阅

_variant_t 类

Last updated on 2025/11/07

_variant_t::Attach

Microsoft 专用

将 `VARIANT` 对象附加到 _variant_t 对象。

语法

```
C++
void Attach(VARIANT& varSrc);
```

参数

varSrc
要附加到此 _variant_t 对象中的 `VARIANT` 对象。

备注

通过封装取得 `VARIANT` 的所有权。 此成员函数将释放所有现有封装的 `VARIANT`，然后复制提供的 `VARIANT`，并将其 `VARTYPE` 设置为 VT_EMPTY 以确保其资源只能由 _variant_t 析构函数释放。

结束 Microsoft 专用

另请参阅

_variant_t 类

Last updated on 2025/11/07

_variant_t::Clear

Microsoft 专用

清除封装的 `VARIANT` 对象。

语法

```
C++
void Clear( );
```

备注

对封装的 `VariantClear` 对象调用 `VARIANT`。

结束 Microsoft 专用

另请参阅

_variant_t 类

Last updated on 2025/11/07

_variant_t::ChangeType

Microsoft 专用

将 `_variant_t` 对象的类型更改为指示的 `VARTYPE`。

语法

C++

```
void ChangeType(
   VARTYPE vartype,
   const _variant_t* pSrc = NULL
);
```

参数

vartype
`VARTYPE` 对象的 `_variant_t`。

pSrc
指向要转换的 `_variant_t` 对象的指针。 如果此值为 NULL，则转换就地执行。

注解

此成员函数将 `_variant_t` 对象转换为指示的 `VARTYPE`。 如果 pSrc 为 NULL，转换将就地执行，否则，将从 pSrc 复制此 对象，然后进行转换`_variant_t`。

结束 Microsoft 专用

另请参阅

_variant_t 类

Last updated on 2025/11/07

_variant_t::Detach

Microsoft 专用

从此 `VARIANT` 对象中分离已封装的 `_variant_t` 对象。

语法

```
VARIANT Detach( );
```

返回值

封装的 `VARIANT`。

备注

提取和返回封装的 `VARIANT`，然后清除此 `_variant_t` 对象而不销毁它。 此成员函数从封装中移除 `VARIANT`，并将此 `VARTYPE` 对象的 `_variant_t` 设置为 VT_EMPTY。 你可以自己决定是否要通过调用 `VARIANT` 函数来释放返回的 。

结束 Microsoft 专用

另请参阅

_variant_t 类

Last updated on 2025/11/07

_variant_t::SetString

Microsoft 专用

将字符串分配给此 `_variant_t` 对象。

语法

C++

```
void SetString(const char* pSrc);
```

参数

pSrc
指向字符串的指针。

备注

将 ANSI 字符串转换为 Unicode `BSTR` 字符串并将其分配给此 `_variant_t` 对象。

结束 Microsoft 专用

另请参阅

_variant_t 类

Last updated on 2025/11/07

_variant_t 运算符

有关“_variant_t”运算符的详细信息，请参阅 **_variant_t 类。**

另请参阅

_variant_t 类

Last updated on 2025/11/07

`_variant_t::operator=`

为 `_variant_t` 实例分配新值。

`_variant_t` 类及其 `operator=` 成员特定于 Microsoft。

语法

C++

```
_variant_t& operator=( const VARIANT& varSrc );
_variant_t& operator=( const VARIANT* pVarSrc );
_variant_t& operator=( const _variant_t& var_t_Src );
_variant_t& operator=( short sSrc );
_variant_t& operator=( long lSrc );
_variant_t& operator=( float fltSrc );
_variant_t& operator=( double dblSrc );
_variant_t& operator=( const CY& cySrc );
_variant_t& operator=( const _bstr_t& bstrSrc );
_variant_t& operator=( const wchar_t* wstrSrc );
_variant_t& operator=( const char* strSrc );
_variant_t& operator=( IDispatch* pDispSrc );
_variant_t& operator=( bool bSrc );
_variant_t& operator=( IUnknown* pSrc );
_variant_t& operator=( const DECIMAL& decSrc );
_variant_t& operator=( BYTE byteSrc );
_variant_t& operator=( char cSrc );
_variant_t& operator=( unsigned short usSrc );
_variant_t& operator=( unsigned long ulSrc );
_variant_t& operator=( int iSrc );
_variant_t& operator=( unsigned int uiSrc );
_variant_t& operator=( __int64 i8Src );
_variant_t& operator=( unsigned __int64 ui8Src );
```

参数

`varSrc`

对要从中复制内容和 `VARIANT` 类型的 `VT_*` 的引用。

`pVarSrc`

指向要从中复制内容的 `VARIANT` 和 `VT_*` 类型的指针。

`var_t_Src`

对要从中复制内容和 `_variant_t` 类型的 `VT_*` 的引用。

sSrc
要复制的 **short** 整数值。 如果 VT_BOOL 属于 *this 类型，则给定类型 VT_BOOL。 否则，给定类型 VT_I2。

lSrc
要复制的 **long** 整数值。 如果 VT_BOOL 属于 *this 类型，则给定类型 VT_BOOL。 如果 VT_ERROR 属于 *this 类型，则给定类型 VT_ERROR。 否则，给定类型 VT_I4。

fltSrc
要复制的 **float** 数值。 给定类型 VT_R4。

dblSrc
要复制的 **double** 数值。 如果 VT_DATE 属于 this 类型，则给定类型 VT_DATE。 否则，给定类型 VT_R8。

cySrc
要复制的 CY 对象。 给定类型 VT_CY。

bstrSrc
要复制的 BSTR 对象。 给定类型 VT_BSTR。

wstrSrc
要复制的 Unicode 字符串，存储为 BSTR 并给定类型 VT_BSTR。

strSrc
要复制的多字节字符串，存储为 BSTR 并给定类型 VT_BSTR。

pDispSrc
要通过调用 IDispatch 复制的 AddRef 指针。 给定类型 VT_DISPATCH。

bSrc
要复制的 **bool** 值。 给定类型 VT_BOOL。

pSrc
要通过调用 IUnknown 复制的 AddRef 指针。 给定类型 VT_UNKNOWN。

decSrc
要复制的 DECIMAL 对象。 给定类型 VT_DECIMAL。

byteSrc
要复制的 BYTE 值。 给定类型 VT_UI1。

cSrc
要复制的 **char** 值。 给定类型 VT_I1。

usSrc
要复制的 **unsigned short** 值。 给定类型 VT_UI2。

ulSrc
要复制的 **unsigned long** 值。 给定类型 VT_UI4。

iSrc
要复制的 **int** 值。 给定类型 VT_INT。

uiSrc
要复制的 **unsigned int** 值。 给定类型 VT_UINT。

i8Src
要复制的 **__int64** 或 **long long** 值。 给定类型 VT_I8。

ui8Src
要复制的 **unsigned __int64** 或 **unsigned long long** 值。 给定类型 VT_UI8。

备注

operator= 赋值运算符清除任何现有值，这会删除对象类型，或为 Release 和 IDispatch* 类型调用 IUnknown*。 然后，它将一个新值复制到 _variant_t 对象中。 它会更改 _variant_t 类型以匹配分配的值，除非对 **short**、**long** 和 **double** 参数进行了说明。 直接复制值类型。 VARIANT 或 _variant_t 指针或引用参数复制分配对象的内容和类型。 其他指针或引用类型参数创建分配对象的副本。 赋值运算符为 AddRef 和 IDispatch* 参数调用 IUnknown*。

如果发生错误， operator= 会调用 _com_raise_error。

operator= 返回对更新的 _variant_t 对象的引用。

另请参阅

_variant_t 类

Last updated on 2025/11/07

_variant_t 关系运算符

Microsoft 专用

比较两个 `_variant_t` 对象是否相等。

语法

```
bool operator==(
   const VARIANT& varSrc) const;
bool operator==(
   const VARIANT* pSrc) const;
bool operator!=(
   const VARIANT& varSrc) const;
bool operator!=(
   const VARIANT* pSrc) const;
```

参数

varSrc
要与 `VARIANT` 对象进行比较的 `_variant_t`。

pSrc
指向将与 `VARIANT` 对象进行比较的 `_variant_t` 的指针。

返回值

如果比较成立，则返回 **`true`**，否则返回 **`false`**。

注解

将 `_variant_t` 对象与 `VARIANT` 进行比较，测试是否相等。

结束 Microsoft 专用

另请参阅

_variant_t 类

Last updated on 2025/11/07

_variant_t 提取器

Microsoft 专用

从封装的 `VARIANT` 对象中提取数据。

语法

```
operator short( ) const;
operator long( ) const;
operator float( ) const;
operator double( ) const;
operator CY( ) const;
operator _bstr_t( ) const;
operator IDispatch*( ) const;
operator bool( ) const;
operator IUnknown*( ) const;
operator DECIMAL( ) const;
operator BYTE( ) const;
operator VARIANT() const throw();
operator char() const;
operator unsigned short() const;
operator unsigned long() const;
operator int() const;
operator unsigned int() const;
operator __int64() const;
operator unsigned __int64() const;
```

备注

从封装的 `VARIANT` 中提取原始数据。 如果 `VARIANT` 还不是正确的类型，则使用 `VariantChangeType` 尝试转换，并在失败时生成错误：

- operator short( ) 提取 **`short`** 整数值。
- operator long( ) 提取 **`long`** 整数值。
- operator float( ) 提取 **`float`** 数值。
- operator double( ) 提取 **`double`** 整数值。
- operator CY( ) 提取 `CY` 对象。
- operator bool( ) 提取 **`bool`** 值。

- operator DECIMAL( ) 提取 `DECIMAL` 值。
- operator BYTE( ) 提取 `BYTE` 值。
- operator _bstr_t( ) 提取封装在 `_bstr_t` 对象中的字符串。
- operator IDispatch*( ) 从封装的 `VARIANT` 提取调度接口指针。 将对生成的指针调用 `AddRef`，因此由你决定是否调用 `Release` 来释放它。
- operator IUnknown*( ) 从封装的 `VARIANT` 提取 COM 接口指针。 将对生成的指针调用 `AddRef`，因此由你决定是否调用 `Release` 来释放它。

结束 Microsoft 专用

另请参阅

_variant_t 类

Last updated on 2025/11/07

Microsoft扩展

`asm-statement`:

`__asm` `assembly-instruction` `;` 选择

`__asm {` `assembly-instruction-list` `} ;` 选择

`assembly-instruction-list`:

`assembly-instruction` `;` 选择

`assembly-instruction` `;` `assembly-instruction-list` `;` 选择

`ms-modifier-list`:

`ms-modifier` `ms-modifier-list` 选择

`ms-modifier`:

`__cdecl`

`__fastcall`

`__stdcall`

`__syscall` （保留供将来实现使用）

`__oldcall` （保留供将来实现使用）

`__unaligned` （保留供将来实现使用）

`based-modifier`

`based-modifier`:

`__based (` `based-type` `)`

`based-type`:

`name`

Last updated on 2025/11/07

非标准行为

以下几节将列出 C++ 的 Microsoft 实现不符合 C++ 标准的几处内容。 下面给出的章节号引用了 C++ 11 标准 (ISO/IEC 14882:2011 [E]) 中的章节号。

编译器限制中列出了与 C++ 标准中所定义的限制不同的编译器限制。

协变返回类型

当虚函数具有可变数量的自变量时，不支持虚拟基类作为协变返回类型。 这不符合 C++ 11 ISO 规范第 10.3 节第 7 段的要求。 以下示例没有编译；它会生成编译器错误 C2688：

C++

```
// CovariantReturn.cpp
class A
{
   virtual A* f(int c, ...);   // remove ...
};

class B : virtual A
{
   B* f(int c, ...);   // C2688 remove ...
};
```

绑定模板中的非依赖性名称

在最初分析模板时，Microsoft C++ 编译器暂不支持绑定非依赖性名称。 这不符合 C++ 11 ISO 规范的第 14.6.3 节的要求。 这可能导致在模板之后（但在模板实例化之前）声明的重载出现。

C++

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

namespace N {
   void f(int) { cout << "f(int)" << endl;}
}

template <class T> void g(T) {
    N::f('a');   // calls f(char), should call f(int)
}

namespace N {
    void f(char) { cout << "f(char)" << endl;}
}
```

```
int main() {
    g('c');
}
// Output: f(char)
```

函数异常说明符

分析但不使用除 `throw()` 以外的函数异常说明符。 这不符合 C++ 11 ISO 规范的第 15.4 节的要求。 例如：

C++

```
void f() throw(int); // parsed but not used
void g() throw();    // parsed and used
```

有关异常规范的详细信息，请参阅异常规范。

char_traits::eof()

C++ 标准声明 char_traits::eof 不能对应于有效的 `char_type` 值。 Microsoft C++ 编译器对 **`char`** 类型强制实行该约束，但不对 **`wchar_t`** 类型实行。 这不符合 C++ 11 ISO 规范的第 12.1.1 节中表 62 的要求。 下面的示例说明了此行为。

C++

```
#include <iostream>

int main()
{
    using namespace std;

    char_traits<char>::int_type int2 = char_traits<char>::eof();
    cout << "The eof marker for char_traits<char> is: " << int2 << endl;

    char_traits<wchar_t>::int_type int3 = char_traits<wchar_t>::eof();
    cout << "The eof marker for char_traits<wchar_t> is: " << int3 << endl;
}
```

对象的存储位置

C++ 标准（第 1.8 节第 6 段）要求完整的 C++ 对象具有唯一的存储位置。 但是，在使用 Microsoft C++ 时存在这样的情况：没有数据成员的类型会在对象的生命周期内与其他类型共享一个存储位置。

Last updated on 2025/11/07

编译器限制

C++ 标准建议对各种语言构造施加限制。 下面是 Microsoft C++ 编译器不会在其中实施建议的限制的情况的列表。 第一个数字是 ISO C++ 11 标准（INCITS/ISO/IEC 14882-2011[2012]，附件 B）中建立的限制，而第二个数字是由 Microsoft C++ 编译器实现的限制：

- 复合语句、迭代控制结构和选择控制结构的嵌套级别 - C++ 标准：256，Microsoft C++ 编译器：具体取决于嵌套语句的组合，但通常介于 100 和 110 之间。
- 一个宏定义中的参数 - C++ 标准：256，使用 `/Zc:preprocessor-`:127 或使用 `/Zc:preprocessor`:32767 的 Microsoft C++ 编译器。
- 一个宏调用中的自变量 - C++ 标准：256，使用 `/Zc:preprocessor-`:127 或使用 `/Zc:preprocessor`:32767 的 Microsoft C++ 编译器。
- 字符串文本或宽字符串文本中的字符（串联后）- C++ 标准：65536，Microsoft C++ 编译器：包括 NULL 终止符的 65535 个单字节字符，以及包括 NULL 终止符的 32767 个双字节字符。
- 单个 `struct-declaration-list` 中的嵌套类、结构或联合定义的级别 - C++ 标准：256，Microsoft C++ 编译器：16。
- 构造函数定义中的成员初始值设定项 - C++ 标准：6144，Microsoft C++ 编译器：至少为 6144。
- 一个标识符的范围限定 - C++ 标准：256，Microsoft C++ 编译器：127。
- 嵌套的 **`extern`** 规范 - C++ 标准：1024，Microsoft C++ 编译器：9（不包括全局范围内的隐式 **`extern`** 规范，或计算全局范围内隐式 **`extern`** 规范则为10）。
- 模板声明中的模板自变量 - C++ 标准：1024，Microsoft C++ 编译器：2046。

另请参阅

非标准行为

Last updated on 2025/11/07

C/C++ 预处理器参考

“C/C++ 预处理器参考”对在 Microsoft C/C++ 中实现的预处理器进行了说明。 在将 C 和 C++ 文件传递到编译器之前，预处理器将对这些文件执行预先操作。 可以使用预处理器有条件地编译代码、插入文件、指定编译时错误消息以及将计算机特定规则应用于代码节。

在 Visual Studio 2019 中，/Zc:preprocessor 编译器选项提供完全一致的 C11 和 C17 预处理器。使用编译器标志 `/std:c11` 或 `/std:c17` 时，这是默认设置。

本部分内容

预处理器
概述传统预处理器和符合要求的新预处理器。

预处理器指令
介绍通常用于使源程序易于在不同的执行环境中更改和编译的指令。

预处理器运算符
讨论在 `#define` 指令的上下文中使用的四个预处理器特定运算符。

预定义宏
讨论由 C 和 C++ 标准以及 Microsoft C++ 指定的预定义宏。

杂注
讨论杂注，杂注提供了一种方法来让每个编译器提供计算机和操作系统特定的功能，同时保持与 C 和 C++ 语言的整体兼容性。

相关章节

C++ 语言参考
提供有关 Microsoft 的 C++ 语言实现的参考材料。

C 语言参考
提供有关 Microsoft 的 C 语言实现的参考材料。

C/C++ 生成参考
提供指向讨论编译器和链接器选项的主题的链接。

Visual Studio projects - C++
描述 Visual Studio 中使您能够指定目录（项目系统将在其中进行搜索以找到 C++ 项目的文件）的用户界面。

Last updated on 2025/11/07

C++ 标准库参考 (STL)

C++ 程序可以从符合标准的 C++ 标准库实现中调用大量函数。 这些函数执行服务（如输入和输出），并提供常用操作的高效实现。

若要详细了解如何与相应的 Visual C++ 运行时 `.lib` 文件链接，请参阅 C 运行时 (CRT) 和 C++ 标准库 (STL) .lib 文件。

> **注意**
>
> Microsoft 对 C++ 标准库的实现通常称为 STL 或标准模板库。 尽管 C++ 标准库是 ISO 14882 中定义的库的正式名称，但由于搜索引擎中常用“STL”和“标准模板库”，因此我们偶尔使用这些名称，以便更轻松地查找文档。

根据历史记录，“STL”最初是指 Alexander Stepanov 编写的标准模板库。 该库的一部分与 ISO C 运行时库、Boost 库的一部分和其他功能一起在 C++ 标准库中进行了标准化。 有时，“STL”是指根据 Stepanov 的 STL 改编的 C++ 标准库的容器和算法部分。 在本文档中，标准模板库 (STL) 是指整个 C++ 标准库。

本部分内容

C++标准库概述
概述C++标准库的Microsoft实现。

iostream 编程
`iostream`概述编程。

头文件参考
提供指向有关C++标准库头文件（包含代码示例）的参考主题的链接。

Last updated on 2025/11/07