

# Contents

[C++ 語言文件](#)

[C++ 語言參考](#)

[C++ 語言參考](#)

[歡迎回到 C++ \(新式 C++\)](#)

[語彙慣例](#)

[語彙慣例](#)

[語彙基元和字元集](#)

[註解](#)

[識別項](#)

[關鍵字](#)

[標點符號](#)

[數值、布林值和指標常值](#)

[字串和字元常值](#)

[使用者定義常值](#)

[基本概念](#)

[基本概念](#)

[C++ 類型系統](#)

[範圍](#)

[標頭檔](#)

[編譯單位和連結](#)

[main 函式和命令列引數](#)

[程式終止](#)

[Lvalues 和 Rvalues](#)

[暫存物件](#)

[對齊](#)

[Trivial、標準配置和 POD 類型](#)

[值類型](#)

[類型轉換與類型安全](#)

[標準轉換](#)

內建類型

  內建類型

  資料類型範圍

  nullptr

  void

  bool

  False

  True

  char、wchar\_t、char16\_t、char32\_t

  \_int8、\_int16、\_int32、\_int64

  \_m64

  \_m128

  \_m128d

  \_m128i

  \_ptr32、\_ptr64

  數值限制

    數值限制

    整數限制

    浮點數限制

  宣告和定義

    宣告和定義

    儲存類別

      auto

      const

      constexpr

      extern

    初始設定式

  別名和 typedef

    using 壓告

    volatile

    decltype

  屬性

內建運算子、優先順序和關聯性

內建運算子、優先順序和關聯性

alignof 運算子

\_uuidof 運算子

加法類運算子:+ 和 -

傳址運算子:&

指派運算子

位元 AND 運算子:&

位元互斥 OR 運算子:^

位元非互斥 OR 運算子:|

轉換運算子:()

逗號運算子:,

條件運算子:?:

delete 運算子

等號比較運算子:== 和 !=

明確類型轉換運算子:()

函式呼叫運算子:()

間接取值運算子:\*

左移和右移運算子(>> 和 <<)

邏輯 AND 運算子:&&

邏輯否定運算子:!

邏輯 OR 運算子:||

成員存取運算子:. 和 ->

乘法類運算子和模數運算子

new 運算子

一補數運算子:~

成員指標運算子:.\* 和 ->\*

後置遞增和遞減運算子:++ 和 --

前置遞增和遞減運算子:++ 和 --

關係運算子:<、>、<= 和 >=

範圍解析運算子:::

sizeof 運算子

下標運算子：

typeid 運算子

一元正和負運算子：+ 和 -

運算式

運算式

運算式的類型

運算式的類型

主要運算式

省略符號及可變參數範本

後置運算式

具有一元運算子的運算式

具有二元運算子的運算式

常數運算式

運算式的語意

轉型

轉型

轉換運算子

轉換運算子

dynamic\_cast 運算子

bad\_cast 例外狀況

static\_cast 運算子

const\_cast 運算子

reinterpret\_cast 運算子

執行階段類型資訊 (RTTI)

執行階段類型資訊 (RTTI)

bad\_typeid 例外狀況

type\_info 類別

陳述式

陳述式

C++ 陳述式概觀

標記陳述式

運算陳述式

運算陳述式  
Null 陳述式  
複合陳述式 (區塊)  
選取範圍陳述式  
選取範圍陳述式  
if-else 陳述式  
\_if\_exists 陳述式  
\_if\_not\_exists 陳述式  
switch 陳述式  
反覆運算陳述式  
反覆運算陳述式  
while 陳述式  
do-while 陳述式  
for 陳述式  
以範圍為基礎的 for 陳述式  
跳躍陳述式  
跳躍陳述式  
break 陳述式  
continue 陳述式  
return 陳述式  
goTo 陳述式  
控制權轉移  
命名空間  
列舉  
等位  
函式  
函式  
具有變數引數清單的函式  
函式多載  
明確的預設和已刪除函式  
函式上的引數相依名稱 (Koenig) 查閱  
預設引數

內嵌函式

運算子多載

  運算子多載

  運算子多載的一般規則

  多載一元運算子

  多載一元運算子

  遞增和遞減運算子多載

  二元運算子

  指派

  函式呼叫

  下標

  成員存取

類別和結構

  類別和結構

  Class - 類別

  struct

  類別成員概觀

  成員存取控制

  成員存取控制

  friend

  private

  protected

  public

大括弧初始化

物件存留期及資源管理 (RAII)

適用於編譯時間封裝的 Pimpl 慣用語

在 ABI 界限的可攜性

建構函式

  建構函式

  複製建構函式和複製指派運算子

  移動建構函式和移動指派運算子

  委派建構函式

解構函式

成員函式概觀

成員函式概觀

virtual 規範

override 規範

final 規範

繼承

繼承

虛擬函式

單一繼承

基底類別

多個基底類別

明確覆寫

抽象類別

範圍規則摘要

繼承關鍵字

virtual

\_super

\_interface

特殊成員函式

靜態成員

用為實值型別的 C++ 類別

使用者定義類型轉換

可變動的資料成員

巢狀類別宣告

匿名類別類型

成員的指標

this 指標

位元欄位

C++ 中的 Lambda 運算式

C++ 中的 Lambda 運算式

Lambda 運算式語法

Lambda 運算式的範例

constexpr Lambda 運算式

陣列

reference

reference

左值參考宣告子: &

右值參考宣告子: &&

參考型別函式引數

參考型別函式傳回

指標的參考

指標

指標

原始指標

const 和 volatile 指標

new 和 delete 運算子

智慧指標

作法: 建立和使用 unique\_ptr 執行個體

作法: 建立和使用 shared\_ptr 執行個體

作法: 建立和使用 weak\_ptr 執行個體

作法: 建立和使用 CComPtr 與 CComQIPtr 執行個體

編譯時間封裝的 Pimpl

C++ 中的例外狀況處理

C++ 中的例外狀況處理

新型 C++ 的最佳做法

如何設計例外狀況的安全

如何連結例外狀況與非例外狀況代碼

try、throw 和 catch 陳述式

Catch 區塊的評估方式

例外狀況與堆疊回溯

例外狀況規格 (throw)

noexcept

未處理的 C++ 例外狀況

混合 C (結構化) 和 C++ 例外狀況

混合 C (結構化) 和 C++ 例外狀況

使用 setjmp-longjmp

使用 C++ 處理結構化例外狀況

結構化例外狀況處理 (SEH) (C/C++)

結構化例外狀況處理 (SEH) (C/C++)

撰寫例外狀況處理常式

撰寫例外狀況處理常式

try-except 陳述式

撰寫例外狀況篩選條件

引發軟體例外狀況

硬體例外狀況

例外狀況處理常式的限制

撰寫終止處理常式

撰寫終止處理常式

try-finally 陳述式

清除資源

例外狀況處理的時機 : 摘要

終止處理常式的限制

在執行緒之間傳輸例外狀況

判斷提示和使用者提供的訊息

判斷提示和使用者提供的訊息

static\_assert

模組

C++ 中的模組概觀

模組、匯入、匯出

範本

範本

typename

類別範本

函式範本

函式範本

函式範本具現化  
明確初始化  
函式範本的明確特製化  
函式範本的部分排序  
成員函式範本  
範本特製化  
範本和名稱解析  
範本和名稱解析  
相依類型的名稱解析  
區域宣告名稱的名稱解析  
函式範本呼叫的多載解析  
原始程式碼組織 (C++ 範本)

事件處理

事件處理

`_event`

`_hook`

`_raise`

`_unhook`

原生 C++ 中的事件處理

COM 中的事件處理

Microsoft 特定修飾詞

Microsoft 特定修飾詞

基底定址

基底定址

`_based` 文法

基底指標

呼叫慣例

呼叫慣例

引數傳遞和命名慣例

引數傳遞和命名慣例

`_cdecl`

`_clrcall`

`_stdcall`

`_fastcall`

`_thiscall`

`_vectorcall`

呼叫範例 : 函式原型和呼叫

呼叫範例 : 函式原型和呼叫

呼叫範例的結果

Naked 函式呼叫

Naked 函式呼叫

Naked 函式的規則和限制

撰寫初構/終解程式碼的考量

浮點數副處理器和呼叫慣例

淘汰呼叫慣例

`restrict` (C++ AMP)

`tile_static` 關鍵字

`_declspec`

`_declspec`

`align`

`allocate`

`allocator`

`appdomain`

`code_seg` (`_declspec`)

`deprecated`

`dllexport`, `dllimport`

`dllexport`, `dllimport`

定義和宣告

使用 `dllexport` 和 `dllimport` 定義內嵌 C++ 函式

一般規則和限制

在 C++ 類別中使用 `dllimport` 和 `dllexport`

`jit intrinsic`

`naked`

`noalias`

noinline  
noreturn  
nothrow  
novtable  
處理序  
屬性  
restrict  
safebuffers  
selectany  
spectre  
執行緒  
uuid  
\_restrict  
\_sptr、\_uptr  
\_unaligned  
\_w64  
\_func\_  
編譯器 COM 支援  
編譯器 COM 支援  
編譯器 COM 全域函式  
編譯器 COM 全域函式  
\_com\_raise\_error  
ConvertStringToBSTR  
ConvertBSTRToString  
\_set\_com\_error\_handler  
編譯器 COM 支援類別  
編譯器 COM 支援類別  
\_bstr\_t 類別  
\_bstr\_t 類別  
\_bstr\_t 成員函式  
\_bstr\_t 成員函式  
\_bstr\_t::Assign

\_bstr\_t::Attach  
\_bstr\_t::\_bstr\_t  
\_bstr\_t::copy  
\_bstr\_t::Detach  
\_bstr\_t::GetAddress  
\_bstr\_t::GetBSTR  
\_bstr\_t::length  
**\_bstr\_t 運算子**  
  **\_bstr\_t 運算子**  
  \_bstr\_t::operator =  
  \_bstr\_t::operator +=、+  
  \_bstr\_t::operator !=  
**\_bstr\_t 關係運算子**  
  \_bstr\_t::wchar\_t \*、\_bstr\_t::char\*

**\_com\_error 類別**  
  **\_com\_error 類別**  
  **\_com\_error 成員函式**  
    **\_com\_error 成員函式**  
    \_com\_error::\_com\_error  
    \_com\_error::Description  
    \_com\_error::Error  
    \_com\_error::ErrorInfo  
    \_com\_error::ErrorMessage  
    \_com\_error::GUID  
    \_com\_error::HelpContext  
    \_com\_error::HelpFile  
    \_com\_error::HRESULTToWCode  
    \_com\_error::Source  
    \_com\_error::WCode  
    \_com\_error::WCodeToHRESULT  
**\_com\_error 運算子**  
  **\_com\_error 運算子**

\_com\_error::operator =  
\_com\_ptr\_t 類別  
\_com\_ptr\_t 類別  
\_com\_ptr\_t 成員函式  
\_com\_ptr\_t 成員函式  
\_com\_ptr\_t::com\_ptr\_t  
\_com\_ptr\_t::AddRef  
\_com\_ptr\_t::Attach  
\_com\_ptr\_t::CreateInstance  
\_com\_ptr\_t::Detach  
\_com\_ptr\_t::GetActiveObject  
\_com\_ptr\_t::GetInterfacePtr  
\_com\_ptr\_t::QueryInterface  
\_com\_ptr\_t::Release  
\_com\_ptr\_t 運算子  
\_com\_ptr\_t 運算子  
\_com\_ptr\_t::operator =  
\_com\_ptr\_t 關係運算子  
\_com\_ptr\_t 擷取器

關係函式範本

\_variant\_t 類別  
\_variant\_t 類別  
\_variant\_t 成員函式  
\_variant\_t 成員函式  
\_variant\_t::variant\_t  
\_variant\_t::Attach  
\_variant\_t::Clear  
\_variant\_t::ChangeType  
\_variant\_t::Detach  
\_variant\_t::SetString  
\_variant\_t 運算子  
\_variant\_t 運算子

`_variant_t::operator =`

`_variant_t` 關係運算子

`_variant_t` 擷取器

Microsoft 延伸模組

非標準行為

編譯器限制

C/C++ 前置處理器參考

C++ 標準程式庫參考

# C++ 語言參考

2020/11/2 • [Edit Online](#)

本參考說明在 Microsoft C++ 編譯器中實作為 C++ 程式設計語言。組織是根據 Margaret Ellis 和 Bjarne Stroustrup 以及 ANSI/ISO C++ 國際標準(ISO/IEC FDIS 14882)上標注的 C++ 參考手冊。已包含 Microsoft 專有 C++ 語言功能實作。

如需現代化 C++ 程式設計實務的總覽，請參閱[歡迎回到 C++](#)。

請參閱下表，快速尋找關鍵字或運算子：

- [C++ 關鍵字](#)
- [C++ 運算子](#)

## 本節內容

### 詞法慣例

C++ 程式的基本語彙元素：語彙基元、註解、運算子、關鍵字、標點符號、常值。另外還有檔案轉譯、運算子優先順序/關聯性。

### 基本概念

範圍、連結、程式啟動和結束、儲存類別以及類型。

**內建類型** C++ 編譯器和其值範圍內建的基本類型。

### 標準轉換

內建類型之間的類型轉換。另外還有算術轉換，以及指標、參考與成員指標類型之間的轉換。

**宣告和定義** 宣告和定義變數、類型和函數。

### 運算子、優先順序及關聯性

C++ 中的運算子。

### 運算式

運算式類型、運算式語意、運算子參考主題、轉型和轉型運算子、執行階段類型資訊。

### Lambda 運算式

程式設計技巧，可隱含定義函式物件類別和建構該類別類型的函式物件。

### 陳述式

運算式、null、複合、選取、反覆項目、跳躍和宣告陳述式。

### 類別和結構

類別、結構和等位簡介。此外，成員函式、特殊成員函式、資料成員、位欄位、`this` 指標、嵌套類別。

### 等位

使用者定義的類型，其中所有成員都會共用相同的記憶體位置。

### 衍生類別

單一和多重繼承、`virtual` 函數、多個基類、抽象類、範圍規則。此外，`__super` 和 `__interface` 關鍵字。

### 成員存取控制

控制對類別成員的存取：`public`、`private` 和 `protected` 關鍵字。Friend 函式和類別。

### 多載化

多載運算子，運算子多載的規則。

## 例外狀況處理

C++ 例外狀況處理、結構化例外狀況處理 (SEH)、用於撰寫例外狀況處理陳述式的關鍵字。

## 判斷提示和使用者提供的訊息

`#error` 指示詞、`static_assert` 關鍵字、`assert` 宏。

## 範本

範本規格、函式樣板、類別樣板、`typename` 關鍵字、範本與宏、範本和智慧型指標。

## 事件處理

宣告事件及事件處理常式。

## Microsoft 專有的修飾詞

Microsoft C++ 專有的修飾詞。記憶體定址、呼叫慣例、`naked` 函數、擴充的儲存類別屬性(`__declspec`)、`__w64`。

## 內嵌組譯工具

在區塊中使用元件語言和 C++ `__asm`。

## 編譯器 COM 支援

Microsoft 專有類別和全域函式的參考，可用來支援 COM 類型。

## Microsoft 擴充功能

Microsoft C++ 擴充功能。

## 非標準行為

Microsoft C++ 編譯器非標準行為的相關資訊。

## 歡迎回到 C++

概述撰寫安全、正確且有效率的程式的新式 C++ 程式設計實務。

# 相關章節

## 執行階段平台的元件延伸模組

有關使用 Microsoft C++ 編譯器以 .NET 為目標的參考資料。

## C/C++ 建置參考

編譯器選項、連結器選項和其他建置工具。

## C/c++ 預處理器參考

有關 `pragma`、前置處理器指示詞、預先定義巨集和前置處理器的參考資料。

## Visual C++ 程式庫

各種 Microsoft C++ 程式庫參考起始頁的連結清單。

# 另請參閱

## C 語言參考

# 歡迎回到 C++ (現代 C++)

2020/11/2 • [Edit Online](#)

在建立之後, C++ 已經成為世界上最廣泛使用的程式設計語言之一。編寫完善的 C++ 程式不但執行快速, 而且有效率。語言比其他語言更有彈性: 它可以在最高層級的抽象層級運作, 並在晶片的層級下運作。C++ 提供高度優化的標準程式庫。它可讓您存取低層級的硬體功能, 以最快速度和將記憶體需求降至最低。您可以使用 C++ 建立各式各樣的應用程式。遊戲、設備磁碟機, 以及高效能的科學軟體。內嵌程式。Windows 用戶端應用程式。甚至其他程式設計語言的程式庫和編譯器都是以 C++ 撰寫。

C++ 的其中一個原始需求是與 C 語言的回溯相容性。因此, C++ 一律允許 C 樣式的程式設計, 以及原始指標、陣列、以 null 結束的字元字串, 以及其他功能。它們可能會啟用絕佳的效能, 但也會產生 bug 和複雜度。C++ 的演進具有強調的功能, 可大幅減少使用 C 樣式慣用語的需求。當您需要舊版的 C 程式設計工具時, 您需要使用新式 C++ 程式碼。新式 C++ 程式碼較簡單、更安全、更簡潔, 而且仍會像以往一樣快速。

下列各節提供新式 C++ 主要功能的總覽。除非另有說明, 否則此處所列的功能可在 C++ 11 和更新版本中使用。在 Microsoft C++ 編譯器中, 您可以設定 `/std` 編譯器選項, 以指定要用於專案的標準版本。

## 資源和智慧型指標

C 樣式程式設計中的其中一個主要 bug 類別是 記憶體流失。遺漏的原因通常是因為無法呼叫配置 `delete` 給的記憶體 `new`。新式 C++ 強調資源取得的準則是 (RAII) 的 初始化。概念很簡單。資源 (堆積記憶體、檔案控制代碼、通訊端等) 應該由物件 擁有。該物件會在其函式中建立或接收新配置的資源, 並在其函式中將它刪除。RAII 的原則可保證當擁有物件超出範圍時, 所有資源都會正確地傳回作業系統。

為了支援簡單採用 RAII 準則, C++ 標準程式庫提供三種智慧型指標類型: `std::unique_ptr`、`std::shared_ptr` 和 `std::weak_ptr`。智慧型指標可處理它所擁有之記憶體的配置和刪除。下列範例顯示的類別具有在的呼叫中, 在堆積上配置的陣列成員 `make_unique()`。對和的 `new` 呼叫 `delete` 會由類別封裝 `unique_ptr`。當 `widget` 物件超出範圍時, 就會叫用 `unique_ptr` 的函式, 並釋放為數組配置的記憶體。

```
#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
```

請盡可能在配置堆積記憶體時使用智慧型指標。如果您必須明確地使用 `new` 和 `delete` 運算子, 請遵循 RAII 的原則。如需詳細資訊, 請參閱 [物件存留期和資源管理 \(RAII\)](#)。

`std::string` 和 `std::string_view`

C 樣式字串是另一個主要的錯誤來源。藉由使用 `std::string` 和 `std::wstring`，您可以消除幾乎所有與 C 樣式字串相關的錯誤。您也會獲得搜尋、附加、前置等等的成員函式的優點。兩者都經過高度優化，以加快速度。將字串傳遞至只需要唯讀存取權的函式時，您可以使用 c + + 17，`std::string_view` 以獲得更好的效能優勢。

## `std::vector` 和其他標準程式庫容器

標準程式庫容器全都遵循 RAI 的原則。它們提供反覆運算器來進行專案的安全遍歷。而且，它們已針對效能進行高度優化，並已針對正確性進行徹底測試。藉由使用這些容器，您可以消除自訂資料結構可能引進的錯誤或效率不佳的可能性。不是原始陣列，而是以 `vector` c + + 中的連續容器來使用。

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

使用 `map` (不 `unordered_map`) 為預設的關聯容器。`set` `multimap` `multiset` 針對退化和多重案例使用、和。

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

當需要效能優化時，請考慮使用：

- 內嵌 `array` 時的型別很重要，例如，做為類別成員。
- 未排序的關聯容器（例如）`unordered_map`。這些專案具有較低的每個專案的額外負荷和長期查閱，但可能難以正確且有效率地使用。
- 排序 `vector`。如需詳細資訊，請參閱[演算法](#)。

請勿使用 C 樣式陣列。對於需要直接存取資料的舊版 API，請改用存取子方法（例如）`f(vec.data(), vec.size())`。如需容器的詳細資訊，請參閱[c + + 標準程式庫容器](#)。

## 標準程式庫演算法

在假設您需要為程式撰寫自訂演算法之前，請先參閱 c + + 標準程式庫 [演算法](#)。標準程式庫包含不斷成長的演算法，適用於許多常見的作業，例如搜尋、排序、篩選和隨機化。數學程式庫很廣泛。從 c + + 17 開始，提供許多演算法的平行版本。

以下是一些重要的範例：

- `for_each`，預設的「遍歷演算法」（以及以範圍為基礎的 `for` 迴圈）。
- `transform`，用於就地修改容器元素
- `find_if`，預設搜尋演算法。
- `sort`、`lower_bound` 和其他預設排序和搜尋演算法。

若要撰寫比較子，請使用 strict，`<` 並在可以時使用 命名的 *lambda*。

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

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

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

## auto 而不是明確的型別名稱

C + + 11 引進了在 `auto` 變數、函式和範本宣告中使用的關鍵字。`auto` 告訴編譯器推斷物件的型別，讓您不必明確地輸入它。`auto` 當推算的型別是嵌套的範本時特別有用：

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

## 以範圍為基礎的 for 迴圈

在陣列和容器上進行 C 樣式反覆運算很容易就會產生錯誤的索引，而且鍵入的繁瑣也相當繁瑣。若要排除這些錯誤，並讓您的程式碼更容易閱讀，請使用以範圍為基礎的 `for` 迴圈搭配標準程式庫容器和原始陣列。如需詳細資訊，請參閱以 [範圍為基礎的 for 語句](#)。

```
#include <iostream>  
#include <vector>  
  
int main()  
{  
    std::vector<int> v {1,2,3};  
  
    // C-style  
    for(int i = 0; i < v.size(); ++i)  
    {  
        std::cout << v[i];  
    }  
  
    // Modern C++:  
    for(auto& num : v)  
    {  
        std::cout << num;  
    }  
}
```

## constexpr 運算式而非宏

C 和 c + + 中的宏是在編譯之前由預處理器處理的權杖。在編譯檔案之前，會將宏標記的每個實例取代為其定義的值或運算式。宏通常用於 C 樣式程式設計中，以定義編譯時間常數值。不過，宏很容易出錯，而且難以進行調試。在新式 c + + 中，您應該偏好將 `constexpr` 變數用於編譯時期常數：

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

## 統一初始化

在新式 c + + 中，您可以針對任何類型使用括弧初始化。初始化陣列、向量或其他容器時，這種形式的初始化特別方便。在下列範例中，`v2` 會使用三個實例來初始化 `s`。`v3` 初始化時，會使用以 `s` 大括弧初始化的三個實例。編譯器會根據的宣告型別推斷每個元素的型別 `v3`。

```

#include <vector>

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

int main()
{
    // C-style initialization
    std::vector<S> v;
    S s1("Norah", 2.7);
    S s2("Frank", 3.5);
    S s3("Jeri", 85.9);

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

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

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

}

```

如需詳細資訊，請參閱 [括弧初始化](#)。

## 移動語義

新式 C++ 提供 **移動語義**，讓您可以消除不必要的記憶體複製。在舊版的語言中，在某些情況下無法避免複製。移動作業會將資源的擁有權從某個物件轉移至下一個物件，而不需要進行複製。某些類別擁有堆積記憶體、檔案控制代碼等資源。當您執行資源擁有的類別時，您可以定義 **移動函式** 和 **移動指派運算子**。編譯器會在不需要複製的情況下，于多載解析期間選擇這些特殊成員。標準程式庫容器類型會叫用物件上的移動函式（如果有定義的話）。如需詳細資訊，請參閱 [移動函數和移動指派運算子 \(C++\)](#)。

## Lambda 運算式

在 C 樣式程式設計中，函式可以使用函式 **指標傳遞** 至另一個函式。函式指標不太方便維護和瞭解。它們所參考的函式可以在原始程式碼中的其他地方定義，而不是從叫用它的位置。此外，它們不是型別安全。新式 C++ 提供的函式 **物件** 是覆寫運算子的類別 [operator\(\)](#)，可讓它們以函式的形式來呼叫。建立函式物件最方便的方式，就是使用內嵌 **lambda 運算式**。下列範例示範如何使用 lambda 運算式來傳遞函式物件，該函式會在 **for\_each** 向量中的每個專案上叫用函式：

```

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; });

```

Lambda 運算式可以讀取為「函式，該函式會 `[=](int i) { return i > x && i < y; }` 採用類型的單一引數 `int`，並傳回布林值，指出引數是否大於 `x` 且小於 `y`」。請注意，您 `x` `y` 可以在 lambda 中使用來自周圍內容的變數和。指定以傳 `[=]` 值方式 **捕捉** 這些變數；換句話說，lambda 運算式有自己的這些值的複本。

## 例外狀況

新式 C++ 強調例外狀況，而不是錯誤碼，作為報告和處理錯誤狀況的最佳方式。如需詳細資訊，請參閱 [新式 C++ 的例外狀況和錯誤處理最佳做法](#)。

## std::atomic

針對執行緒間通訊機制，使用 C++ 標準程式庫 [std::atomic](#) 結構和相關類型。

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

等位通常用於 C 樣式程式設計中，藉由讓不同類型的成員佔用相同的記憶體位置來節省記憶體。不過，等位不是型別安全，而且很容易發生程式設計錯誤。C++ 17 引進了類別，做為等位 [std::variant](#) 的更健全且安全的替代方案。[std::visit](#) 函數可以用型別安全的方式，用來存取型別的成員 [variant](#)。

## 另請參閱

[C++ 語言參考](#)

[Lambda 運算式](#)

[C++ 標準程式庫](#)

[Microsoft C++ 語言一致性資料表](#)

# 語彙慣例

2020/3/25 • [Edit Online](#)

本節將介紹 C++ 程式的基本項目。您可以使用這些稱為「語彙項目」或「語彙基元」的項目，建構用來建構完整程式的陳述式、定義、宣告等等。本節將討論下列語彙項目：

- [標記和字元集](#)
- [註解](#)
- [識別碼](#)
- [關鍵字](#)
- [標點符號](#)
- [數值、布林值和指標常值](#)
- [字串和字元常值](#)
- [使用者定義常值](#)

如需有關如何 C++ 剖析原始程式檔的詳細資訊，請參閱[轉譯階段](#)。

## 另請參閱

[C++ 語言參考](#)

[轉譯單位和連結](#)

# 標記和字元集

2020/1/10 • [Edit Online](#)

C++ 程式的文字包含標記和空白字元。語彙基元是 C++ 程式中對編譯器有意義的最小項目。C++ 剖析器會辨識這類權杖：

- [關鍵字](#)
- [識別項](#)
- [數值、布林值和指標常值](#)
- [字串和字元常值](#)
- [使用者定義常值](#)
- [運算子](#)
- [標點符號](#)

標記通常會以空白字元分隔，這可以是一或多個：

- 空白
- 水平或垂直定位字元
- 新行
- 表單摘要
- 註解

## 基本來源字元集

C++ 標準指定可用於原始程式檔的基本來源字元集。為了表示不在這個集合中的字元，還可使用「通用字元名稱」(Universal Character Name) 來指定額外的字元。MSVC 執行允許額外的字元。基本來源字元集是由可用於原始程式檔中的 96 個字元所組成。這個集合包含空白字元、水平定位字元、垂直定位字元、換頁字元和換行控制字元，以及下列一組圖形字元：

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 專屬

MSVC 會將 \$ 字元包含為基本來源字元集的成員。MSVC 也允許根據檔案編碼，在原始程式檔中使用一組額外的字元。根據預設，Visual Studio 會使用預設字碼頁來儲存原始程式檔。當使用地區設定特定字碼頁或 Unicode 字碼頁來儲存原始程式檔時，MSVC 可讓您在原始程式碼中使用該字碼頁的任何字元，但基本來源字元集中未明確允許的控制碼除外。例如，如果您使用日文字碼頁來儲存檔案，則可以將日文字元放在註解、識別項或字串常值中。MSVC 不允許無法轉譯成有效多位元組字元或 Unicode 程式碼點的字元序列。根據編譯器選項，並非所有允許的字元都可出現在識別項中。如需詳細資訊，請參閱 [Identifiers](#)。

### 結束 Microsoft 專屬

#### 通用字元名稱

由於 C++ 程式可以使用比基本來源字元集所指定的字元還要多的字元，因此您可以使用「通用字元名稱」(Universal Character Name)，透過移植的方式來指定這些字元。通用字元名稱是由代表 Unicode 字碼指標的字元序列所組成。這些字元採用兩種格式。使用 \UNNNNNNNN 來表示 U+NNNNNNNN 格式的 Unicode 字碼指標，其中

NNNNNNNN 是八位數的十六進位字碼指標數字。使用四位數的 `\uNNNN` 來表示 U+0000NNNN 格式的 Unicode 字碼指標。

通用字元名稱可用於識別項、字串和字元常值中。通用字元名稱不可用來代表範圍 0xD800-0xDFFF 中的 Surrogate 字碼指標。請改用想要的字碼指標；編譯器會自動產生任何必要的 Surrogate。可用於識別項的通用字元名稱還有其他限制。如需詳細資訊，請參閱 [Identifiers](#) 和 [String and Character Literals](#)。

## Microsoft 專屬

Microsoft C++ 編譯器會將通用字元名稱格式的字元與常值形式交換。例如，您可以使用通用字元名稱格式宣告一個識別項，然後以常值格式來使用該識別項：

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

Windows [剪貼簿] 上的擴充字元格式會因應用程式地區設定而有所不同。將這些字元從另一個應用程式剪下並貼到您的程式碼可能會採用未預期的字元編碼方式。這會導致程式碼中出現沒有明顯原因的剖析錯誤。建議您先將原始程式檔編碼方式設定為 Unicode 字碼頁，再剖析擴充字元。此外，也建議您使用 IME 或字元對應表應用程式來產生擴充字元。

## 結束 Microsoft 專屬

### 執行字元集

執行字元集代表可以在已編譯器中出現的字元和字串。這些字元集是由原始程式檔中允許的所有字元所組成，以及代表警示、倒退鍵、回車和 null 字元的控制字元。執行字元集具有地區設定特定表示法。

# 批註 ( C++ )

2020/3/25 • • [Edit Online](#)

註解是會被編譯器忽略，但對程式設計人員而言很有用的文字。註解通常用來標註程式碼供未來參考。編譯器會將它們視為空白字元。您可以使用測試中的批註，讓特定的程式程式碼處於非使用中狀態；不過，`#if` / `#endif` 預處理器指示詞的效果更好，因為您可以圍繞包含批註的程式碼，但無法嵌套批註。

C++ 註解以下列其中一種方式撰寫：

- `/*` (斜線、星號) 字元，後面接著任何字元序列 (包括新行)，後面接著 `*/` 字元。這語法與 ANSI C 相同。
- `//` (兩個斜線) 字元，後面接著任何字元序列。一個沒有前置反斜線的新行會終止這種形式的註解。因此，它通常稱為「單行註解」。

註解字元 (`/*`、`*/` 和 `//`) 在字元常數、字串常值或註解中沒有特殊意義。因此，使用第一種語法的註解不可以是巢狀。

## 另請參閱

[語彙慣例](#)

# 識別項 (C++)

2020/11/2 • [Edit Online](#)

識別項是字元序列，用來表示下列其中一項：

- 物件或變數名稱
- 類別、結構或等位名稱
- 列舉類型名稱
- 類別、結構、等位或列舉的成員
- 函式或類別成員函式
- `typedef` 名稱
- 標籤名稱
- 巨集名稱
- 巨集參數

下列字元可做為識別項的任何字元：

```
_ 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
```

識別項中也允許特定範圍的通用字元名稱。識別項中的通用字元名稱無法指定控制字元或基本來源字元集中的字元。如需詳細資訊，請參閱 [Character Sets](#)。下列 Unicode 字碼指標數字範圍可做為通用字元名稱，以代表識別項中的任何字元：

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

下列字元在識別項中，除了第一個字元以外，都可做為其任何字元：

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

下列 Unicode 字碼指標數字範圍也可做為通用字元名稱，以代表識別項中除了第一個字元以外的任何字元：

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

## Microsoft 特定的

Microsoft C++ 識別項只有前 2048 個字元是有意義的。使用者定義類型的名稱已由編譯器「裝飾」來保存類型資訊。產生的名稱（含類型資訊）不能超過 2048 個字元（如需詳細資訊，請參閱 [裝飾名稱](#)）。可能影響裝飾識別碼長度的因素如下：

- 識別項表示的是使用者定義類型的物件還是衍生自使用者定義類型的類型。
- 識別項表示的是函式還是衍生自函式的類型。
- 函式的引數數目。

貨幣符號 `$` 是 Microsoft C++ 編譯器 (MSVC) 中的有效識別碼字元。MSVC 也可讓您使用識別碼中允許範圍的通用字元名稱所代表的實際字元。若要使用這些字元，您必須使用包含這些字元的檔案編碼字碼頁來儲存檔案。下列範例示範如何在程式碼中交換使用擴充字元和通用字元名稱。

```
// 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
}
```

編譯 C++/CLI 程式碼時，識別項中允許之字元範圍的限制較少。使用 /clr 編譯之程式碼中的識別項應該遵循 [標準 ECMA-335：通用語言基礎結構 \(CLI\)](#)。

## 結束 Microsoft 專有

識別項的第一個字元必須是字母 (大寫或小寫) 字元或底線 (`_`)。由於 C++ 識別項要區分大小寫，`fileName` 與 `FileName` 不同。

識別項的拼字、大小寫不能與關鍵字完全相同。包含關鍵字的識別項是合法的。例如，`Pint` 是合法的識別碼，即使它包含，也 `int` 就是關鍵字。

在識別碼中使用兩個連續底線字元 (`__`)，或單一前置底線後面接著大寫字母，則會保留給所有範圍中的 C++ 程式。因為可能與目前或未來保留的識別項相衝突，您應該避免在具有檔案範圍的名稱中使用一個後面接著小寫字母的前置底線。

## 另請參閱

[詞法慣例](#)

# 關鍵字 (C++)

2020/11/2 • [Edit Online](#)

關鍵字是具有特殊意義的預先定義保留識別項。它們無法當做程式中的識別碼使用。下列是 Microsoft C++ 的保留關鍵字。具有前置底線的名稱和針對 c++/CX 和 c++/CLI 指定的名稱是 Microsoft 擴充功能。

## Standard C++ 關鍵字

alignas  
alignof  
and<sup>b</sup>  
and\_eq<sup>b</sup>  
asm<sup>a</sup>  
auto  
bitand<sup>b</sup>  
bitor<sup>b</sup>  
bool  
break  
case  
catch  
char  
char8\_t<sup>c</sup>  
char16\_t  
char32\_t  
class  
compl<sup>b</sup>  
concept<sup>c</sup>  
const  
const\_cast  
constexpr<sup>c</sup>  
constinit<sup>c</sup>  
continue  
co\_await<sup>c</sup>  
co\_return<sup>c</sup>  
co\_yield<sup>c</sup>  
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 申報
```

Microsoft 特定的 `__asm` 關鍵字取代 C++ `asm` 語法。`asm` 是為了與其他 C++ 執行相容而保留，但

不會執行。`__asm` 在 x86 目標上使用內嵌元件。Microsoft C++ 不支援其他目標的內嵌元件。

b當 `/permissive-` 指定或 `/za` (停用語言延伸) 時，擴充運算子同義字是關鍵字。當啟用 Microsoft 擴充功能時，它們不是關鍵字。

指定時支援`/std:c++latest`。

## Microsoft 特定的 C++ 關鍵字

在 C++ 中，包含兩個連續底線的識別碼會保留給編譯器執行。Microsoft 慣例是在 Microsoft 特定關鍵字之前加上雙底線。這些字組無法用作識別碼名稱。

Microsoft 擴充功能預設為啟用。為了確保您的程式可完整移植，您可以在 `/permissive-` 編譯期間指定或停用 `/za` (語言延伸) 選項，以停用 Microsoft 擴充功能。這些選項會停用部分 Microsoft 特定的關鍵字。

啟用 Microsoft 擴充功能後，您可以在程式中使用 Microsoft 專有的關鍵字。為符合 ANSI 標準，這些關鍵字前面都加上雙底線。為了回溯相容性，支援多個雙底線關鍵字的單一底線版本。`__cdecl` 關鍵字的可用方式沒有前置底線。

`__asm` 關鍵字取代 C++ `asm` 語法。`asm` 是為了與其他 C++ 執行相容而保留，但不會執行。使用 `__asm`。

`__based` 關鍵字的用途有限，可用於 32 位和 64 位目標編譯。

`__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` pci-e

```
__raise
__restrict e
__single_inheritance pci-e
__sptr pci-e
__stdcall e

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

用於事件處理的<sup>d</sup>內建函式。

<sup>e</sup> 為了與舊版的回溯相容性，在預設) (啟用 Microsoft 擴充功能時，可使用兩個前置底線和單一前置底線來使用這些關鍵字。

## \_\_Declspec 修飾詞中的 Microsoft 關鍵字

這些識別碼是修飾詞的擴充屬性 `__declspec`。它們在該內容中會被視為關鍵字。

```
align
allocate
allocator
appdomain
code_seg
deprecated

dllexport
dllimport
jitintrinsic
naked
noalias
noinline

noreturn
nothrow
novtable
process
property
restrict

safebuffers
selectany
spectre
thread
uuid
```

## C++/CLI 和 C++/CX 關鍵字

<code>__abstract</code>	f
<code>__box</code>	f
<code>__delegate</code>	f
<code>__gc</code>	f
<code>__identifier</code>	
<code>__nogc</code>	f
<code>__noop</code>	
<code>__pin</code>	f
<code>__property</code>	f
<code>__sealed</code>	f
<code>__try_cast</code>	f
<code>__value</code>	f
<code>abstract</code>	g
<code>array</code>	g
<code>as_friend</code>	
<code>delegate</code>	g
<code>enum class</code>	
<code>enum struct</code>	
<code>event</code>	g
<code>finally</code>	
<code>for each in</code>	
<code>gcnew</code>	g
<code>generic</code>	g
<code>initonly</code>	
<code>interface class</code>	g
<code>interface struct</code>	g
<code>interior_ptr</code>	g
<code>literal</code>	g
<code>new</code>	g
<code>property</code>	g
<code>ref class</code>	
<code>ref struct</code>	
<code>safecast</code>	
<code>sealed</code>	g
<code>typeid</code>	
<code>value class</code>	g
<code>value struct</code>	g

<sup>f</sup> 僅適用於 Managed Extensions for C++。這個語法現在不建議使用。如需詳細資訊，請參閱[執行階段平台的元件延伸模組](#)。

<sup>g</sup> 適用於 C++/cli

## 另請參閱

[語彙慣例](#)

[C++ 內建運算子、優先順序和關聯性](#)

# 標點符號 ( C++ )

2020/3/25 • [Edit Online](#)

在 C++ 中的標點符號對於編譯器具有語法和語意上的含意，但本身並不指定產生值的作業。不論是單獨或組合，有些標點符號也可以是 C++ 運算子，或對前置處理器具有重大意義。

下列任何字元皆視為標點符號：

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

標點符號 [] 、() 和 {} 必須在[轉譯階段4](#) 之後成對出現。

## 另請參閱

[語彙慣例](#)

# 數值、布林值和指標常值

2020/11/2 • [Edit Online](#)

常值是直接代表值的程式項目。本文涵蓋整數、浮點數、布林值和指標類型的常值。如需字串和字元常值的相關資訊，請參閱 [\(C++\) 的字串和字元常值](#)。您也可以根據任何類別來定義您自己的常值。如需詳細資訊，請參閱 [\(C++\) 的使用者定義常值](#)

您可以在許多內容中使用常值，但最常見的是初始化具名變數，以及將引數傳遞給函式：

```
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
```

有時，重要的是告訴編譯器如何解譯常值，或提供給它哪個特定類型。它是藉由將前置詞或尾碼附加至常值來完成。例如，前置詞 `0x` 會指示編譯器將其後的數位解讀為十六進位值 `0x35`。`ULL` 尾碼會告知編譯器將值視為 `unsigned long long` 類型，如下所示 `5894345ULL`。如需每個常值類型之前置詞和後置詞的完整清單，請參閱下列各節。

## 整數常值

整數常值的開頭是數字，而且沒有小數部分或指數。您可以指定十進位、二進位、八進位或十六進位格式的整數常值。您可以使用後置詞，選擇性地將整數常值指定為不帶正負號的整數常值，以及 `long` 或 `long long` 類型。

當沒有前置詞或後置詞時，如果值符合，則編譯器會提供整數常值型別 `int` (32 位)，否則它會提供 `long long` (64 位) 的類型。

若要指定十進位整數常值，請使用非零數字開始指定。例如：

```
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
```

若要指定八進位整數常值，請使用 0 開始指定，後接範圍 0 到 7 的一連串數字。數字 8 和 9 是在指定八進位常值中的錯誤。例如：

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

若要指定十六進位整數常值，請使用 `0x` 或 `0X` ("x" 的大小寫) 不重要，後面接著範圍中的數位序列，`0` 到 `9` 並 `a` (或 `A`) 到 `f` (或 `F`)。十六進位數字 `a` (或 `A`) 到 `f` (或 `F`) 代表範圍 10 到 15 的值。例如：

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

若要指定不帶正負號的類型，請使用 `u` 或 `U` 尾碼。若要指定 `long` 型別，請使用 `l` 或 `L` 尾碼。若要指定 64 位元整數類型，請使用 `LL` 或 `ll` 後置詞。仍支援 `i64` 尾碼，但我們不建議這麼做。這是 Microsoft 專屬的，不可移植。例如：

```
unsigned val_1 = 328u;           // Unsigned value
long val_2 = 0x7FFFFFFL;         // Long value specified
                                // as hex literal
unsigned long val_3 = 0776745ul; // Unsigned long value
auto val_4 = 108LL;              // signed long long
auto val_4 = 0x800000000000000ULL << 16; // unsigned long long
```

**數位分隔符號**: 您可以使用單引號字元 (單引號) 以較大的數位分隔位置值，讓人們更容易閱讀。分隔符號對於編譯沒有作用。

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

## 浮點常值

浮點常值指定必須有小數部分的值。這些值包含小數點 (.)，而且可以包含指數。

浮點數常值具有 **有效位數** (有時稱為 **尾數**)，可指定數位的值。它們具有 **指數**，可指定數位的大小。而且，它們具有指定常數值型別的選擇性尾碼。有效位數會指定為一系列的數位，後面接著句點，然後是選擇性的數位序列，代表數位的小數部分。例如：

```
18.46
38.
```

如果有指數，指數會以 10 的次方指定數字的大小，如下列範例所示：

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

您可以使用或指定指數 e E，其意義相同，後面接著選擇性的正負號 (+ 或-) 和一連串的數位。如果有指數，18E0 之類的整數就不需要尾端的小數點。

浮點數常值預設為類型 `double`。使用尾碼 f l F L (尾碼不區分大小寫)，可以將常值指定為 `float` 或 `long double`。

雖然 `long double` 和 `double` 具有相同的標記法，但它們並不是相同的類型。例如，您可以有多載的函式，例如

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

及

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

## 布林常值

布林常值為 `true` 和 `false`。

## 指標常值 (C++11)

C++ 引進 `nullptr` 常值來指定以零初始化的指標。在可移植程式碼中，`nullptr` 應該使用，而非整數類型零或宏 (例如) `NULL`。

## 二進位常值 (C++14)

使用 `0B` 或 `0b` 前置詞，後接一連串的 1 和 0，即可指定二進位常值：

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

請避免使用常值做為「幻方常數」。

您可以直接在運算式和陳述式中使用常值，雖然這不一定是好的程式設計做法：

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

在上述範例中，較好的作法是使用可傳達明確意義的命名常數，例如 "MAXIMUM\_ERROR\_THRESHOLD"。而且，如果使用者看到傳回值「成功」，則使用命名字串常數可能比較好。您可以將字串常數保留在檔案中的單一位置，該檔案可以當地語系化為其他語言。使用命名常數可協助您自己和其他人瞭解程式碼的意圖。

## 另請參閱

[語彙慣例](#)

[C++ 字串常值](#)

[C++ 使用者定義常值](#)

# 字串和字元常值 ( C++ )

2020/11/2 • [Edit Online](#)

C++ 支援各種字串和字元類型，並提供方法來表示所有這些類型的常值。在原始程式碼中，您可以使用字元集表示字元和字串常值的內容。通用字元名稱和逸出字元允許您只使用基本來源字元集表示任何字串。原始字串常值可讓您避免使用逸出字元，而且可用來表示所有類型的字串常值。您也可以建立 `std::string` 常值，而不需要執行額外的結構或轉換步驟。

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

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

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

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

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

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

    // Combining raw string literals with standard s-suffix
    auto S5 = R"("Hello \ world")"s; // std::string from a raw const char*
    auto S6 = u8R"("Hello \ world")"s; // std::string from a raw const char*, encoded as UTF-8
    auto S7 = LR"("Hello \ world")"s; // std::wstring from a raw const wchar_t*
    auto S8 = uR"("Hello \ world")"s; // std::u16string from a raw const char16_t*, encoded as UTF-16
    auto S9 = UR"("Hello \ world")"s; // std::u32string from a raw const char32_t*, encoded as UTF-32
}
```

字串常值可以沒有前置詞，或者以 `u8`、`L`、`u` 和 `U` 前置詞分別表示半形字元 (單一位元組或多位元組)、UTF-8、全形字元 (UCS-2 或 UTF-16)、UTF-16 和 UTF-32 編碼。原始字串常值可以有 `R`、`u8R`、`LR`、和前置詞，適用于 `uR`、`UR` 這些編碼的原始版本。若要建立暫存或靜態 `std::string` 值，您可以使用字串常值或具有尾碼的原始字串常值 `s`。如需詳細資訊，請參閱下面的「[字串常值](#)」一節。如需基本來源字元集、通用字元名稱，以及使用原始程式碼中的擴充字碼頁字元的詳細資訊，請參閱[character sets](#)。

# 字元常值

「字元常值」(character literal) 是由常數字元所組成。它是以單引號括住的字元來表示。字元常值有五種類型：

- 類型的一般字元常值 `char`，例如 `'a'`
- 類型的 UTF-8 字元常值 `char` (`char8_t` C ++ 20)，例如 `u8'a'`
- 類型的寬字元常值 `wchar_t`，例如 `L'a'`
- 類型的 UTF-16 字元常值 `char16_t`，例如 `u'a'`
- 類型的 UTF-32 字元常值 `char32_t`，例如 `U'a'`

用於字元常值的字元可以是任何字元，但保留字元反斜線(`\`)、單引號(`'`)或分行符號除外。使用逸出序列可指定保留字元。只要類型大到足以容納字元，就可以使用通用字元名稱指定字元。

## 編碼

字元常值會根據其前置詞以不同的方式進行編碼。

- 不含前置詞的字元常值是一般的字元常值。包含可以在執行字元集中表示之單一字元、轉義順序或通用字元名稱的一般字元常值，其值會等於其在執行字元集中的編碼數值。包含一個以上字元、escape 序列或通用字元名稱的一般字元常值是一個多重字元常值。無法在執行字元集中表示的多重字元常值或一般字元常值具有類型 `int`，且其值為執行定義。如需 MSVC，請參閱下面的Microsoft 專有章節。
- 開頭為前置詞的字元常 `L` 值是寬字元常值。包含單一字元、escape 序列或通用字元名稱的寬字元常值，其值會等於其在執行寬字元集中的編碼數值，除非該字元常值在執行寬字元集中沒有任何標記法，在此情況下，該值為「執行定義」。包含多個字元、escape 序列或通用字元名稱的寬字元常值，是由實作為定義的。如需 MSVC，請參閱下面的Microsoft 專有章節。
- 開頭為前置詞的字元常 `u8` 值是 utf-8 字元常值。包含單一字元、escape 序列或通用字元名稱的 UTF-8 字元常值，其值若可由單一 UTF-8 程式碼單位(對應于 C0 控制項和基本拉丁 Unicode 區塊)來表示，則其值會等於其 ISO 10646 碼點值。如果這個值不能以單一 UTF-8 程式碼單位表示，則程式格式不正確。包含一個以上字元、escape 序列或通用字元名稱的 UTF-8 字元常值格式不正確。
- 開頭為前置詞的字元常 `u` 值是 utf-16 字元常值。包含單一字元、escape 序列或通用字元名稱的 UTF-16 字元常值，如果可以由單一 UTF-16 程式碼單位(對應至基本多語言平面)表示，則其值會等於其 ISO 10646 代碼點值。如果這個值不能以單一 UTF-16 程式碼單位表示，則程式格式不正確。包含一個以上字元、escape 序列或通用字元名稱的 UTF-16 字元常值格式不正確。
- 開頭為前置詞的字元常 `U` 值是 UTF-32 字元常值。包含單一字元、escape 序列或通用字元名稱的 UTF-32 字元常值，其值會等於其 ISO 10646 程式碼點值。包含一個以上字元、escape 序列或通用字元名稱的 UTF-32 字元常值格式不正確。

## Escape 序列

逸出序列有三種：簡單的、八進位和十六進位。Escape 序列可以是下列其中一個值：

新行字元	<code>\n</code>
反斜線	<code>\\"</code>
水平定位字元	<code>\t</code>
問號	? 或 <code>\?</code>

【	】
垂直定位字元	\&
單引號	\'
退格鍵	\位元組
雙引號	\"
歸位字元	\r
null 字元	\0
換頁字元	\f
八進位	\ooo
警示 (鈴聲)	\a
十六進位	\xhhh

八進位的逸出序列是反斜線，後面接著一連串的一到三個八進位數位。如果比第三個數字更早發生，八進位逸出序列會在不是八進位數位的第一個字元終止。可能的最大八進位值為 `\377`。

十六進位的逸出序列是反斜線，後面接著字元 `x`，後面接著一或多個十六進位數位的序列。系統會忽略前置零。在一般或 `u8` 首碼的字元常值中，最高的十六進位值是 `0xFF`。在 `L` 前置詞或 `u` 前置詞的全形字元常值中，最高的十六進位值是 `0xFFFF`。在 `U` 前置詞的全形字元常值中，最高的十六進位值是 `0xFFFFFFFF`。

這個範例程式碼會顯示一些使用一般字元常值的逸出字元範例。相同的轉義順序語法對其他字元常數型別而言是有效的。

```
#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
*/
```

反斜線字元 (`\`) 是放在行尾時的行接續字元。如果您想要讓反斜線字元顯示為字元常值，您必須在資料列中輸

入兩個反斜線( `\\"` )。如需行接續字元的詳細資訊，請參閱 [Phases of Translation](#)。

#### Microsoft 專有

若要從窄多重字元常值建立值，編譯器會將單引號之間的字元或字元序列轉換成32位整數內的8位值。常值中的多個字元，會視需要從高序位到低序位填入對應的位元組。然後編譯器會依照一般規則，將整數轉換成目的地類型。例如，若要建立 `char` 值，編譯器會採用低序位位元組。若要建立 `wchar_t` 或 `char16_t` 值，編譯器會採用低序位字組。如果任何位元設定高出指定的位元組或字組，編譯器會警告結果遭截斷。

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

會將超過三位數的八進位 escape 序列視為3位數的八進位序列，後面接著後續的數位做為多重字元常值中的字元，這可能會產生令人驚訝的結果。例如：

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

似乎包含非八進位字元的 Escape 序列會評估為最後一個八進位字元的八進位序列，後面接著其餘字元做為多重字元常值中的後續字元。如果第一個非八進位字元是十進位數，則會產生警告 C4125。例如：

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

值大於的八進位逸出序列 [\377](#) 會導致錯誤 C2022：'值-十進位':字元太大。

具有十六進位和非十六進位字元的 escape 序列會評估為多重字元常值，其中包含最多到最後一個十六進位字元的十六進位逸出序列，後面接著非十六進位字元。不包含十六進位數位的十六進位逸出序列會導致編譯器錯誤 C2153：「十六進位常值至少必須有一個十六進位數位」。

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

如果在前面加上的寬字元常 `L` 值包含多重字元序列，則會從第一個字元取得該值，而編譯器會引發警告 C4066。系統會忽略後續的字元，不同于對等的一般多重字元常值的行為。

```
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
```

Microsoft 專屬章節將于此結束。

#### 通用字元名稱

在字元常值和原生(非原始)的字串常值中，任何字元都可能使用通用字元名稱表示。通用字元名稱是由前置片語成 `\u`，後面接著八位數的 unicode 程式碼片段，或前置詞 `\u` 後面接著四位數的 unicode 程式碼點。所有的八位數或四位數，都一定要出現才是格式正確的通用字元名稱。

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

### Surrogate 字組

通用字元名稱無法編碼代理程式碼點範圍 D800-DFFF 中的值。至於 Unicode surrogate 字組，請使用 `\UNNNNNNNN` 指定通用字元名稱，這裡的 NNNNNNNN 為字元的八位數字碼指標。必要時，編譯器會產生代理配對。

過去在 C++03 中，語言只允許由其通用字元名稱所代表的字元子集，以及未實際代表任何有效 Unicode 字元的一些通用字元名稱。此錯誤已在 C++11 標準中修正。在 C++11 中，字元和字串常值及識別項可以使用通用字元名稱。如需通用字元名稱的詳細資訊，請參閱 [Character Sets](#)。如需 Unicode 的詳細資訊，請參閱 [Unicode](#)。如需 Surrogate 字組的詳細資訊，請參閱 [Surrogate 字組和補充字元](#)。

## 字串常值

字串常值代表構成 null 結束字串的一連串字元。這些字元必須使用雙引號括起來。有下列字串常值類型：

### 窄字串常值

窄字串常值是類型的非前置詞、雙引號分隔、null 結束陣列 `const char[n]`，其中 n 是陣列的長度（以位元組為單位）。半形字串常值可能包含任何圖形字元，但雙引號 ("")、反斜線 (\) 或新行字元除外。半形字串常值也可能包含上文列出的逸出序列，以及符合一個位元組大小的通用字元名稱。

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

### UTF-8 編碼的字串

UTF-8 編碼的字串是 u8 前置詞、以雙引號分隔、以 null 結束的類型陣列 `const char[n]`，其中 n 是已編碼陣列的長度（以位元組為單位）。u8 前置字串常值可以包含任何圖形字元，但雙引號 ("")、反斜線 (\) 或新行字元除外。u8 前置字串常值也可以包含上列逸出序列，以及任何通用字元名稱。

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

### 寬字元串常值

寬字元串常值是以 null 結束的常數陣列，`wchar_t` 前面會加上 'L'，並包含雙引號 ("")、反斜線 (\) 或分行符號以外的任何圖形字元。全形字串常值可以包含上列逸出序列，以及任何通用字元名稱。

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

### char16\_t 和 char32\_t (C++11)

C++11 引進可攜 `char16_t` (16位 unicode) 和 `char32_t` (32位 unicode) 字元類型：

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

### 原始字串常值 (C++11)

原始字串常值是以 null 結束的陣列（任何字元類型），其中包含任何圖形字元，包括雙引號 ("")、反斜線 (\) 或

換行字元。原始字串常值通常用於使用字元類別的規則運算式，以及 HTML 字串和 XML 字串。如需範例，請參閱下列文章：[Bjarne Stroustrup 的 C++11 常見問題集](#)。

```
// represents the string: An unescaped \ character
const char* raw_narrow = R"(An unescaped \ character)";
const wchar_t* raw_wide = LR"(An unescaped \ character)";
const char* raw_utf8 = u8R"(An unescaped \ character)";
const char16_t* raw_utf16 = uR"(An unescaped \ character)";
const char32_t* raw_utf32 = UR"(An unescaped \ character);
```

「分隔符號」是一種使用者定義的序列，最多16個字元，緊接在原始字串常值的左括弧前面，然後緊接在其右括弧後面。例如，`R"abc(Hello"\()abc"` 中的分隔符號順序是 `abc`，字串內容是 `Hello"\(`。您可以使用分隔符號，釐清包含雙引號和括號的原始字串。這個字串常值會造成編譯器錯誤：

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

但是分隔符號解決這個錯誤：

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

您可以在來源中，建立包含分行符號(而不是逸出字元)的原始字串常值：

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

### std::string 常值(c + + 14)

`std::string` 常值是使用者定義常值的標準程式庫執行(如下所示)，以表示為 `"xyz"s` (`s` 尾碼)。這種字串常值 `std::string`、`std::wstring`、`std::u32string`、`std::u16string` 會根據指定的前置詞，產生類型為、、或的暫存物件。如果未使用前置詞，`std::string` 則會產生。`L"xyz"s` 產生 `std::wstring`。`u"xyz"s` 會產生 `std::u16string`，並 `u"xyz"s` 產生 `std::u32string`。

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

`s` 尾碼也可以用於原始字串常值：

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

`std::string` 常值是在標頭檔的命名空間中定義 `std::literals::string_literals <string>`。因為 `std::literals::string_literals`、和 `std::literals` 都宣告為 **內嵌命名空間**，所以會 `std::literals::string_literals` 自動將視為直接屬於命名空間 `std`。

### 字串常值的大小

針對 ANSI `char*` 字串和其他單一位元組編碼(但不是 utf-8)，字串常值的大小(以位元組為單位)是結束的 null 字元的字元數加1。對於所有其他字串類型，大小不會與字元數完全相關。UTF-8 使用最多四個 `char` 元素來編碼某

些程式代碼單位，`char16_t` 或 `wchar_t` 編碼為 utf-16 可能會使用兩個元素（總共四個位元組）來編碼單一程式代碼單位。本例顯示全形字串常值以位元組為單位的大小：

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

請注意，`strlen()` 和 `wcslen()` 不包括結束 null 字元的大小，其大小等於字串類型的專案大小：或字串上的一個位元組 `char*` `char8_t*`、兩個位元組的 `wchar_t*` 或 `char16_t*` 字串，以及字串的四個位元組 `char32_t*`。

字串常值的最大長度是 65535 個位元組。這個限制適用於半形字串常值和全形字串常值。

## 修改字串常值

因為字串常值（不包括 `std::string` 常數）是常數，所以嘗試修改它們（例如）`str[2] = 'A'` 會造成編譯器錯誤。

### Microsoft 專有

在 Microsoft C++ 中，您可以使用字串常值來初始化非 `const` 或的指標 `char` `wchar_t`。C99 程式碼允許此非 `const` 初始化，但在 C++ 98 中已被取代，並已在 C++ 11 中移除。嘗試修改字串造成存取違規，如此範例所示：

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

當您設定 `/Zc:strictStrings`（停用字串常數型別轉換）編譯器選項時，如果字串常值轉換為非 `const` 字元指標，您可以讓編譯器發出錯誤。我們建議使用標準相容的可攜式程式碼。使用關鍵字來宣告字串常值初始化的指標也是很好的作法 `auto`，因為它會解析為正確的(`const`)型別。例如，這個程式碼範例會攔截在編譯時期嘗試寫入字串常值的動作：

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

在某些情況下，可以結合相同字串常值來節省可執行檔中的空間。在字串常值共用中，編譯器會讓特定字串常值的所有參考指向記憶體內部相同位置，而不是讓每個參考都指向字串常值的個別執行個體。若要啟用字串共用，請使用 `/GF` 編譯器選項。

Microsoft 專屬章節將于此結束。

## 串連相鄰的字串常值

會串連相鄰的全形或半形字串常值。這個宣告：

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

等同於此宣告：

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

以及這個宣告：

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

使用內嵌十六進位逸出程式碼指定字串常值可能造成未預期的結果。下列範例是要建立包含 ASCII 5 字元的字串常值，後面接著字元 f、i、v 和 e：

```
"\x05five"
```

實際結果是十六進位 5F (即底線字元的 ASCII 碼), 後面接 i、v、e 字元。若要取得正確的結果, 您可以使用下列其中一個 escape 序列:

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

`std::string` 常值 (因為它們是 `std::string` 類型) 可以與 `+` 為類型定義的運算子串連 `basic_string`。它們的串連方式也與相鄰字串常值相同。在這兩種情況下, 字串編碼和後置詞必須相符:

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

### 含通用字元名稱的字串常值

原生 (非原始) 的字串常值可以使用通用字元名稱來代表任何字元, 只要通用字元名稱可以編碼為字串類型中的一或多個字元。例如, 代表擴充字元的通用字元名稱無法使用 ANSI 字碼頁以窄字串編碼, 但是可以在某些多位元組字碼頁或 UTF-8 字串或寬字元串中, 以窄字串編碼。在 c++11 中, Unicode 支援是由 `char16_t*` 和 `char32_t*` 字串類型擴充:

```
// ASCII smiling face  
const char*     s1 = ":-)";  
  
// UTF-16 (on Windows) encoded WINKING FACE (U+1F609)  
const wchar_t*   s2 = L"\u0001F609 is ;-)";  
  
// UTF-8 encoded SMILING FACE WITH HALO (U+1F607)  
const char*     s3 = u8"\u0001F607 is O:-)";  
  
// 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-)";
```

## 另請參閱

[字元集](#)

[數值、布林值和指標常值](#)

[使用者定義常值](#)

# 使用者定義常值

2020/11/2 • [Edit Online](#)

C++ 中有六個主要的常值分類：整數、字元、浮點、字串、布林值和指標。從 C++ 11 開始，您可以根據這些類別定義您自己的常值，以提供一般慣用語的語法快捷方式並增加型別安全。例如，假設您有一個 `Distance` 類別。您可以定義公里的常值和另一個代表英里的文字，並藉由撰寫來鼓勵使用者明確瞭解測量單位：`auto d = 42.0_km` 或 `auto d = 42.0_mi`。使用者定義的常值沒有任何效能優勢或缺點；它們主要是為了方便或編譯時間類型推斷。標準程式庫具有的使用者定義常值 `std::string`、適用於 `std::complex`，以及標頭中時間和持續時間作業的單位 `<chrono>`：

```
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
```

## 使用者定義常值運算子簽章

您可以使用下列其中一種形式，在命名空間範圍定義運算子 ""，以執行使用者定義的常值：

```
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 "" _j(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
```

先前範例中的運算子名稱都是您所提供的名稱的預留位置；不過，需要前置底線。（只允許標準程式庫定義不含底線的常值）。傳回類型是您自訂轉換或常值所執行之其他作業的位置。此外，這些運算子中的任何一個都可以定義為 `constexpr`。

## 處理後的常值

在原始程式碼中，任何常值（不論使用者是否定義）基本上是一連串的英數位元，例如 `101`、或、或 `54.7`、`"hello"`、`true`。編譯器會將序列解讀為整數、浮點數、`const char*` 字串等等。使用者定義的常值，它會接受指派給常值的任何型別做為輸入，非正式稱為處理後常值。所有上面的運算子（`_r` 和 `_t` 除外）都是處理後的常值。例如，常值 `42.0_km` 會繫結至簽章與 `_b` 類似的 `_km` 運算子，常值 `42_km` 則繫結至簽章與 `_a` 類似的運算子。

下列範例示範使用者定義常值如何鼓勵呼叫端明確指定其輸入。若要建構 `Distance`，使用者必須使用適當的使用者定義常值來明確指定公里或英哩。您可以用其他方式達到相同的結果，但使用者定義的常值比替代方案的詳細資訊還少。

```

// 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;
}

```

常值數位必須使用十進位。否則，此數位會被視為整數，而類型也不會與運算子相容。對於浮點輸入，類型必須是 `long double`，而對於整數類型則必須是 `long long`。

## 原始常值

在未經處理的使用者定義常值中，您定義的運算子會接受常值做為 `char` 值的序列。您必須以數位或字串或其他類型來解讀該序列。在此頁面上先前顯示的運算子清單中，`_r` 和 `_t` 可以用來定義原始常值：

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

您可以使用原始常值來提供輸入序列的自訂轉譯，這與編譯器的一般行為不同。例如，您可以定義將序列

4.75987 轉換為自訂 Decimal 類型的常值，而不是 IEEE 754 浮點類型。原始常值(例如處理後常值)也可以用於輸入序列的編譯階段驗證。

### 範例: 原始常值的限制

原始常值運算子和常值運算子樣板只適用於整數和浮點使用者定義常值(如下列範例所示):

```
#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;
}

```

```

operator "" _dump(unsigned long long int) : ===>42<===
operator "" _dump(long double) : ===>3.141593<===
operator "" _dump(long double) : ===>3139999999999998506827776.000000<===
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<===

```

# 基本概念 ( C++ )

2020/3/25 • [Edit Online](#)

本節說明瞭解C++的重要概念。C 程式設計人員將熟悉其中許多概念，但有一些細微的差異可能會導致非預期的程式結果。本文包含下列主題：

- [C++ 類型系統](#)
- [範圍](#)
- [轉譯單位和連結](#)
- [main 函式和命令列引數](#)
- [程式終止](#)
- [左值和右值](#)
- [暫存物件](#)
- [對齊](#)
- [簡單、標準版面配置和 POD 類型](#)

## 另請參閱

[C++ 語言參考](#)

# C++ 類型系統

2020/11/2 • [Edit Online](#)

類型的概念在 C++ 中非常重要。每個變數、函式引數和函式傳回值都必須有類型才能編譯。此外，在評估之前，編譯器會以隱含的方式指定每一個運算式（包括常值）的類型。某些類型的範例包括 `int` 儲存整數值、儲存 `double` 浮點值（也稱為純量資料類型）`scalar` 或標準程式庫類別 `std::basic_string` 以儲存文字。您可以藉由定義或來建立自己的類型 `class` `struct`。此類型會指定配置給變數（或運算式結果）的記憶體數量、可在該變數中存放的值種類、這些值（位元模式）的解譯方式，以及可對其執行的作業。本文包含 C++ 類型系統主要功能的簡略概觀。

## 術語

**變數**：資料數量的符號名稱，讓名稱可在其定義所在的程式碼範圍內，用來存取它所參考的資料。在 C++ 中，變數通常用來參考純量資料類型的實例，而其他類型的實例通常稱為物件。

**物件**：為了簡化和一致性，本文使用「物件」一詞來參考類別或結構的任何實例，而在一般意義中使用時，則包含所有類型，甚至是純量變數。

**POD 類型（一般舊資料）**：C++ 中的這個非正式資料類型類別是指純量類型（請參閱基本類型一節）或 **POD 類別**。POD 類別的靜態資料成員同時也是 POD，沒有使用者定義的建構函式、使用者定義解構函式或使用者定義的指派運算子。此外，POD 類別沒有虛擬函式、沒有基底類別，也沒有私用或受保護的非靜態資料成員。POD 類型通常用於外部資料交換，例如以 C 語言撰寫的模組（其中只有 POD 類型）。

## 指定變數和函式類型

C++ 是強型別語言，而且也是靜態類型；每個物件都有類型，而且該類型永遠不會變更（不會與靜態資料物件混淆）。當您在程式碼中宣告變數時，您必須明確指定其類型，或使用 `auto` 關鍵字指示編譯器從初始化運算式推算類型。當您在程式碼中宣告函式時，您必須指定每個引數的類型及其傳回值，或者，如果函式 `void` 未傳回任何值，則為。例外情況是在您使用函式樣板，可允許任意類型的引數。

在您初次宣告變數之後，就不能再變更其類型。不過，您可以將變數值或函式的傳回值複製到另一個不同類型的變數。這類作業稱為「類型轉換」，有時是必要的，但也是資料遺失或 `incorrectness` 的潛在來源。

當您宣告 POD 類型的變數時，強烈建議您將其初始化，也就是指定其初始值。在您初始化變數以前，變數都會包含由先前遺留在該記憶體位置之任何位元所構成的「垃圾」值。這是一個要記住的 C++ 重要層面，如果您是從另一個用來處理初始化的語言轉換而來時則更是如此。宣告非 POD 類別類型的變數時，建構函式會處理初始化。

下列範例示範一些簡單的變數宣告，這些宣告各有一些描述。這個範例也會示範編譯器如何使用類型資訊允許或不允許對變數進行某些後續作業。

```

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

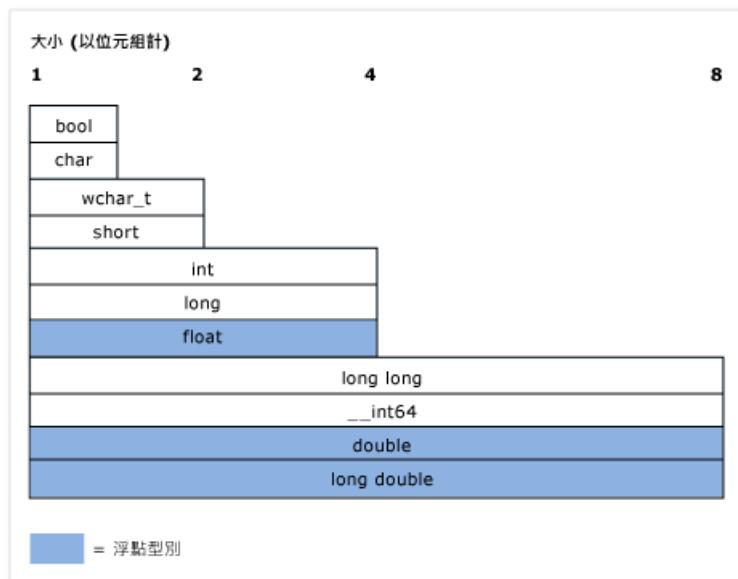
```

## 基本 (內建) 類型

有別於某些程式語言, C++ 並沒有可衍生出所有其他類型的通用基底類型。此語言包含許多基本類型, 也稱為**內建類型**。這包括數數值型別, 例如 `int`、`double`、`long`、`bool`, 以及 `char` `wchar_t` 分別為 ASCII 和 UNICODE 字元的和類型。最重要的基本類型(`bool`、`double`、`wchar_t` 和相關類型除外)都有 `unsigned` 版本, 可修改變數可以儲存的值範圍。例如, `int` 儲存32位帶正負號的整數, 可以代表從-2147483648 到2147483647 的值。`unsigned int` 也儲存為32位的, 可以儲存0到4294967295 之間的值。每個案例中的可能值總數都相同; 只有範圍不同。

基本類型是由編譯器辨識, 其內建規則會控制可對這些類型執行哪些作業, 以及如何轉換成其他基本類型。如需內建類型及其大小和數值限制的完整清單, 請參閱[內建類型](#)。

下圖顯示 Microsoft c++ 執行中內建類型的相對大小:



下表列出最常使用的基本類型, 以及它們在 Microsoft c++ 中的大小:

II	II	II
<code>int</code>	4 個位元組	整數值的預設選項。
<code>double</code>	8 個位元組	浮點值的預設選項。
<code>bool</code>	1 個位元組	表示可以是 <code>true</code> 或 <code>false</code> 的值。

<code>char</code>	1 個位元組	使用較舊 C-Style 字串或 <code>std::string</code> 物件中永遠不需要轉換成 UNICODE 之的 ASCII 字元。
<code>wchar_t</code>	2 個位元組	表示可能以 UNICODE 格式 (在 Windows 上為 UTF-16, 而其他作業系統可能不同) 編碼的「寬」字元值。這是用於 <code>std::wstring</code> 類型字串的字元類型。
<code>unsigned char</code>	1 個位元組	C++ 沒有內建的 byte 類型。用 <code>unsigned char</code> 來代表位元組值。
<code>unsigned int</code>	4 個位元組	位元旗標的預設選項。
<code>long long</code>	8 個位元組	表示極大的整數值。

其他 C++ 的執行可能會針對特定數數值型別使用不同的大小。如需 C++ 標準所需之大小和大小關聯性的詳細資訊，請參閱[內建類型](#)。

## void 類型

`void` 類型是一種特殊類型；您無法宣告類型的變數 `void`，但可以宣告類型的變數 `void *` (的指標 `void`)，這在配置原始(不具類型)的記憶體時有時是必要的。不過，的指標 `void` 並不是型別安全的，而且通常不建議在現代 C++ 中使用。在函式宣告中，傳回 `void` 值表示函數不會傳回值，這是常見且可接受的用法 `void`。雖然 C 語言所需的函式在參數清單中會宣告零個參數(例如)，但 `void fou(void)` 在現代 C++ 中並不建議使用這個做法，而且應該進行宣告 `fou()`。如需詳細資訊，請參閱[類型轉換和型別安全](#)。

## 常數類型限定詞

任何內建或使用者定義的類型都可以由 `const` 關鍵字限定。此外，成員函式可以是限定的，甚至是多載 `const const`。類型的值在 `const` 初始化之後就無法修改。

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

`const` 限定詞在函式和變數宣告中廣泛使用，而「`const` 正確性」是 C++ 中的重要概念，基本上，這表示要使用 `const` 來保證在編譯時期，這些值不會不慎修改。如需詳細資訊，請參閱[const](#)。

`const` 類型與其非 `const` 版本不同，例如，`const int` 是來自的不同類型 `int`。`const_cast` 當您必須從變數中移除常數性質時，您可以在很罕見的情況下使用 C++ 運算子。如需詳細資訊，請參閱[類型轉換和型別安全](#)。

## 字串類型

嚴格來說，C++ 語言沒有內建的字串類型。`char` 並 `wchar_t` 儲存單一字元 - 您必須宣告這些類型的陣列，以近似字串，並將結束的 null 值(例如 ASCII)新增 `'\0'` 至最後一個有效字元之後的陣列元素(也稱為 C 樣式字串)。C-Style 字串需要撰寫更多程式碼或使用外部字串公用程式庫函式。但是在現代 C++ 中，我們有標準程式庫類型 `std::string` (適用於 8 位 `char` 類型的字元字串) 或 `std::wstring` (適用於 16 位 `wchar_t` 類型的字元字串)。這些 C++ 標準程式庫容器可以視為原生字串類型，因為它們是包含在任何相容 C++ 組建環境中的標準程式庫的一部分。只要使用 `#include <string>` 指示詞，即可在您的程式中使用這些類型。(如果您使用 MFC 或 ATL，`cstring` 類別也可以使用，但不是 C++ 標準的一部分)。強烈建議您不要使用以 null 終止的字元陣列(先前提到的 C 樣式字

串)。

## 使用者定義型別

當您定義 `class`、`struct`、或時 `union` `enum`，該結構會用於程式碼的其餘部分，如同它是基本類型。它在記憶體中的大小已知，而且套用了有關其在編譯時間檢查、執行階段和程式存留期之使用方式的特定規則。基本內建類型和使用者定義類型之間的主要差異如下：

- 編譯器沒有使用者定義類型的內建知識。它會在編譯過程中第一次遇到定義時學習類型。
- 您會藉由定義 (透過多載) 適當的運算子做為類別成員或非成員函式，指定對類型執行哪些作業，以及如何轉換為其他類型。如需詳細資訊，請參閱[函數多載](#)

## 指標類型

可追溯回到 C 語言的最早版本，C++ 會繼續讓您使用特殊的宣告子(星號)來宣告指標類型的變數 `*`。指標類型會將在儲存實際資料值之位置的位址儲存在記憶體中。在現代 C++ 中，這些稱為原始指標，可透過特殊運算子 `*` (星號) 或 `->` (使用大於的虛線) 在您的程式碼中存取。這稱為「取值」，而您使用哪一個，取決於您要對純量的指標或物件中成員的指標取值。處理指標類型一直以來都是 C 及 C++ 程式開發最具挑戰性和令人困惑的一面。本章節會概述一些事實和作法，以協助使用原始指標(如果您想要的話)，但在現代 C++ 中，由於智慧型指標的演進(這會在本節結尾討論)，因此不再需要(或建議)使用原始指標來取得物件擁有權。使用原始指標觀察物件仍然有用且安全，但若要將其用於物件擁有權，就必須小心使用，並非常仔細地考量如何建立和終結所擁有的物件。

您首先應該知道的事就是，宣告原始指標變數時只會配置，在儲存該指標在被取值時所參考之記憶體位置位址時所需的記憶體。尚未配置資料值本身的記憶體配置(也稱為備份存放區)。換句話說，宣告原始指標變數，即是在建立記憶體位址變數，而不是實際資料變數。在確定變數包含可用於備份存放區的有效位址之前就先取值指標變數，會導致程式中產生未定義的行為(通常是嚴重錯誤)。下列範例示範這種錯誤：

```
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.
```

這個範例會對指標類型取值，但不配置任何記憶體的來儲存指派給它的實際整數資料或有效記憶體位址。下列程式碼示範這些錯誤：

```
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".
```

更正的程式碼範例會使用本機堆疊記憶體，建立 `pNumber` 所指向的備份存放區。我們為了簡單起見使用基本類型。實際上，指標的備份存放區最常是使用者定義的型別，這些型別是使用 `heap` 關鍵字運算式(在 C 樣式程式設計中，使用較舊的 C 執行時間程式庫函式)，以動態方式配置在稱為堆積(或「free store」)的記憶體區域中 `new` `malloc()`。一旦配置之後，這些變數通常稱為「物件」，特別是當它們是以類別定義為基礎時。使用配置的記憶體 `new` 必須由對應的 `delete` 語句刪除(或者，如果您使用函式 `malloc()` 來配置它，則為 C 執行時間函式 `free()`)。

不過，很容易忘記刪除動態配置的物件，特別是在複雜的程式碼中，這會造成資源錯誤，稱為記憶體流失。因此，強烈建議您不要在現代 C++ 使用原始指標。將原始指標包裝在智慧型指標中幾乎是最好的作法，這會在叫用其析構函式時自動釋放記憶體(當程式碼超出智慧型指標的範圍時);藉由使用智慧型指標，您幾乎可以消除 C++ 程式中的整個 bug 類別。下列範例中，假設 `MyClass` 是具有公用方法 `DoSomeWork()` 的使用者定義類型

```
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.
```

如需智慧型指標的詳細資訊，請參閱[智慧型指標](#)。

如需指標轉換的詳細資訊，請參閱[類型轉換和型別安全](#)。

如需一般指標的詳細資訊，請參閱[指標](#)。

## Windows 資料類型

在 C 和 C++ 的傳統 Win32 程式設計中，大部分函式會使用 Windows 特定的 `typedef` 和 `#define` 宏(定義于 `windef.h`)來指定參數類型和傳回值。這些 Windows 資料類型大部分都只是指定給 C/C++ 內建類型的特殊名稱(別名)。如需這些 `typedef` 和預處理器定義的完整清單，請參閱[Windows 資料類型](#)。其中一些 `typedef`(例如 `HRESULT` 和 `LCID`)是有用且描述性的。其他專案(例如 `INT`)沒有特殊意義，而且只是基本 C++ 類型的別名。其他 Windows 資料類型有從 C 程式設計和 16 位元處理器時代保留下來的名稱，在現代硬體或作業系統上並無用處或意義。Windows 執行階段程式庫也有相關聯的特殊資料類型，列為[Windows 執行階段基底資料類型](#)。在現代 C++ 中，一般的方針就是，除非 Windows 類型傳達有關如何解譯值的額外涵義，否則優先使用 C++ 基本類型。

## 詳細資訊

如需 C++ 類型系統的詳細資訊，請參閱下列主題：

### 實數值型別

描述實數值型別，以及與使用方式相關的問題。

### 類型轉換和型別安全

描述一般類型轉換問題並顯示如何避免這些問題。

## 另請參閱

[歡迎回到 C++](#)

[C++ 語言參考](#)

[C++ 標準程式庫](#)

# 範圍 (C++)

2020/11/2 • [Edit Online](#)

當您宣告程式元素(例如類別、函式或變數)時，其名稱只能是「可見」，並用於程式的特定部分。顯示名稱的內容稱為其範圍。例如，如果您在函式內宣告變數 `x`，`x` 則只有在該函式主體內才會顯示。它具有區域範圍。您的程式中可能會有相同名稱的其他變數；只要它們位於不同的範圍內，它們就不會違反一個定義規則，也不會引發錯誤。

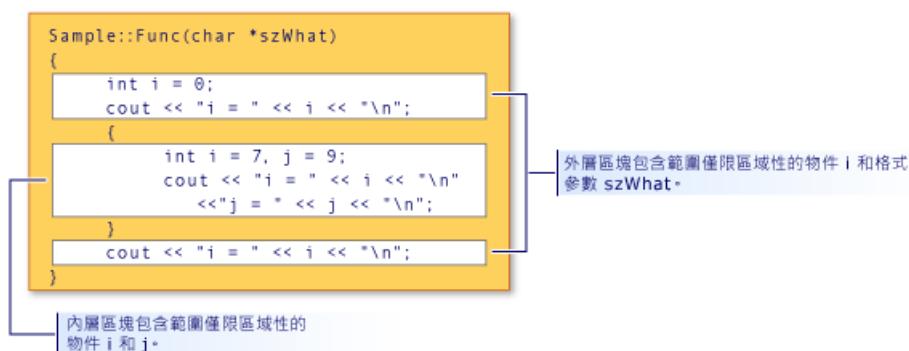
若是自動的非靜態變數，範圍也會決定在程式記憶體中建立和終結它們的時間。

範圍有六種：

- **全域範圍** 全域名稱是在任何類別、函式或命名空間之外宣告的。不過，在 C++ 中，即使這些名稱與隱含的全域命名空間同時存在也一樣。全域名稱的範圍會從宣告的點延伸到宣告它們的目的檔案尾。若為全域名稱，可見度也會受到連結規則的規範，判斷程式中其他檔案的名稱是否可見。
- **命名空間範圍** 在 [命名空間](#) 中，于任何類別或列舉定義或函式區塊外宣告的名稱，在命名空間結尾處可見。命名空間可在不同檔案的多個區塊中定義。
- **區域範圍** 在函式或 lambda 內宣告的名稱(包括參數名稱)具有區域範圍。它們通常稱為「區域變數」。只有從其宣告點到函式或 lambda 主體的結尾時，才會顯示它們。區域範圍是一種區塊範圍，本文稍後會加以討論。
- **類別範圍** 類別成員的名稱具有類別範圍，不論宣告點為何，它都會在整個類別定義中擴充。類別成員存取範圍是由 `public`、`private` 和關鍵字進一步控制 `protected`。只能使用成員選取運算子()來存取公用或受保護的成員。或 `->` 或成員指標運算子(`.*` 或 `->*`)。
- **語句範圍** 在 `for`、、或語句中宣告的名稱，會在 `if` `while` `switch` 語句區塊的結尾處顯示。
- **函數範圍** [標籤](#) 具有函式範圍，這表示它在函式主體中可見，甚至是在其宣告點之前。函式範圍可以在宣告 `goto cleanup` 標籤之前撰寫語句 `cleanup`。

## 隱藏名稱

您可以在封閉的區塊中宣告名稱，以隱藏該名稱。下圖是在內層區塊中重新宣告 `i`，從而隱藏外層區塊範圍中與 `i` 相關的變數。



封鎖範圍和隱藏名稱

本圖所示的程式輸出為：

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

#### NOTE

`szWhat` 引數被視為在函式的範圍中。因此，其會被視為已在函式的最外層區塊中宣告。

## 隱藏類別名稱

您可以藉由宣告函式、物件、變數或相同範圍中的列舉程式，隱藏類別名稱。不過，在前面加上關鍵字時，仍然可以存取類別名稱 `class`。

```
// hiding_class_names.cpp  
// compile with: /EHsc  
#include <iostream>  
using namespace std;  
  
// Declare class Account at global scope.  
class Account  
{  
public:  
    Account( double InitialBalance )  
        { balance = InitialBalance; }  
    double GetBalance()  
        { return balance; }  
private:  
    double balance;  
};  
  
double Account = 15.37;           // Hides class name Account  
  
int main()  
{  
    class Account Checking( Account ); // Qualifies Account as  
                                       // class name  
  
    cout << "Opening account with a balance of: "  
        << Checking.GetBalance() << "\n";  
}  
//Output: Opening account with a balance of: 15.37
```

#### NOTE

針對呼叫類別名稱()的任何位置 `Account`，必須使用關鍵字類別來區別全域範圍的變數帳戶。當類別名稱出現在範圍解析運算子 (::) 的左邊時，不適用這項規則。範圍解析運算子左邊的名稱一律視為類別名稱。

下列範例示範如何使用關鍵字來宣告類型物件的指標 `Account` `class`：

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

在 `Account` 上述語句中，初始化運算式中的(括弧中的)具有全域範圍；其類型為 `double`。

#### NOTE

重複使用識別項名稱 (如這個範例中所示) 會視為不良的程式設計風格。

如需類別物件之宣告和初始化的詳細資訊，請參閱[類別、結構和等位](#)。如需使用 `new` 和 `delete` 自由存放區運算子的詳細資訊，請參閱[new 和 delete 運算子](#)。

## 隱藏具有全域範圍的名稱

您可以藉由在區塊範圍中明確宣告相同的名稱，隱藏具有全域範圍的名稱。不過，您可以使用範圍解析運算子()來存取全域範圍的名稱 `::`。

```
#include <iostream>

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

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

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

## 另請參閱

[基本概念](#)

# 標頭檔 ( C + + )

2020/11/2 • [Edit Online](#)

程式元素的名稱(例如變數、函式、類別等等)必須先經過宣告，才能使用。例如，您不能只在 `x = 42` 未先宣告 'x' 的情況下進行寫入。

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

宣告會告訴編譯器元素是 `int` 、`a`、函式 `double function` `class` 或其他專案。此外，每個名稱都必須在使用它的每個 .cpp 檔案中宣告(直接或間接)。當您編譯器時，每個 .cpp 檔案都會單獨編譯成一個編譯單位。編譯器不知道在其他編譯單位中宣告的名稱。這表示，如果您定義了類別或函式或全域變數，就必須在使用它的每個額外 .cpp 檔案中，提供該內容的宣告。在所有檔案中，該內容的每個宣告必須完全相同。當連結器嘗試將所有編譯單元合併成單一程式時，稍微不一致會導致錯誤或非預期的行為。

為了將錯誤的可能性降到最低，C++ 已採用使用 [標頭檔](#)來包含宣告的慣例。您會在標頭檔中進行宣告，然後在每個 .cpp 檔案或其他需要該宣告的標頭檔中使用 `#include` 指示詞。`#include` 指示詞會在編譯之前，直接在 .cpp 檔案中插入標頭檔的複本。

## NOTE

在 Visual Studio 2019 中，C++ 20 模組功能是針對標頭檔的改進和最終取代而引進的。如需詳細資訊，請參閱[C++ 中的模組總覽](#)。

## 範例

下列範例顯示宣告類別，然後在不同的原始檔中使用它的常見方式。我們會從標頭檔開始 `my_class.h`。它包含類別定義，但請注意，定義不完整；未定義成員函式 `do_something`：

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

接下來，建立一個執行檔(通常使用 .cpp 或類似的副檔名)。我們將會呼叫檔案 `my_class.cpp`，並提供成員宣告的定義。我們會新增 "my\_class.h" 檔案的指示詞，以便在 .cpp 檔案中 `#include` 的這個點插入 `my_class` 嘣告，而且我們 `<iostream>` 要包含以提取的宣告 `std::cout`。請注意，引號會用於與原始程式檔位於相同目錄中的標頭檔，而角括弧則用於標準程式庫標頭。此外，許多標準程式庫標頭並沒有 .h 或任何其他副檔名。

在執行檔案中，我們可以選擇性地使用 `using` 語句，以避免必須以 "N::" 或 "std::" 的每個提及 "my\_class" 或 "cout" 的資格。不要將 `using` 語句放在標頭檔中！

```
// 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;
}
```

現在我們可以 `my_class` 在另一個 .cpp 檔案中使用。我們 `#include` 標頭檔，讓編譯器提取宣告。所有編譯器都必須知道，`my_class` 是具有稱為公用成員函式的類別 `do_something()`。

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

using namespace N;

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

在編譯器完成將每個 .cpp 檔案編譯成 .obj 檔案之後，它會將 .obj 檔案傳遞至連結器。當連結器合併物件檔案時，它只會找到一個 `my_class` 的定義；它位於為 `my_class.cpp` 產生的 .obj 檔案中，而組建成功。

## 包含防護

標頭檔通常會有 *include* 防護或指示詞，`#pragma once` 以確保它們不會多次插入單一 .cpp 檔案中。

```
// 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 */
```

## 要放在標頭檔中的內容

由於標頭檔可能會包含在多個檔案中，因此它不能包含可能會產生多個相同名稱定義的定義。下列是不允許的，或被視為非常不好的作法：

- 命名空間或全域範圍中的內建類型定義
- 非內嵌函式定義
- 非 `const` 變數定義
- 匯總定義

- 未命名的命名空間
- `using` 指示詞

使用指示詞 `using` 不一定會導致錯誤，但可能會造成問題，因為它會將命名空間帶入每個 .cpp 檔案中直接或間接包含該標頭的範圍。

## 範例標頭檔

下列範例顯示標頭檔中允許的各種宣告和定義類型：

```

// 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

#ifndef 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;
}

}

```

# 編譯單位和連結

2020/11/2 • [Edit Online](#)

在 C++ 程式中，符號（例如變數或函式名稱）可以在其範圍內宣告任意次數，但只能定義一次。此規則是「一個定義規則」（ODR）。宣告會在程式中引進（或重新引進）名稱。定義會引進名稱。如果名稱代表變數，定義會明確地將其初始化。函式定義包含簽章加上函數主體。類別定義包含類別名稱，後面接著列出所有類別成員的區塊。（成員函式的主體可以選擇性地分別在另一個檔案中定義）。

下列範例顯示一些宣告：

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

下列範例顯示一些定義：

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

程式是由一個或多個轉譯單位所組成。轉譯單位是由一個執行檔案和它直接或間接包含的所有標頭所組成。執行檔的副檔名通常是 *cpp* 或 *.cxx*。標頭檔的副檔名通常是 *h* 或 *hpp*。每個轉譯單位都是由編譯器獨立編譯。編譯完成後，連結器會將編譯過的轉譯單位合併成單一程式。違反 ODR 規則通常會顯示為連結器錯誤。當相同的名稱在不同的轉譯單位中有兩個不同的定義時，就會發生連結器錯誤。

一般來說，讓變數在多個檔案中可見的最佳方式，是將它放在標頭檔中。然後，在每個需要宣告的 *cpp* 檔案中加入 `#include` 指示詞。藉由新增包含對標頭內容的防護，您可以確保它所宣告的名稱只會定義一次。

在 C++ 20 中，[模組](#) 會引進為標頭檔的改良替代方案。

在某些情況下，可能需要在 *cpp* 檔案中宣告全域變數或類別。在這些情況下，您需要一種方法來告訴編譯器和連結器名稱有何種連結。連結的類型會指定物件的名稱是否只適用於一個檔案或所有檔案。連結的概念僅適用於全域名稱。連結的概念並不適用於在範圍內宣告的名稱。範圍是由一組括住的大括弧所指定，例如函式或類別定義中的。

## 外部與內部連結

*Free 函數* 是在全域或命名空間範圍中定義的函式。非 `const` `global` 變數和 *free* 函數預設具有外部連結；您可以從程式中的任何轉譯單位看到它們。因此，沒有其他全域物件可以擁有該名稱。具有內部連結或沒有連結的符號，只有在宣告它的轉譯單位內才可見。當名稱具有內部連結時，相同的名稱可能會存在於另一個轉譯單位中。在類別定義或函式主體內宣告的變數沒有連結。

您可以將全域名稱明確宣告為，以強制使用內部連結 `static`。這會限制其在宣告時的相同轉譯單位可見度。在此內容中，`static` 表示套用至本機變數時的內容不同。

下列物件預設具有內部連結：

- `const` 物件
- `constexpr` 物件

- `typedefs`
- 命名空間範圍中的靜態物件

若要提供常數物件外部連結，請將它宣告為 `extern`，並將值指派給它：

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

如需詳細資訊，請參閱[extern](#)。

## 另請參閱

[基本概念](#)

# main 函數和命令列引數

2020/12/10 • [Edit Online](#)

所有 C++ 程式都必須有 `main` 函數。如果您嘗試在沒有函式的情況下編譯 C++ 程式 `main`，編譯器會引發錯誤。(動態連結程式庫和程式庫沒有 `static` 函式 `main`。) 函式 `main` 是您的原始程式碼開始執行的位置，但是在程式進入函式之前 `main`，所有 `static` 沒有明確初始化運算式的類別成員都會設定為零。在 Microsoft C++ 中，全域 `static` 物件也會在進入之前初始化 `main`。適用於 `main` 其他任何 C++ 函式的函數會有數項限制。`main` 函數：

- 無法多載 (請參閱 [函數 多載](#))。
- 無法宣告為 `inline`。
- 無法宣告為 `static`。
- 無法取得其位址。
- 無法從程式調用。

## 函數簽章 `main`

`main` 函數沒有宣告，因為它已內建在語言中。如果有的話，的宣告語法如下所 `main` 示：

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

如果未在中指定任何傳回值 `main`，則編譯器會提供傳回值零。

## 標準命令列引數

的引數，可 `main` 方便引數的命令列剖析。`argc` 和 `argv` 的類型是由語言定義。名稱 `argc` 和 `argv` 是傳統的，但您可以將它們命名為任何您喜歡的名稱。

引數定義如下：

`argc`

整數，其中包含後續的引數計數 `argv`。`argc` 參數一律大於或等於1。

`argv`

以 null 終止之字串的陣列，表示由程式的使用者所輸入的命令列引數。依照慣例，`argv[0]` 這是用來叫用程式的命令。`argv[1]` 這是第一個命令列引數。命令列的最後一個引數是 `argv[argc - 1]`，而且 `argv[argc]` 一律為 Null。

如需如何隱藏命令列處理的詳細資訊，請參閱 [自訂 C++ 命令列處理](#)。

### NOTE

依照慣例，`argv[0]` 是程式的檔案名。不過，在 Windows 上，可以使用來產生進程 `CreateProcess`。如果您同時使用第一個和第二個引數 (`LpApplicationName` 和 `LpCommandLine`)，則 `argv[0]` 可能不是可執行檔名稱。您可以使用 `GetModuleFileName` 來取得可執行檔名稱及其完整路徑。

## Microsoft 特定的擴充功能

下列各節說明 Microsoft 特定的行為。

## wmain 函數和 \_tmain 宏

如果您將原始程式碼設計成使用 Unicode 寬 character，您可以使用 Microsoft 特定的 `wmain` 進入點，也就是的全 character 版本 `main`。以下是的有效宣告語法 `wmain`：

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

您也可以使用 Microsoft 特定的 `_tmain`，也就是中定義的預處理器宏 `tchar.h`。`_tmain``main` 除非 `_UNICODE` 已定義，否則解析為。在此情況下，`_tmain` 會解析成 `wmain`。開頭為的 `_tmain` 宏和其他宏適用 `_t` 于必須針對窄和寬 character 集建立不同版本的程式碼 char。如需詳細資訊，請參閱 [使用泛型文字](#) 對應。

## 返回 void 來源 main

做為 Microsoft 擴充功能，`main` 和函式可以宣告 `wmain` 為傳回 `void` (沒有) 的傳回值。此延伸模組也適用於一些其他編譯器，但不建議使用。當沒有傳回值時，就可以進行對稱 `main`。

如果您宣告 `main` 或傳回 `wmain` `void`，就無法使用語句，將程式 exit 代碼傳回父進程或作業系統 `return`。若要 exit 在宣告或宣告為時傳回程序代碼 `main` `wmain` `void`，您必須使用 `exit` 函數。

## envp 命令列引數

或簽章 `main` `wmain` 允許選用的 Microsoft 特定延伸模組來存取環境變數。此延伸模組在 Windows 和 UNIX 系統的其他編譯器中也很常見。名稱 `envp` 是傳統的，但您可以將環境參數命名為任何您喜歡的名稱。以下是包含環境參數之引數清單的有效宣告：

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

### envp

選擇性 `envp` 參數是字串陣列，代表使用者環境中設定的變數。這個陣列由 NULL 項目終止。您可以將它宣告為指標的陣列，以 `char` (`char *envp[]`) 或指標指向 `char` () 的指標 `char **envp`。如果您的程式使用 `wmain` 而非 `main`，請使用 `wchar_t` 資料類型，而不是 `char`。

傳遞至的環境區塊 `main` `wmain` 是目前環境的「凍結」複本。如果您之後藉由呼叫或來變更環境 `putenv` `_wputenv`，則目前的環境 (由或所傳回 `getenv`，`_wgetenv` 而 `_environ` 或 `_wenviron` 變數) 將會變更，但指向的區塊不會 `envp` 變更。如需如何隱藏環境處理的詳細資訊，請參閱 [自訂 c++ 命令列處理](#)。`envp` 引數與 C89 standard 相容，但與 c++ 標準不相容。

### 的範例引數 main

下列範例顯示如何使用 `argc`、`argv` 和 `envp` 引數來 `main`：

```

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

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

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

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

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

```

## 剖析 C++ 命令列引數

Microsoft C/c + + 程式碼所使用的命令列剖析規則是 Microsoft 特有的。在解讀作業系統命令列上指定的引數時，執行時間啟動程式碼會使用這些規則：

- 引數會以空白或定位鍵的泛空白字元 (White Space) 進行分隔。
- 第一個引數 (`argv[0]`) 會以特殊方式處理。它代表程式名稱。因為它必須是有效的路徑名稱，所以允許以雙引號括住的部分 (" )。輸出中不包含雙引號標記 `argv[0]`。以雙引號括住的部分會防止將空格或定位字元 `char acter` 為引數的結尾。這份清單中的後續規則並不適用。
- 以雙引號括住的字串會轉譯為單一引數，其可能包含空白字元 `char acters`。有引號的字串可以內嵌到引數中。未將插入號 (^) 被辨識為 escape `char acter` 或分隔符號。在加上引號的字串內，會將一組雙引號轉譯為單一的已標記雙引號。如果命令列在找到右雙引號標記之前結束，則目前為止讀取的所有 `char acters` 都會輸出為最後一個引數。
- 雙引號前面加上反斜線 (\") 會被視為常值雙引號標記 ( " )。
- 反斜線會以字面方式解讀，除非它們緊接在雙引號前面。
- 如果有偶數的反斜線，後面加上雙引號，則 \ 會在每一對反斜線 () 的陣列中放入一個反斜線 ()，`argv` \\ 而雙引號 ( " ) 會被視為字串分隔符號。
- 如果反斜線後面加上奇數數目的反斜線，則會將一個反斜線 (\) 放置於陣列中，`argv` 以 () 的每一對反斜線 \\ 。雙引號標記會被重設反斜線來被視為 escape 序列 main，這會造成常值雙引號標記 ( " ) 放置在中 `argv` 。

### 命令列引數剖析範例

下列程式示範如何傳遞命令列引數：

```

// 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";
}

```

## 剖析命令列的結果

下表顯示範例輸入和預期的輸出，示範上述清單中的規則。

INPUT	ARGV[1]	ARGV[2]	ARGV3
"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		

## 萬用字元展開

Microsoft 編譯器可選擇性地使用 萬用字元 characters、問號 (?) 和星號 (\*)，在命令列上指定檔案名和路徑引數。

命令列引數是由執行時間啟動程式碼中的內部常式處理，預設不會將萬用字元擴充至字串陣列中的個別字串 `argv`。您可以 `setargv.obj` `wsetargv.obj` `wmain` 在 `/link` 編譯器選項或命令列中包含) 的檔案 (檔案，以啟用萬用字元擴充 `LINK`。

如需執行時間啟動連結器選項的詳細資訊，請參閱 [連結選項](#)。

## 自訂 C + + 命令列處理

如果您的程式未採用命令列引數，您可以隱藏命令列處理常式來節省少量的空間。若要隱藏其使用方式，請 `noarg.obj` `main` `wmain` 在您的 `/link` 編譯器選項或命令列中包含和) 的檔案 (`LINK`)。

同樣地，如果您從未透過引數存取環境資料表 `envp`，可以隱藏內部環境處理常式。若要隱藏其使用方式，請 `noenv.obj` `main` `wmain` 在您的 `/link` 編譯器選項或命令列中包含和) 的檔案 (`LINK`)。

您的程式可能會呼叫 `spawn` `exec` C 執行時間程式庫中的或系列常式。如果有的話，您就不應該隱藏環境處理常式，因為它是用來將環境從父進程傳遞至子進程。

另請參閱

[基本概念](#)

# C++ 程式終止

2020/12/10 • [Edit Online](#)

在 C++ 中，您可以用下列方式結束程式：

- 呼叫 `exit` 函數。
- 呼叫 `abort` 函數。
- 從執行 `return` 語句 `main`。

## exit 函式

在中宣告的函式會 `exit` <stdlib.h> 終止 C++ 程式。提供作為引數的值 `exit` 會以程式的傳回碼或結束代碼傳回作業系統。依照慣例，傳回碼為零，表示程式順利完成。您可以使用常數，`EXIT_FAILURE` `EXIT_SUCCESS` 也可以在中定義，<stdlib.h> 以表示程式的成功或失敗。

從函式發出 `return` 語句 `main` 相當於 `exit` 使用傳回值做為其引數來呼叫函式。

## abort 函式

函式 `abort` 也會在標準 include 檔中宣告 <stdlib.h>，以終止 C++ 程式。和之間的差異在於，可 `exit` `abort` `exit` 讓 C++ 執行時間終止處理發生（全域物件的析構函數），但會 `abort` 立即終止程式。函式會 `abort` 略過初始化的全域靜態物件的一般終結進程。它也會略過使用函數指定的任何特殊處理 `atexit`。

## atexit 函式

使用函式 `atexit` 來指定程式終止之前執行的動作。在執行結束處理函式之前，不會先將呼叫之前初始化的全域靜態物件終結 `atexit`。

## return 中的語句 main

發出的 `return` 語句在 `main` 功能上相當於呼叫函式 `exit`。請考慮下列範例：

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

`exit` `return` 上述範例中的和語句的功能相同。一般來說，C++ 要求具有傳回類型的函式不會傳回 `void` 值。函 `main` 式是例外狀況；它的結尾不能是 `return` 語句。在這種情況下，它會傳回叫用進程的實值指定值。`return` 語句可讓您指定的傳回值 `main`。

## 靜態物件的銷毀

當您 `exit` 從呼叫或執行 `return` 語句時 `main`，靜態物件會在呼叫之後，以其初始化（的反向順序終結（如果有話）`atexit`）。下列範例將示範這類初始化和清除如何運作：

### 範例

在下列範例中，`sd1` `sd2` 會在進入之前建立和初始化靜態物件和 `main`。使用語句終止這個程式之後 `return`，第一個 `sd2` 就會終結，然後是 `sd1`。`ShowData` 類別的解構函式會關閉與這些靜態物件相關聯的檔案

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

另一種撰寫這個程式碼的方式，是使用區塊範圍宣告 `ShowData` 物件，讓物件能夠在超出範圍時終結：

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

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

## 另請參閱

[main 函數和命令列引數](#)

# Lvalues 和 Rvalues (C++)

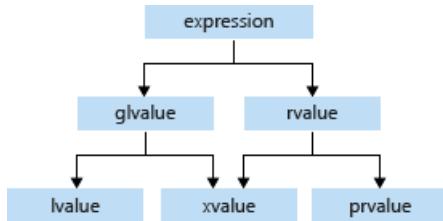
2020/11/2 • [Edit Online](#)

每個 C++ 運算式都有類型，而且屬於值分類。值類別目錄是編譯器在運算式評估期間建立、複製和移動暫存物件時必須遵循之規則的基礎。

C++ 17 標準定義了運算式值分類，如下所示：

- *Gvalue* 是一種運算式，其評估會決定物件、位欄位或函數的識別。
- *Xprvalue* 是一種運算式，其評估會初始化物件或位欄位，或計算運算子的運算元值，如其出現所在內容所指定。
- *Xvalue* 是一種 *gvalue*，代表可以重複使用其資源的物件或位欄位（通常是因為它在其存留期的結尾附近）。範例：牽涉到右值參考(8.3.2)的特定類型運算式會產生 *xvalues*，例如呼叫的函式，其傳回類型是 *rvalue* 參考，或轉換成右值參考型別。
- 左值是不是 *xvalue* 的 *gvalue*。
- 右值是 *xprvalue* 或 *xvalue*。

下圖說明類別之間的關聯性：



左值具有您的程式可以存取的位址。左值運算式的範例包括變數名稱，包括 `const` 變數、陣列元素、傳回左值參考的函式呼叫、位欄位、等位和類別成員。

Xprvalue 運算式沒有可供您的程式存取的位址。Xprvalue 運算式的範例包括常值、傳回非參考型別的函式呼叫，以及在運算式 evalution 期間建立的暫存物件，但只能由編譯器存取。

Xvalue 運算式具有您的程式無法再存取的位址，但可用來初始化右值參考，以提供運算式的存取權。範例包括傳回右值參考的函式呼叫，以及陣列或物件為 *rvalue* 參考之成員運算式的陣列注標、成員和指標。

## 範例

下列範例將示範數種正確和不正確的左值和右值用法：

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

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

    // Incorrect usage: The left operand must be an lvalue (C2106).`j * 4` is a prvalue.
    7 = i; // C2106
    j * 4 = 7; // C2106

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

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

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

#### NOTE

本主題中的範例將說明運算子未多載時的正確和不正確用法。藉由多載運算子，您就可以讓像是 `j * 4` 這樣的運算式變成左值。

當您參考物件參考時，通常會使用「左值」和「右值」等詞彙。如需參考的詳細資訊，請參閱左值參考宣告子：[&](#)和右值參考宣告子：[&&](#)。

## 另請參閱

[基本概念](#)

[左值參考宣告子 :&](#)

[右值參考宣告子 :&&](#)

# 暫存物件

2020/11/2 • [Edit Online](#)

在某些情況下，編譯器必須建立暫存物件。可能需要建立這些暫存物件的原因如下：

- 若要 `const` 使用與所初始化之參考的基礎類型不同之類型的初始化運算式來初始化參考。
- 為了儲存傳回使用者定義類型之函式的傳回值。只有在您的程式不會將傳回值複製到物件時，才會建立這些暫存物件。例如：

```
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.
```

由於傳回值不會複製到另一個物件，因此會建立暫存物件。建立暫存物件的更常見案例是在評估運算式期間，必須呼叫多載運算子函式時。這些多載運算子函式傳回的使用者定義類型通常不會複製到另一個物件。

以 `ComplexResult = Complex1 + Complex2 + Complex3` 運算式為例。`Complex1 + Complex2` 運算式會加以評估，而且結果會儲存在暫存物件中。接下來，會評估運算式暫存 `+ Complex3`，並將結果複製到 `ComplexResult`（假設指派運算子未多載）。

- 為了將轉型的結果儲存為使用者定義類型。當特定類型的物件明確轉換成使用者定義類型時，該新物件會建構為暫存物件。

暫存物件具有存留期，該存留期是從物件建立的時間點開始到終結的時間點為止。任何建立超過一個暫存物件的運算式，最終都會依建立物件的反向順序終結這些物件。下表顯示發生終結的點。

## 暫存物件的終結點

運算式評估結果	運算式評估結果所建立的所有而暫存物件都會在運算式語句的結尾(也就是分號)，或在 <code>for</code> 、 <code>if</code> 、 <code>while</code> 、 <code>do</code> 和 <code>switch</code> 語句的控制運算式結尾處終結。
初始化 <code>const</code> 參考	如果初始設定式不是與所要初始化的參考相同類型的左值，則會建立基礎物件類型的暫存，並且使用初始化運算式進行初始化。這個暫存物件會在其所繫結的參考物件終結時立即終結。

# 對齊

2020/11/2 • [Edit Online](#)

C++ 的其中一項低階功能可指定記憶體中物件的精確對齊方式，以充份利用特定的硬體架構。根據預設，編譯器會將類別和結構成員對齊其大小值：`bool` 在 `char` 1 個位元組的界限上、`int` 在 2 個位元組的界限上、`short` 和 `long` 在 4 個位元組的界限上，以及 `float`、`long long`、`double`、和 `long double` 在 8 位元組界限。

在大部分的情況下，您永遠不需要擔心對齊，因為預設的對齊方式已經是最佳的。不過，在某些情況下，您可以藉由指定資料結構的自訂對齊，來達到顯著的效能改進或節省記憶體。在 Visual Studio 2015 之前，您可以使用 Microsoft 特有的關鍵字 `_alignof`，並 `_declspec(align)` 指定大於預設值的對齊方式。從 Visual Studio 2015 開始，您應該使用 C++ 11 標準關鍵字 `alignof`，以及 `alignas` 最大的程式碼可攜性。新關鍵字的行為與 Microsoft 專屬的擴充功能相同。這些延伸模組的檔也適用於新的關鍵字。如需詳細資訊，請參閱 [alignof 運算子和 align](#)。C++ 標準不會指定在界限小於目標平臺的編譯器預設值時，進行對齊的封裝行為，因此您仍然需要 `#pragma pack` 在該情況下使用 Microsoft。

針對具有自訂對齊的資料結構，使用 `aligned_storage` 類別來進行記憶體配置。`Aligned_union` 類別是用來指定具有非一般程式或析構函式之等位的對齊方式。

## 對齊和記憶體位址

對齊是記憶體位址的屬性，以數位位址取 2 乘幕的模數來表示。例如，位址 0x0001103F 模數 4 是 3。該位址會被視為對齊  $4n + 3$ ，其中 4 表示所選的 2 乘幕。位址的對齊方式取決於所選的 2 乘幕。相同位址取 8 的模數為 7。如果位址的對齊方式為  $Xn+0$ ，則稱為對齊 X。

CPU 會執行指示，以處理儲存在記憶體中的資料。資料會以其在記憶體中的位址來識別。單一 datum 也有大小。如果它的位址對齊其大小，我們就會呼叫以自然對齊的基準。否則會將其稱為未對齊。例如，如果用來識別它的位址具有 8 位元組對齊，則會自然對齊 8 位元組的浮點基準。

## 資料對齊的編譯器處理

編譯器嘗試以防止資料對齊的方式進行資料分配。

針對簡單的資料類型，編譯器指派的位址會是以位元組為單位之資料類型大小的倍數。例如，編譯器會將位址指派給類型的變數 `long`，這是 4 的倍數，將位址的底部 2 位設定為零。

編譯器也會以自然對齊結構的每個元素的方式來填補結構。請考慮 `struct x_` 下列程式碼範例中的結構：

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

編譯器會填補這個結構，以便強制執行自然對齊。

下列程式碼範例顯示編譯器如何將填補的結構放在記憶體中：

```
// 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];
```

這兩個宣告都會傳回 `sizeof(struct x_)` 12 個位元組。

第二個宣告包含兩個填補項目：

1. `char _pad0[3]` 將成員對齊 `int b` 4 位元組界限。
2. `char _pad1[1]` 將結構的陣列元素對齊 `struct _x bar[3];` 四個位元組的界限。

填補 `bar[3]` 會以允許自然存取的方式來對齊的元素。

下列程式碼範例顯示 `bar[3]` 陣列版面配置：

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` 類型規範是可移植的 C++ 標準方法，可指定自訂變數和使用者定義類型的對齊方式。`alignof` 運算子同樣是可移植的標準方式，以取得指定類型或變數的對齊方式。

## 範例

您可以 `alignas` 在類別、結構或等位上，或個別成員上使用。當遇到多個指定名稱時 `alignas`，編譯器會選擇最嚴格的一個（具有最大值的規範）。

```
// 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
}
```

## 另請參閱

[資料結構對齊](#)

# Trivial、標準配置、POD 與常值類型

2020/4/22 • [Edit Online](#)

「配置」\*\* 這個字詞，指的是類別、結構或等位型別的物件成員，在記憶體中如何安排。在某些情況下，語言規格會完善地定義配置。但是當類別或結構包含某些 C++ 語言特徵標記時（例如虛擬基底類別、虛擬函式、存取控制不同的成員），編譯器就可以自由選擇配置。配置可能取決於執行的最佳化而有不同，在許多情況下，物件甚至可能不會佔用記憶體的連續區域。例如，如果某個類別有虛擬函式，該類別的所有執行個體就可能共用單一虛擬函式表。這樣的類型非常實用，但也有其限制。由於配置未經定義，因此無法傳遞到以其他語言（例如 C）寫成的程式，而且也因為配置可能是不連續的，而無法透過快速的低階函式（例如 `memcpy`）確實複製，或透過網路序列化。

為了讓編譯器及 C++ 程式和中繼程式能夠依據特定記憶體配置，對為了作業而提供的任何類型評估合適性，C++14 帶來了三種類別的簡單分類與結構：*trivial*、標準配置\*\* 和 *POD*（即 Plain Old Data）。標準程式庫有函式範本 `is_trivial<T>`、`is_standard_layout<T>` 及 `is_pod<T>`，能判斷提供的類型是否屬於提供的分類。

## Trivial 類型

當 C++ 中的類別或結構有編譯器提供或明確預設的特殊成員函式時，就是 Trivial 類型。其佔有連續的記憶體區域。其成員有不同的存取指定名稱。在 C++ 中，編譯器可以自由選擇在此情況下如何為成員排序。因此，您可對這類物件進行 `memcpy`，但無法確實從 C 程式加以取用。Trivial 類型的 T 可以複製到 Char 或未簽署 Char 的陣列中，並可安全複製回 T 變數中。請注意，基於對齊需求，類型成員之間可能會有填補的位元組。

Trivial 類型有 Trivial 預設建構函式、Trivial 複製建構函式、Trivial 複製指派運算子及 Trivial 解構函式。在各個情況下，*Trivial* 分別表示建構函式/運算子/解構函式並非使用者提供的，而且所屬類別

- 沒有虛擬函式或虛擬基底類別。
- 沒有對應非 Trivial 建構函式/運算子/解構函式的基底類別
- 沒有對應非 Trivial 建構函式/運算子/解構函式的類別類型資料成員

下列範例顯示了 Trivial 類型。在 Trivial2 中，您必須提供預設建構函式，`Trivial2(int a, int b)` 建構函式才會存在。對於要符合 Trivial 資格的類型，您必須明確預設建構函式。

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

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

## Standard layout 類型

當某個類別或結構不包含某些 C++ 語言特徵標記（例如在 C 語言中找不到的虛擬函式），而且所有成員都有相同的存取控制時，即為標準配置類型。這個類型可進行 `memcpy`，而且配置會經過充分定義，能夠供 C 程式取用。標準配置類型可以有使用者定義的特殊成員函式。此外，標準配置類型有以下特性：

- 沒有虛擬函式或虛擬基底類別
- 所有非靜態資料成員都有相同的存取控制
- 類別類型的所有非靜態資料成員均具有標準配置
- 任何基底類別均具有標準配置
- 第一個非靜態資料成員不會是相同類型的基底類別。
- 符合以下其中一個條件：
  - 衍生程度最高的類別中沒有非靜態資料成員，最多只有一個具非靜態資料成員的基底類別，或
  - 沒有具非靜態資料成員的基底類別

下列程式碼示範了其中一個標準配置類型的範例：

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

最後兩個需求以程式碼來說明可能較為合適。在下一個範例中，即使 `Base` 具有標準配置，`Derived` 也不是標準配置，因為其 (衍生程度最高的類別) 與 `Base` 都有非靜態的資料成員：

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

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

在此範例中，因為 `Base` 沒有非靜態的資料成員，所以 `Derived` 具有標準配置：

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

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

如果 `Base` 有資料成員，而 `Derived` 只有成員函式，則 `Derived` 也會具有標準配置。

## POD 類型

當類別或結構同時為 Trivial 或標準配置時，即為 POD (Plain Old Data) 類型。因此，POD 類型的記憶體配置會是連

續的，而各個成員的位址高於在其之前宣告的成員，這樣一來，位元組複製的位元組和二進位 I/O 才可在這些類型執行。純量類型 (例如 int) 也屬於 POD 類型。若 POD 類型是類別，非靜態資料成員就只能是 POD 類型。

## 範例

下列範例示範了 Trivial、標準配置和 POD 類型的差異：

```
#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;
}
```

## 常值類型

常值類型的配置可以在編譯時期決定。以下是常值類型：

- void
- 純量類型
- 參考
- Void、純量類型或參考的陣列
- 具有 trivial 解構函式和一或多個非移動或複製建構函式之 constexpr 建構函式的類別。此外，其所有的非靜態資料成員和基底類別必須是常值類型，且不會變動。

## 另請參閱

[基本概念](#)

# 用為實值型別的 C++ 類別

2019/12/2 • [Edit Online](#)

C++ 類別預設為實數值型別。您可以將它們指定為參考型別，讓多型行為能夠支援物件導向程式設計。數值型別有時可以從記憶體和版面配置控制項的角度來查看，而參考型別則是關於基類和虛擬函式，以進行多型的用途。根據預設，實值型別為可複製，這表示一定會有複製的函式和複製指派運算子。針對參考型別，您可以將類別設為非可複製（停用複製程式設計函數和複製指派運算子），並使用支援其所需多型的虛擬析構函式。實值型別也是內容的相關資訊，當複製它們時，一律會提供兩個獨立的值供您分別修改。參考型別是關於身分識別—這是什麼類型的物件？因此，「參考型別」也稱為「多型類型」。

如果您真的想要類似參考的型別（基類、虛擬函式），您必須明確停用複製，如下列程式碼中的 `MyRefType` 類別所示。

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

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

public:
    MyRefType () {}
};

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

編譯上述程式碼將會產生下列錯誤：

```
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'
```

## 實數值型別和移動效率

由於新的複製優化，因此可避免複製配置額外負荷。例如，當您在字串向量的中間插入字串時，將不會有任何複製重新配置額外負荷，只有移動—即使它導致向量本身增加也一樣。這也適用於其他作業，例如在兩個非常大型的物件上執行 add 運算。如何啟用這些值作業優化？在某些C++編譯器中，編譯器會隱含地為您啟用此功能，就像複製函式可以由編譯器自動產生一樣。不過，在C++中，您的類別必須在類別定義中宣告，以「加入宣告」來移動指派和函數。在適當的成員函式宣告中使用 double 符號(&&)右值參考，並定義移動函數和移動指派方法，即可完成這項作業。您也必須插入正確的程式碼，以「竊取 getmembers」來源物件。

您要如何決定是否需要啟用移動？如果您已經知道您需要啟用「複製結構」，您可能會想要在可能比深層複本更便宜的情況下，啟用移動。不過，如果您知道需要移動支援，則不一定表示您想要啟用複製。第二種情況則稱為「僅限移動類型」。標準程式庫中已有一個範例 `unique_ptr`。請注意，舊的 `auto_ptr` 已被取代，而且已 `unique_ptr` 精確地取代，因為舊版中缺少 move 語義支援C++。

藉由使用移動語義，您可以傳回傳值或插入中間。Move 是複製的優化。需要堆積配置做為因應措施。請考慮下列虛擬程式碼：

```

#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

```

## 啟用適當數值型別的移動

針對類似值的類別，移動可以比深層複本便宜，針對效率啟用移動結構和移動指派。請考慮下列虛擬程式碼：

```

#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!");
    }
};

```

如果您啟用 [複製結構/指派]，也可以啟用 [移動結構/指派]（如果它比深層複本便宜）。

某些非實值類型為僅限移動，例如當您無法複製資源時，只會傳送擁有權。範例：`unique_ptr`。

## 另請參閱

[C++ 類型系統](#)

[歡迎回到C++](#)

[C++ 語言參考](#)

[C++ 標準程式庫](#)

# 類型轉換與類型安全

2020/11/2 • [Edit Online](#)

本檔會識別常見的類型轉換問題，並說明如何在您的 C++ 程式碼中加以避免。

當您撰寫 C++ 程式時，請務必確定它是型別安全的。這表示每個變數、函式引數和函式傳回值都是儲存可接受的資料類型，而涉及不同類型值的作業則不會造成資料遺失、不正確的位模式解讀或記憶體損毀。永遠不會明確或隱含地將值從一個類型轉換成另一個類型的程式，是由定義的型別安全。不過，有時候也需要類型轉換，甚至是不安全的轉換。例如，您可能必須將浮點運算的結果儲存在類型的變數中 `int`，或者您可能必須將中的值傳遞 `unsigned int` 至接受的函式 `signed int`。這兩個範例都會說明不安全的轉換，因為它們可能會導致資料遺失或重新轉譯值。

當編譯器偵測到不安全的轉換時，會發出錯誤或警告。錯誤停止編譯，警告可讓編譯繼續進行，但在程式碼中表示可能發生的錯誤。不過，即使您的程式編譯時不會出現警告，仍然可能包含會導致隱含類型轉換產生不正確結果的程式碼。類型錯誤也可以透過程式碼中的明確轉換或轉換來引入。

## 隱含類型轉換

當運算式包含不同內建類型的運算元，而且沒有明確轉換時，編譯器會使用內建的標準轉換來轉換其中一個運算元，讓類型相符。編譯器會以定義完善的順序來嘗試轉換，直到其中一個成功為止。如果選取的轉換是升級，則編譯器不會發出警告。如果轉換為縮小，則編譯器會發出有關可能遺失資料的警告。實際資料遺失的發生與否取決於相關的實際值，但我們建議您將此警告視為錯誤。如果涉及使用者定義型別，則編譯器會嘗試使用您在類別定義中所指定的轉換。如果找不到可接受的轉換，則編譯器會發出錯誤，而且不會編譯器。如需有關管理標準轉換之規則的詳細資訊，請參閱 [標準轉換](#)。如需使用者定義轉換的詳細資訊，請參閱 [使用者定義的轉換 \(C++/CLI\)](#)。

### 擴輒轉換(升級)

在擴輒轉換中，較小的變數中的值會指派給較大的變數，而不會遺失資料。因為擴輒轉換一律是安全的，所以編譯器會以無訊息模式執行，而且不會發出警告。下列轉換是擴輒轉換。

III	III
<code>signed</code> 或以外的任何或 <code>unsigned</code> 整數類型 <code>long long</code> **** <code>_int64</code>	<code>double</code>
<code>bool</code> 或 ** <code>char</code>	任何其他內建類型
<code>short</code> 或 ** <code>wchar_t</code>	<code>int</code> , <code>long</code> , <code>long long</code>
<code>int</code> , <code>long</code>	<code>long long</code>
<code>float</code>	<code>double</code>

### 縮小轉換(強制型轉)

編譯器會以隱含方式執行縮小轉換，但會警告您可能遺失資料。請務必認真地採取這些警告。如果您確定不會發生任何資料遺失，因為較大的變數中的值一定會符合較小的變數，然後加入明確轉換，讓編譯器不會再發出警告。如果您不確定轉換是安全的，請將某種執行時間檢查新增至程式碼，以處理可能的資料遺失，使其不會造成您的程式產生不正確的結果。

從浮點類型到整數類型的任何轉換都是縮小轉換，因為浮點值的小數部分會被捨棄並遺失。

下列程式碼範例顯示一些隱含的縮小轉換，以及編譯器針對它們發出的警告。

```
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 = 0xffff; //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
```

### 簽署不帶正負號的轉換

帶正負號的整數類資料類型和其不帶正負號的對應一律是相同的大小，但不同之處在於如何解讀值轉換的位模式。下列程式碼範例示範當相同的位模式被視為帶正負號的值和不帶正負號的值時，會發生什麼事。儲存在和中的位 `num` 模式 `num2` 永遠不會變更，如先前的圖例所示。

```
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
```

請注意，這兩個方向的值都是向量的。如果您的程式產生奇怪的結果，其中值的正負號看起來與預期的相反，請尋找帶正負號和不帶正負號的整數類資料類型之間的隱含轉換。在下列範例中，運算式(0-1)的結果會在 `int` 儲存在中時，從隱含地轉換為 `num`。這會導致向量位模式。

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

編譯器不會警告帶正負號和不帶正負號整數類資料類型之間的隱含轉換。因此，我們建議您避免已登入不帶正負號的轉換。如果您無法避免這些問題，請將執行時間檢查新增至您的程式碼，以偵測要轉換的值是否大於或等於零，而且小於或等於帶正負號類型的最大值。此範圍中的值會從帶正負號的轉換成不帶正負號的，或從不帶正負號的未向量。

### 指標轉換

在許多運算式中，C 樣式陣列會隱含地轉換成陣列中第一個專案的指標，而常數轉換則會以無訊息方式發生。雖然這很方便，但也可能容易出錯。例如，下列設計錯誤的程式碼範例看似無意義，但它會編譯並產生 'p' 的結果。首先，"Help" 字串常值會轉換成，指向 `char*` 陣列的第一個元素；該指標接著會由三個元素遞增，使其現在指向最後一個元素 'p'。

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

### 明確轉換(轉換)

藉由使用轉換作業，您可以指示編譯器將一種類型的值轉換成另一種類型。在某些情況下，如果這兩個類型完全無關，編譯器會引發錯誤，但是在其他情況下，即使作業不是型別安全，也不會引發錯誤。請謹慎使用轉換，因為從某種類型轉換成另一種是程式錯誤的潛在來源。不過，有時候會需要轉換，而不是所有的轉換都有同樣的危

險。轉換的有效用法是當您的程式碼執行縮小轉換，而且您知道轉換不會造成您的程式產生不正確的結果。實際上，這會告訴編譯器您知道您正在執行的作業，以及停止中斷您的相關警告。另一個用法是將指標衍生類別轉換為指標對基底類別。另一個用途是將變數的特性轉 `const` 型為不需要非引數的函式 `const`。大部分的轉換作業都牽涉到一些風險。

在 C 樣式程式設計中，會針對所有類型的轉換使用相同的 C 樣式轉型運算子。

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

C 樣式轉換運算子等同于呼叫運算子()，因此會在程式碼中 inconspicuous 並容易忽略。這兩者都是不好的，因為他們難以辨識或搜尋，而且兩者的不同之處在于叫用、和的任何組合 `static` `const` `reinterpret_cast`。找出原本的格式轉換實際上可能很棘手，而且容易出錯。基於上述所有原因，當需要轉換時，建議您使用下列其中一種 c++ 轉型運算子，在某些情況下，它會有更多型別安全，並以更明確的方式表達程式設計意圖：

- `static_cast`，適用於在編譯時期檢查的轉換。`static_cast` 如果編譯器偵測到您嘗試在完全不相容的類型之間進行轉換，則會傳回錯誤。您也可以使用它，在指標對基底和指標衍生之間進行轉換，但編譯器不一定會在執行時間判斷這類轉換是否安全。

```
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);
```

如需詳細資訊，請參閱 [static\\_cast](#)。

- `dynamic_cast`，針對安全的執行時間檢查，將指標轉型轉換成衍生的指標。A `dynamic_cast` 比向下轉換更安全 `static_cast`，但執行時間檢查會產生一些額外負荷。

```
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;
```

如需詳細資訊，請參閱 [dynamic\\_cast](#)。

- `const_cast`，用於轉型為 `const` 變數的性質，或 `const` 將非變數轉換為 `const`。藉 `const` 由使用這個運算子，就像使用 C 樣式的轉型一樣容易發生錯誤，不同之處在於 `const_cast` 您不可能不小心執行轉換。有時候您必須轉換掉變數的 `const` 性質，例如，將 `const` 變數傳遞給採用非參數的函式 `const`。下列範

例示範如何執行。

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

如需詳細資訊，請參閱[const\\_cast](#)。

- `reinterpret_cast`，用於不相關的類型(例如指標類型和)之間的轉換 `int`。

#### NOTE

這個轉型運算子不常使用，而且不保證可移植到其他編譯器。

下列範例說明如何 `reinterpret_cast` 與不同 `static_cast`。

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

如需詳細資訊，請參閱[reinterpret\\_cast 運算子](#)。

## 另請參閱

[C++ 類型系統](#)

[歡迎回到 C++](#)

[C++ 語言參考](#)

[C++ 標準程式庫](#)

# 標準轉換

2020/11/2 • [Edit Online](#)

C++ 語言定義其基本類型之間的轉換。同時定義指標、參考及成員指標衍生類型的轉換。這些轉換稱為**標準轉換**。

本節將討論下列標準轉換：

- 整數提升
- 整數轉換
- 浮動轉換
- 浮動和整數轉換
- 算術轉換
- 指標轉換
- 參考轉換
- 成員指標轉換

## NOTE

使用者定義的類型可以自行指定其轉換。使用者定義類型的轉換會涵蓋在函式和轉換中。

下列程式碼會引發轉換 (本範例是整數提升)：

```
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;
```

必須產生參考類型，轉換結果才會是左值。例如，宣告為的使用者定義轉換會傳回 `operator int&()` 參考，而且是左值。不過，宣告為的轉換會傳回 `operator int()` 物件，而不是左值。

## 整數提升

整數類資料類型的物件可以轉換成另一個較大的整數類型，也就是可以代表一組大型值的類型。這種擴展類型的轉換稱為「**整數提升**」。使用整數提升時，您可以在運算式中使用下列類型，其中可以使用另一個整數類型：

- 和類型的物件、常值和常數 `char` `****` `short` `int`
- 列舉類型
- `int` 位欄位
- 列舉值

C++ 升級為「保留值」，因為升級後的值保證會與升級前的值相同。在值保留的升級中，較短的整數類資料類型(例如，位欄位或類型的物件)的物件 `char` 會升級為類型(`int` 如果 `int` 可以代表原始類型的完整範圍)。如果 `int` 無法代表值的完整範圍，則物件會升級為類型 `unsigned int`。雖然這項策略與標準 C 所使用的策略相同，但保留值的轉換不會保留物件的 "正負號狀態"。

保留值的提升和保留正負號狀態的提升通常會產生相同的結果。不過，如果提升的物件顯示為，它們可能會產生不同的結果：

- `/`、`%`、`/=`、`%=`、`<`、`<=`、`>` 或的運算元 `>=`

這些運算子需要依據正負號判斷結果。在套用到這些運算元時，保留值和簽署的升級會產生不同的結果。

- 或的左運算元 `>>`>>=`

這些運算子會在 shift 作業中以不同的方式來處理帶正負號和未簽署的數量 對於帶正負號的數量，右移作業會將符號位傳播到空出的位位置，而空出的位位置則會以不帶正負號的數量填入零。

- 多載函式的引數，或多載運算子的運算元，這取決於引數比對的運算元類型正負號狀態。如需定義多載運算子的詳細資訊，請參閱多載運算子。

## 整數轉換

整數轉換是整數類資料類型之間的轉換。整數類資料類型為 `char`、`short` (或 `short int`)、`int` `long` 和 `long long`。這些類型可能會以 `signed` 或限定 `unsigned`，而且 `unsigned` 可以當做的速記使用 `unsigned int`。

### 帶正負號到不帶正負號

帶正負號整數類型的物件可以轉換成對應的不帶正負號的類型。當這些轉換發生時，實際的位模式不會變更。不過，資料的轉譯也會變更。請參考下列程式碼：

```
#include <iostream>

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

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

在上述範例中，`signed short` `i` 會定義、，並將其初始化為負數。運算式 `(u = i)` 會在 `i` 指派給之前，將轉換成 `unsigned short` `u`。

### 不帶正負號到帶正負號

不帶正負號整數類資料類型的物件可以轉換成對應的帶正負號資料類型。不過，如果不帶正負號的值不在帶正負號類型的可顯示範圍內，則結果不會有正確的值，如下列範例所示：

```

#include <iostream>

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

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

```

在上述範例中，`u` 是 `unsigned short` 必須轉換成帶正負號數量的整數物件，以評估運算式 `(i = u)`。因為無法在中正確表示其值 `signed short`，所以會誤解資料，如下所示。

## 浮點轉換

浮動類型的物件可以安全地轉換為更精確的浮動類型，也就是說，轉換時不會遺失精確度。例如，從 `float` 到 `double` 或到的轉換 `double` `long double` 是安全的，且值不變。

浮動類型的物件也可以轉換成較不精確的類型（如果它是在該類型所表示的範圍內）。（如需浮動類型的範圍，請參閱[浮動限制](#)）。如果原始值無法精確地表示，則可以轉換成下一個較高或下一個較低的可顯示值。如果沒有這樣的值，則會產生未定義的結果。請考慮下列範例：

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

類型所能表示的最大值 `float` 為 3.402823466 3.402823 e 38-比1E300 更小的數位。因此，數位會轉換成無限大，而結果會是 "inf"。

## 整數與浮點類型之間的轉換

某些運算式可能會導致浮點類型的物件轉換成整數類型，反之亦然。當整數類資料類型的物件轉換成浮點類型，且原始值無法精確地表示時，結果會是下一個較高或下一個較低的可顯示值。

當浮動類型的物件轉換成整數類資料類型時，分數部分會被截斷，或四捨五入到零。例如1.3 的數位會轉換成1，而-1.3 會轉換為-1。如果截斷的值高於最大可表示的值，或小於最小的可顯示值，則結果為未定義。

## 算術轉換

許多二元運算子（在[具有二元運算子的運算式](#)中討論）會導致運算元的轉換，並以相同的方式產生結果。這些運算子的轉換會被稱為一般算術轉換。具有不同原生類型之運算元的算術轉換會依照下表所示完成。Typedef 類型是根據其基礎原生類型而運作。

### 類型轉換的條件

任一個運算元的類型為 <code>long double</code> 。	其他運算元會轉換成類型 <code>long double</code> 。
不符合上述條件，且任一個運算元的類型為 <code>double</code> 。	其他運算元會轉換成類型 <code>double</code> 。
不符合上述條件，且任一個運算元的類型為 <code>float</code> 。	其他運算元會轉換成類型 <code>float</code> 。

不符合上述條件 (所有運算元都不是浮動類型)。

運算元會取得整數提升, 如下所示:

- 如果任一個運算元的類型為 `unsigned long`, 則會將另一個運算元轉換成類型 `unsigned long`。
- 如果不符上述條件, 而且任一運算元的類型為, 而另一個運算元的類型為 `long` `unsigned int`, 則兩個運算元都會轉換成類型 `unsigned long`。
- 如果不符上述兩個條件, 而且任一運算元的類型為 `long`, 則會將另一個運算元轉換成類型 `long`。
- 如果不符上述三個條件, 而且任一運算元的類型為 `unsigned int`, 則會將另一個運算元轉換成類型 `unsigned int`。
- 如果不符上述任何條件, 則會將兩個運算元轉換成類型 `int`。

下列程式碼說明表格中所述的這些轉換規則:

```
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;
}
```

在上述範例中的第一個陳述式會顯示兩個整數類資料類型 (`iVal` 和 `ulVal`) 的乘法。符合的條件為兩個運算元都不是浮動類型, 而一個運算元的類型為 `unsigned int`。因此, 另一個運算元 `iVal` 會轉換成類型 `unsigned int`。然後會將結果指派給 `dVal`。這裡符合的條件是一個運算元的型別為 `double`, 因此 `unsigned int` 乘法的結果會轉換成型別 `double`。

上述範例中的第二個語句會顯示加上 `float` 和整數類資料類型: `fVal` 和 `ulVal`。`ulVal` 變數會轉換成類型 `float` (資料表中的第三個條件)。加法的結果會轉換成類型 `double` (資料表中的第二個條件), 並指派給 `dVal`。

## 指標轉換

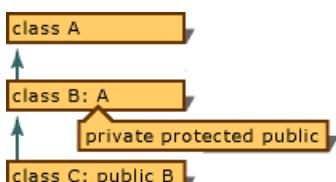
在執行指派、初始化、比較和其他運算式時, 可以轉換指標。

### 類別的指標

在兩種情況下, 類別的指標可以轉換成基底類別的指標。

第一種情況是可存取指定的基底類別, 而且轉換明確。如需不明確的基類參考的詳細資訊, 請參閱[多個基類](#)。

基底類別是否可存取, 取決於衍生中所使用的繼承種類。請參考下圖中說明的繼承。



## 說明基底類別存取範圍的繼承圖表

下表說明圖中所示情況的基底類別存取範圍。

基底類別 存取範圍	子類別 存取範圍	是否能 夠存取
外部 (非類別範圍) 函式	Private	否
	Protected	否
	公開	是
B 成員函式 (B 範圍中)	Private	是
	Protected	是
	公開	是
C 成員函式 (C 範圍中)	Private	否
	Protected	是
	公開	是

類別指標可以轉換成基底類別指標的第二種情況，是使用明確類型轉換。如需明確類型轉換的詳細資訊，請參閱[明確類型轉換運算子](#)。

這類轉換的結果是子物件的指標，這是由基類完整描述的物件部分。

下列程式碼會定義兩種類別 `A` 和 `B`，其中 `B` 衍生自 `A`（如需繼承的詳細資訊，請參閱[衍生類別](#)）。它接著會定義 `bObject`、類型的物件 `B`，以及指向物件的兩個指標（`pA` 和 `pB`）。

```
// 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
}
```

`pA` 指標是 `A *` 類型，可以解譯為表示「`A` 類型物件的指標」。的成員 `bObject` (例如 `BComponent` 和 `BMemberFunc`) 對於類型是唯一的 `B`，因此無法透過存取 `pA`。`pA` 指標只允許存取 `A` 類別中定義的那些物件特性 (成員函式和資料)。

## 函式的指標

`void *` 如果類型 `void *` 夠大，足以容納該指標，則函式的指標可以轉換成類型。

## void 的指標

類型的指標 `void` 可以轉換成任何其他類型的指標，但只能使用明確類型轉換 (不同于 C)。任何類型的指標都可以隱含地轉換成類型的指標 `void`。類型之不完整物件的指標可以轉換為 `void` (隱含) 和反向 (明確) 的指標。這類轉換的結果等於原始指標的值。如果物件已宣告，但沒有足夠的可用資訊來判斷其大小或基類，則會將它視為不完整。

任何不是 `const` 或 `volatile` 可以隱含地轉換為類型指標的物件指標 `void *`。

## const 和 volatile 指標

C++ 不會提供從 `const` 或 `volatile` 類型到不是或之類型的標準轉換 `const` `volatile`。不過，使用明確類型轉型 (包括不安全的轉換) 即可指定任何類型的轉換。

### NOTE

成員的 C++ 指標 (靜態成員指標除外) 與一般指標不同，而且沒有相同的標準轉換。靜態成員的指標是一般指標，具有與一般指標相同的轉換。

## null 指標轉換

評估為零的整數常數運算式，或轉換成指標類型的運算式會轉換成稱為 `null` 指標的指標。這個指標一律會比較不等於任何有效物件或函式的指標。例外狀況是物件的指標，可以具有相同的位移，而且仍然指向不同的物件。

在 C++11 中，`nullptr` 類型應該慣用於 C 樣式的 null 指標。

## 指標運算式轉換

具有陣列類型的任何運算式都可以轉換成相同類型的指標。轉換的結果是第一個陣列元素的指標。下列範例將示範這類轉換：

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

導致函式傳回特殊類型的運算式會轉換成傳回該類型之函式的指標，但下列情形例外：

- 運算式用來做為傳址運算子 () 的運算元 `&`。
- 運算式做為函式呼叫運算子的運算元。

## 參考轉換

在這些情況下，可以將類別的參考轉換為基類的參考：

- 可存取指定的基類。
- 轉換不可以模稜兩可。(如需不明確的基類參考的詳細資訊，請參閱 [多個基類](#))。

轉換的結果是表示基底類別子物件的指標。

## 成員的指標

在執行指派、初始化、比較和其他運算式時，可以轉換類別成員的指標。本節將描述下列成員指標的轉換：

### 基底類別成員的指標

當下列條件符合時，就可以將基底類別成員的指標轉換為從其衍生之類別成員的指標：

- 反向轉換 (即從衍生類別的指標轉換為基底類別指標) 是可以存取的。
- 衍生的類別無法由基底類別進行虛擬繼承。

當左運算元為成員的指標時，右運算元必須是成員指標類型，或者必須是判斷值為 0 的常數運算式。這項指派只在下列情況下有效：

- 右運算元為與左運算元相同類別的成員指標。
- 左運算元是一種公開且明確地從右運算元類別所衍生的類別成員指標。

### 成員轉換的 null 指標

評估為零的整數常數運算式會轉換成 null 指標。這個指標一律會比較不等於任何有效物件或函式的指標。例外狀況是物件的指標，可以具有相同的位移，而且仍然指向不同的物件。

下列程式碼說明在 `i` 類別中針對成員 `A` 所定義的指標。指標 (即 `pai`) 已初始化為 0，為 Null 指標。

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

int A::*pai = 0;

int main()
{
}
```

## 另請參閱

[C++ 語言參考](#)

# 內建類型 ( C + + )

2020/11/2 • [Edit Online](#)

內建類型(也稱為基本類型)是由 c + + 語言標準所指定，並內建于編譯器中。內建類型並未定義于任何標頭檔中。內建類型分為三個主要類別：整數、浮點數和 `void`。整數類資料類型代表整數。浮點類型可以指定可能有小數部分的值。編譯器會將大部分的內建類型視為不同的類型。不過，某些類型是同義字，或由編譯器視為對等類型。

## Void 類型

此 `void` 類型描述一組空的值。無法指定類型的變數 `void`。`void` 類型主要是用來宣告不傳回任何值的函式，或宣告不具類型或任意類型資料的泛型指標。任何運算式都可以明確轉換或轉換成類型 `void`。不過，這類運算式僅限於下列用法：

- 運算陳述式 (如需詳細資訊，請參閱[運算式](#))。
- 逗號運算子的左運算元 (如需詳細資訊，請參閱[逗號運算子](#))。
- 條件運算子 (`? :`) 的第二個或第三個運算元 (如需詳細資訊，請參閱[具有條件運算子的運算式](#))。

## `std::nullptr_t`

關鍵字 `nullptr` 是類型的 null 指標常數 `std::nullptr_t`，可以轉換成任何原始指標類型。如需詳細資訊，請參閱 [nullptr](#)。

## 布林值類型

`bool` 型別可以具有值 `true` 和 `false`。類型的大小 `bool` 為「執行特定」。如需 Microsoft 特定的執行詳細資料，請參閱[內建類型的大小](#)。

## 字元類型

`char` 型別是字元表示型別，可有效率地將基本執行字元集的成員編碼。C + + 編譯器會將類型為、和的變數視為 `char` `signed char` `unsigned char` 具有不同的類型。

Microsoft 專有：`char` `int` `signed char` 除非使用編譯選項，否則類型的變數會升級為，如同從類型的預設值一樣 [/J](#)。在此情況下，它們會被視為型別 `unsigned char`，而且會升級為 `int` 不含正負號的延伸。

類型的變數 `wchar_t` 是寬字元或多位元組字元類型。在 `L` 字元或字串常值之前使用前置詞，以指定寬字元類型。

Microsoft 特定：根據預設，`wchar_t` 是原生類型，但您可以使用 [/Zc:wchar\\_t-](#) 來建立 `wchar_t` 的 `typedef unsigned short`。`_wchar_t` 類型是適用於原生類型的 Microsoft 特定同義字 `wchar_t`。

`char8_t` 類型會用於 utf-8 字元標記法。它與具有相同的標記法 `unsigned char`，但編譯器會將其視為不同的類型。`char8_t` 類型在 c + + 20 中是新的。Microsoft 專有：使用 `char8_t` 需要 [/std:c++latest](#) 編譯器選項。

`char16_t` 類型用於 utf-16 字元標記法。它必須夠大，才能代表任何 UTF-16 程式碼單位。編譯器會將它視為不同的類型。

`char32_t` 類型用於 UTF-32 字元標記法。它必須夠大，才能代表任何 UTF-32 程式碼單位。編譯器會將它視

為不同的類型。

## 浮點類型

浮點類型會使用 IEEE-754 標記法，來提供範圍廣泛巨量的分數值近似值。下表列出 c++ 中的浮點類型，以及浮點類型大小的比較限制。這些限制是由 c++ 標準所規定，而且與 Microsoft 的實行無關。標準中不會指定內建浮點類型的絕對大小。

II	II
<code>float</code>	類型 <code>float</code> 是 c++ 中最小的浮點類型。
<code>double</code>	類型 <code>double</code> 是大於或等於類型 <code>float</code> ，但短於或等於類型大小的浮點類型 <code>long double</code> 。
<code>long double</code>	類型 <code>long double</code> 是大於或等於類型的浮點類型 <code>double</code> 。

Microsoft 特定：和的表示方式 `long double` `double` 完全相同。不過，`long double` `double` 編譯器會將視為不同的類型。Microsoft c++ 編譯器會使用4和8個位元組的 IEEE-754 浮點標記法。如需詳細資訊，請參閱[IEEE 浮點標記法](#)。

## 整數類型

此 `int` 類型是預設的基本整數類型。它可以代表實值範圍內的所有整數。

帶正負號的整數標記法是可以同時保留正值和負值的值。根據預設，或當修飾詞關鍵字存在時，會使用它 `signed`。`unsigned` 修飾詞關鍵字指定只能保存非負數值的不帶正負號表示。

大小修飾詞會指定所使用之整數表示的寬度(以位為單位)。語言支援 `short`、`long` 和修飾詞 `long long`。`short` 類型必須至少有16位寬。`long` 類型必須至少為32位寬。`long long` 類型必須至少為64位寬。標準會指定整數類型之間的大小關聯性：

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

執行必須同時維護每個類型的最小大小需求和大小關聯性。不過，實際的大小可能和在不同的執行之間有所不同。如需 Microsoft 特定的執行詳細資料，請參閱[內建類型的大小](#)。

`int` 當 `signed` 指定、或大小修飾詞時，可能會省略關鍵字 `unsigned`。修飾詞和 `int` 類型(如果有的話)可能會以任何順序出現。例如，`short unsigned` 和會 `unsigned int short` 參考相同的類型。

### 整數類型同義字

編譯器會將下列類型群組視為同義字：

- `short`, `short int`, `signed short`, `signed short int`
- `unsigned short`, `unsigned short int`
- `int`, `signed`, `signed int`
- `unsigned`, `unsigned int`
- `long`, `long int`, `signed long`, `signed long int`
- `unsigned long`, `unsigned long int`
- `long long`, `long long int`, `signed long long`, `signed long long int`

- `unsigned long long`, `unsigned long long int`

Microsoft 特定的整數類型包含特定寬度 `_int8`、`_int16`、`_int32` 和 `_int64` 類型。這些類型可以使用和修飾詞 `signed` `unsigned`。`_int8` 資料類型與類型同義，是與類型同義、與類型同義，`char` `_int16` 而且與 `short` `_int32` `int` `_int64` 類型同義 `long long`。

## 內建類型的大小

大部分的內建類型都具有實作為定義的大小。下表列出 Microsoft C++ 內建類型所需的儲存空間數量。特別是，`long` 即使在 64 位元作業系統上，也是 4 個位元組。

類型	大小
<code>bool</code> , <code>char</code> , <code>char8_t</code> , <code>unsigned char</code> , <code>signed char</code> , <code>_int8</code>	1 個位元組
<code>char16_t</code> , <code>_int16</code> , <code>short</code> , <code>unsigned short</code> , <code>wchar_t</code> , <code>_wchar_t</code>	2 個位元組
<code>char32_t</code> , <code>float</code> , <code>_int32</code> , <code>int</code> , <code>unsigned int</code> , <code>long</code> , <code>unsigned long</code>	4 個位元組
<code>double</code> , <code>_int64</code> , <code>long double</code> , <code>long long</code> , <code>unsigned long long</code>	8 個位元組

如需每個類型值範圍的摘要，請參閱[資料類型範圍](#)。

如需類型轉換的詳細資訊，請參閱[標準轉換](#)。

## 另請參閱

[資料類型範圍](#)

# 資料類型範圍

2020/11/2 • [Edit Online](#)

Microsoft c + + 32 位和64位編譯器會辨識本文稍後的表格中的類型。

- `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` )

如果其名稱開頭為兩個底線 (`_`), 則資料類型是非標準的。

下表中指定的範圍是兩端皆包含。

範圍	範圍	範圍	範圍
<code>int</code>	4	<code>signed</code>	-2,147,483,648 至 2,147,483,647
<code>unsigned int</code>	4	<code>unsigned</code>	0 到 4,294,967,295
<code>_int8</code>	1	<code>char</code>	-128 到 127
<code>unsigned _int8</code>	1	<code>unsigned char</code>	0 至 255
<code>_int16</code>	2	<code>short</code> , <code>short int</code> , <code>signed short int</code>	-32,768 至 32,767
<code>unsigned _int16</code>	2	<code>unsigned short</code> , <code>unsigned short int</code>	0 到 65,535
<code>_int32</code>	4	<code>signed</code> , <code>signed int</code> , <code>int</code>	-2,147,483,648 至 2,147,483,647
<code>unsigned _int32</code>	4	<code>unsigned</code> , <code>unsigned int</code>	0 到 4,294,967,295
<code>_int64</code>	8	<code>long long</code> , <code>signed long long</code>	- 9,223,372,036,854,775,808 至 9,223,372,036,854,775,807

IEEE	IEEE	IEEE	IEEE
<code>unsigned __int64</code>	8	<code>unsigned long long</code>	0 到 18,446,744,073,709,551,61 5
<code>bool</code>	1	無	<code>false</code> 或** <code>true</code>
<code>char</code>	1	無	-預設為 -128 到 127 使用編譯時，為 0 到 255 <a href="#">/J</a>
<code>signed char</code>	1	無	-128 到 127
<code>unsigned char</code>	1	無	0 至 255
<code>short</code>	2	<code>short int</code> , <code>signed short int</code>	-32,768 至 32,767
<code>unsigned short</code>	2	<code>unsigned short int</code>	0 到 65,535
<code>long</code>	4	<code>long int</code> , <code>signed long int</code>	-2,147,483,648 至 2,147,483,647
<code>unsigned long</code>	4	<code>unsigned long int</code>	0 到 4,294,967,295
<code>long long</code>	8	無(但相當於 <code>__int64</code> )	- 9,223,372,036,854,775,808 至 9,223,372,036,854,775,807
<code>unsigned long long</code>	8	無(但相當於 <code>unsigned __int64</code> )	0 到 18,446,744,073,709,551,61 5
<code>enum</code>	視情況而異	無	
<code>float</code>	4	無	3.4E +/- 38 (7 位數)
<code>double</code>	8	無	1.7E +/- 308 (15 位數)
<code>long double</code>	與相同** <code>double</code> **	無	與相同** <code>double</code> **
<code>wchar_t</code>	2	<code>__wchar_t</code>	0 到 65,535

視使用的方式而定，的變數會 `__wchar_t` 指定寬字元類型或多位元組字元類型。在字元或字串常數之前使用 `L` 前置詞可指定寬字元類型常數。

`signed` 和 `unsigned` 是可以與任何整數類資料類型搭配使用的修飾詞，但除外 `bool`。請注意 `char`，  
`signed char` 和 `unsigned char` 是三種不同類型，用於多載和範本等機制。

`int` 和 `unsigned int` 類型的大小為四個位元組。不過，可移植的程式碼不應依賴的大小，`int` 因為語言標準允許此功能成為特定執行。

Visual Studio 中的 C/C++ 也支援具大小的整數類型。如需詳細資訊，請參閱 [\\_int8, \\_int16, \\_int32, \\_int64](#) 和 [整數限制](#)。

如需每個類型之大小限制的詳細資訊，請參閱 [內建類型](#)。

列舉類型的範圍會根據語言內容和指定的編譯器旗標而變更。如需詳細資訊，請參閱 [C 列舉宣告](#) 和 [列舉](#)。

## 另請參閱

[關鍵字](#)

[內建類型](#)

# nullptr

2020/11/2 • [Edit Online](#)

`nullptr` 關鍵字會指定類型為的 null 指標常數 `std::nullptr_t`，這會轉換成任何原始指標類型。雖然您可以使用關鍵字 `nullptr`，而不包含任何標頭，但如果您的程式碼使用類型 `std::nullptr_t`，則必須包含標頭來定義它 `<cstddef>`。

## NOTE

`nullptr` 關鍵字也是在 managed 程式碼應用程式的 C++/CLI 中定義，而且無法與 ISO Standard C++ 關鍵字互換。如果您的程式碼可能會使用 `/clr` 以 managed 程式碼為目標的編譯器選項進行編譯，則請 `__nullptr` 在任何行程式碼中使用，您必須保證編譯器會使用原生 C++ 轉譯。如需詳細資訊，請參閱 `nullptr` (C++/CLI 和 C++/CX)。

## 備註

請避免使用 `NULL` 或零 (`0`) 做為 null 指標常數；`nullptr` 較不容易誤用，而且在大部分情況下效果較佳。例如，若使用 `func(std::pair<const char *, double>)`，呼叫 `func(std::make_pair(NULL, 3.14))` 就會造成編譯器錯誤。宏 `NULL` 會展開為 `0`，因此呼叫會傳回 `std::make_pair(0, 3.14)` `std::pair<int, double>`，而這無法轉換成 `std::pair<const char *, double>` 中的參數類型 `func`。呼叫 `func(std::make_pair(nullptr, 3.14))` 可以成功編譯，因為 `std::make_pair(nullptr, 3.14)` 會傳回 `std::pair<std::nullptr_t, double>` (可轉換為 `std::pair<const char *, double>`)。

## 另請參閱

### 關鍵字

`nullptr` (C++/CLI 和 C++/CX)

# void (C++)

2020/11/2 • [Edit Online](#)

當做函式傳回型別使用時，`void` 關鍵字會指定函式不會傳回值。當用於函式的參數清單時，`void` 會指定函式不接受任何參數。在指標的宣告中使用時，`void` 會指定指標為「通用」。

如果指標的類型為`void * **`，指標可以指向未使用或關鍵字宣告的任何變數 `const` `volatile`。除非將 `* void` 指標轉換成另一種類型，否則無法將其解除引用。`Void * **`指標可以轉換成任何其他類型的資料指標。

`void` 指標可以指向函式，而不是 C++ 中的類別成員。

您不能宣告類型為的變數 `void`。

## 範例

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

## 另請參閱

[關鍵字](#)

[內建類型](#)

# bool (C++)

2020/11/2 • [Edit Online](#)

這個關鍵字是內建類型。此類型的變數可以具有值 `true` 和 `false`。條件運算式的類型為 `bool`，因此具有類型的值 `bool`。例如，`i != 0` 現在具有 `true` 或，`false` 視的值而定 `i`。

Visual Studio 2017 15.3 和更新版本 (適用於 /std: c++17)：後置或前置遞增或遞減運算子的運算元不可以是類型 `bool`。換句話說，假設有一個類型為 `b` 的變數 `bool`，則不再允許這些運算式：

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

值 `true` 和 `false` 具有下列關聯性：

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

在下列陳述式中：

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

如果 `condexpr1` 為 `true`，`statement1` 則一律會執行；如果 `condexpr1` 為 `false`，則永遠不會 `statement1` 執行。

當後置或前置 `++` 運算子套用至類型的變數時 `bool`，變數會設定為 `true`。

Visual Studio 2017 15.3 版和更新版本：`operator++` for `bool` 已從語言中移除，不再受到支援。

後置或前置 `--` 運算子不能套用至這個類型的變數。

此 `bool` 類型會參與預設的整數提升。類型的 `r` 值 `bool` 可以轉換成類型的 `r` 值，而變成零，`int` `false` 而且 `true` 變成一個。作為相異類型，會 `bool` 參與多載解析。

## 另請參閱

[關鍵字](#)

[內建類型](#)

# false (C++)

2020/11/2 • [Edit Online](#)

關鍵字是 `bool` 類型變數或條件運算式 (條件運算式現在是布林運算式) 的兩個值之一 `true`。例如，如果 `i` 是類型的變數 `bool`，則語句會將 `i = false;` 指派 `false` 給 `i`。

## 範例

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

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

```
1
0
```

## 另請參閱

[關鍵字](#)

# true (C++)

2020/11/2 • [Edit Online](#)

## 語法

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

## 備註

這個關鍵字是 `bool` 類型變數或條件運算式 (條件運算式現在是 `true` 布林運算式) 的兩個值之一。如果 `i` 的型別為，則語句會將 `bool i = true;` 指派 `true` 給 `i`。

## 範例

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

```
1
0
```

## 另請參閱

[關鍵字](#)

# char、wchar\_t、char16\_t、char32\_t

2020/11/2 • [Edit Online](#)

型別 `char`、`wchar_t`、`char16_t` 和 `char32_t` 都是內建型別，代表英數位元，以及非英數位元和非列印字元。

## 語法

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

## 備註

`char` 類型是 C 和 c++ 中的原始字元類型。類型 `unsigned char` 通常用來表示 *byte*，這不是 c++ 中的內建類型。此 `char` 類型可用來儲存 ASCII 字元集或任何 ISO-8859 字元集的字元，以及多位元組字元的個別位元組（例如，Shift-JIS 或 Unicode 字元集的 utf-8 編碼）。類型的字串 `char` 稱為 *窄字串*，即使用來編碼多位元組字元亦然。在 Microsoft 編譯器中，`char` 是 8 位類型。

`wchar_t` 類型是實作為定義的寬字元類型。在 Microsoft 編譯器中，它代表一個 16 位的寬字元，用來將 Unicode 編碼為 UTF-16LE，這是 Windows 作業系統上的原生字元類型。通用 C 執行時間(UCRT)程式庫函式的寬字元版本會使用 `wchar_t` 和其指標和陣列類型做為參數和傳回值，如同原生 WINDOWS API 的寬字元版本。

`char16_t` 和 `char32_t` 類型分別代表 16 位和 32 位寬字元。以 UTF-16 編碼的 unicode 可以儲存在型別中 `char16_t`，而 unicode 編碼為 utf-32 可以儲存在型別中 `char32_t`。這些類型的字串和 `wchar_t` 全都稱為 *寬字串*，不過，這一詞通常會特別參考類型的字串 `wchar_t`。

在 c++ 標準程式庫中，`basic_string` 類型是針對窄和寬字元串特製化。當字元的類型為、字元的類型為、字元的類型為，`std::string`、`char`、`std::u16string`、`char16_t`、`std::u32string`、`char32_t` 且 `std::wstring` 字元的類型 `wchar_t` 為時，請使用。代表文字的其他類型，包括 `std::stringstream` 並 `std::cout` 具有窄和寬字元串的特製化。

# `_int8`、`_int16`、`_int32`、`_int64`

2020/11/2 • [Edit Online](#)

## Microsoft 特定

Microsoft C/C++ 的功能支援可調整大小的整數類型。您可以使用類型規範來宣告8、16、32或64位整數變數 `_intN`，其中 `N` 是8、16、32或64。

下列範例為其中每個可調整大小整數類型宣告一個變數：

```
_int8 nSmall;      // Declares 8-bit integer
_int16 nMedium;    // Declares 16-bit integer
_int32 nLarge;     // Declares 32-bit integer
_int64 nHuge;      // Declares 64-bit integer
```

類型 `_int8`、`_int16` 和 `_int32` 是具有相同大小之 ANSI 類型的同義字，適用於撰寫可在多個平臺上進行相同行為的可移植程式碼。`_int8` 資料類型與類型同義，與 `char` 類型同義，而且與 `_int16` `short` `_int32` 類型同義 `int`。`_int64` 類型與類型同義 `long long`。

為了與舊版相容，`_int8`、`_int16` 和 `_int32` 和的同義字 `_int16`、`_int32`、`_int64` 除非指定了編譯器選項 `/za` (停用語言擴充功能)。

## 範例

下列範例顯示 `_intN` 參數會升級為 `int`：

```
// sized_int_types.cpp

#include <stdio.h>

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

int main()
{
    _int8 i8 = 100;
    func(i8);    // no void func(_int8 i8) function
                 // _int8 will be promoted to int
}
```

```
func
```

## 另請參閱

[關鍵字](#)

[內建類型](#)

[資料類型範圍](#)

# \_\_m64

2020/11/2 • • [Edit Online](#)

## Microsoft 特定的

**\_\_m64** 資料類型可搭配 MMX 和3DNow 使用！內建函式和定義于中 <xmmmintrin.h> 。

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

## 備註

您不應該直接存取 **\_\_m64** 欄位。不過，可以在偵錯工具中看到這些類型。類型為的變數會 **\_\_m64** 對應至 MM [0-7] 暫存器。

**\_M64** 類型的變數會自動對齊8個位元組的界限。

**\_\_m64** X64 處理器不支援資料類型。將 **\_m64** 做為 MMX 內建一部分使用的應用程式，必須予以重新撰寫，才能使用對等的 SSE 和 SSE2 內建。

結束 Microsoft 專有

## 另請參閱

[關鍵字](#)

[內建類型](#)

[資料類型範圍](#)

# \_\_m128

2020/11/2 • [Edit Online](#)

## Microsoft 特定的

與 Streaming SIMD Extensions 和 Streaming SIMD Extensions 2 指令內建函式搭配使用的 \_\_m128 資料類型，是在中定義 <xmmmintrin.h> 。

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

## 備註

您不應該直接存取 \_\_m128 欄位。不過，可以在偵錯工具中看到這些類型。類型為的變數會 \_\_m128 對應至 XMM [0-7] 暫存器。

類型的變數會 \_\_m128 自動對齊16位元組的界限。

\_\_m128 ARM 處理器不支援資料類型。

結束 Microsoft 專有

## 另請參閱

[關鍵字](#)

[內建類型](#)

[資料類型範圍](#)

# \_\_m128d

2020/11/2 • • [Edit Online](#)

## Microsoft 特定的

`__m128d` 資料類型(與 Streaming SIMD Extensions 2 個指示內建函式搭配使用)定義于中 `<emmintrin.h>`。

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

## 備註

您不應該直接存取 `__m128d` 欄位。不過，可以在偵錯工具中看到這些類型。類型為的變數會 `__m128` 對應至 XMM [0-7] 暫存器。

`_M128d` 類型的變數會自動對齊16位元組的界限。

`__m128d` ARM 處理器不支援資料類型。

結束 Microsoft 專有

## 另請參閱

[關鍵字](#)

[內建類型](#)

[資料類型範圍](#)

# `_m128i`

2020/11/2 • [Edit Online](#)

## Microsoft 特定的

`_m128i` 與 Streaming SIMD Extensions 2 (SSE2) 指示內建函式搭配使用的資料類型，是在中定義 `<emmintrin.h>`。

```
// data_types_m128i.cpp
#include <emmintrin.h>
int main() {
    _m128i x;
}
```

## 備註

您不應該直接存取 `_m128i` 欄位。不過，可以在偵錯工具中看到這些類型。類型為的變數會 `_m128i` 對應至 XMM [0-7] 暫存器。

類型的變數會 `_m128i` 自動對齊16位元組的界限。

### NOTE

使用類型的變數 `_m128i` 會導致編譯器產生 SSE2 `movdqa` 指令。此指令不會造成 Pentium III 處理器上的錯誤，但會導致無訊息失敗，而且可能會造成副作用，因為任何指示都轉譯 `movdqa` 在 PENTIUM III 處理器上。

`_m128i` ARM 處理器不支援資料類型。

結束 Microsoft 專有

## 另請參閱

[關鍵字](#)

[內建類型](#)

[資料類型範圍](#)

# `_ptr32`、`_ptr64`

2020/11/2 • [Edit Online](#)

## Microsoft 特定的

`_ptr32` 表示32位系統上的原生指標，而 `_ptr64` 表示64位系統上的原生指標。

下列範例將示範如何宣告每一個這些類型的指標：

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

在32位系統上，使用宣告的指標 `_ptr64` 會被截斷為32位指標。在64位系統上，使用宣告的指標 `_ptr32` 會強制轉型為64位指標。

### NOTE

使用 `_ptr32` `_ptr64 /clr: pure` 進行編譯時，您無法使用或。否則，將會產生編譯器錯誤 C2472。`/Clr: pure` 和 `/clr: safe` 編譯器選項在 Visual Studio 2015 中已被取代，在 Visual Studio 2017 中不支援。

為了與舊版相容，`_ptr32` 和 `_ptr64` 都是和的同義字，`_ptr32` `_ptr64` 除非指定了編譯器選項 `/za` 停用（語言擴充功能）。

## 範例

下列範例顯示如何使用和關鍵字宣告和配置指標 `_ptr32` `_ptr64`。

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

int main()
{
    using namespace std;

    int * __ptr32 p32;
    int * __ptr64 p64;

    p32 = (int * __ptr32)malloc(4);
    *p32 = 32;
    cout << *p32 << endl;

    p64 = (int * __ptr64)malloc(4);
    *p64 = 64;
    cout << *p64 << endl;
}
```

32  
64

結束 Microsoft 專有

另請參閱

內建類型

# 數值限制 (C++)

2020/3/25 • [Edit Online](#)

這兩個標準包含檔案, <限制.h> 和 <float.h>、定義數值限制, 或指定類型的變數可以保留的最小和最大值。這些最小和最大的保證可以移植到C++使用與ANSI C相同資料標記法的任何編譯器。<限制.h> include 檔案會定義**整數類資料類型的數值限制**, 而 <float.h> 會定義**浮動類型的數值限制**。

## 另請參閱

[基本概念](#)

# 整數限制

2020/11/2 • [Edit Online](#)

## Microsoft 特定

下表列出整數類型的限制。當您包含標準標頭檔時，也會定義這些限制的預處理器宏 `<climits>`。

## 整數常數的限制

宏	說明	值
<code>CHAR_BIT</code>	不是位元欄位之最小變數中的位元數目。	8
<code>SCHAR_MIN</code>	類型變數的最小值 <code>signed char</code> 。	-128
<code>SCHAR_MAX</code>	類型變數的最大值 <code>signed char</code> 。	127
<code>UCHAR_MAX</code>	類型變數的最大值 <code>unsigned char</code> 。	255 (0xff)
<code>CHAR_MIN</code>	類型變數的最小值 <code>char</code> 。	-128;如果使用選項，則為 0 <code>/J</code>
<code>CHAR_MAX</code>	類型變數的最大值 <code>char</code> 。	127;255(如果 <code>/J</code> 使用選項)
<code>MB_LEN_MAX</code>	多重字元常數中位元組數目的上限。	5
<code>SHRT_MIN</code>	類型變數的最小值 <code>short</code> 。	-32768
<code>SHRT_MAX</code>	類型變數的最大值 <code>short</code> 。	32767
<code>USHRT_MAX</code>	類型變數的最大值 <code>unsigned short</code> 。	65535 (0xffff)
<code>INT_MIN</code>	類型變數的最小值 <code>int</code> 。	-2147483648
<code>INT_MAX</code>	類型變數的最大值 <code>int</code> 。	2147483647
<code>UINT_MAX</code>	類型變數的最大值 <code>unsigned int</code> 。	4294967295 (0xffffffff)
<code>LONG_MIN</code>	類型變數的最小值 <code>long</code> 。	-2147483648
<code>LONG_MAX</code>	類型變數的最大值 <code>long</code> 。	2147483647
<code>ULONG_MAX</code>	類型變數的最大值 <code>unsigned long</code> 。	4294967295 (0xffffffff)
<code>LLONG_MIN</code>	類型變數的最小值** <code>long long</code> **	-9223372036854775808
<code>LLONG_MAX</code>	類型變數的最大值** <code>long long</code> **	9223372036854775807

ULLONG_MAX	類型變數的最大值** unsigned long long **	18446744073709551615 (0xffffffffffffffffffff)

如果值超過最大的整數表示，Microsoft 編譯器會產生錯誤。

## 另請參閱

[浮動限制](#)

# 浮點數限制

2020/4/15 • [Edit Online](#)

## Microsoft 特定的

下表列出浮點常數值的限制。這些限制也在標準頭檔<float.h>中定义。

## 浮點常數的限制

名稱	說明	值
<code>FLT_DIG</code> <code>DBL_DIG</code> <code>LDBL_DIG</code>	位數 $q$ , 使得有 $q$ 個小數位數的浮點數可以捨入為浮點表示 (反向亦然), 而不會失去精確度。	6 15 15
<code>FLT_EPSILON</code> <code>DBL_EPSILON</code> <code>LDBL_EPSILON</code>	最小正數 $x$ , 使得 $x + 1.0$ 不會等於 $1.0$ 。	1.192092896e-07F 2.2204460492503131e-016 2.2204460492503131e-016
<code>FLT_GUARD</code>		0
<code>FLT_MANT_DIG</code> <code>DBL_MANT_DIG</code> <code>LDBL_MANT_DIG</code>	在浮點符號中指定的 <code>FLT_RADIX</code> 半徑中的位數。半徑為 2;因此,這些值指定位。	24 53 53
<code>FLT_MAX</code> <code>DBL_MAX</code> <code>LDBL_MAX</code>	最大可表示浮點數。	3.402823466e+38F 1.7976931348623158e+308 1.7976931348623158e+308
<code>FLT_MAX_10_EXP</code> <code>DBL_MAX_10_EXP</code> <code>LDBL_MAX_10_EXP</code>	最多整數,以便 10 個凸起到該數位是可表示的浮點數。	38 308 308
<code>FLT_MAX_EXP</code> <code>DBL_MAX_EXP</code> <code>LDBL_MAX_EXP</code>	<code>FLT_RADIX</code> 升到該數位的最大整數是可表示的浮點數。	128 1024 1024
<code>FLT_MIN</code> <code>DBL_MIN</code> <code>LDBL_MIN</code>	最小正值。	1.175494351e-38F 2.2250738585072014e-308 2.2250738585072014e-308
<code>FLT_MIN_10_EXP</code> <code>DBL_MIN_10_EXP</code> <code>LDBL_MIN_10_EXP</code>	最小負整數,以便 10 提升到該數位是可表示的浮點數。	-37 -307 -307
<code>FLT_MIN_EXP</code> <code>DBL_MIN_EXP</code> <code>LDBL_MIN_EXP</code>	升到該數位的 <code>FLT_RADIX</code> 最小負整數是可表示的浮點數。	-125 -1021 -1021
<code>FLT_NORMALIZE</code>		0

<code>FLT_RADIX</code> <code>_DBL_RADIX</code> <code>_LDBL_RADIX</code>	指數表示的基數。	2 2 2
<code>FLT_ROUNDS</code> <code>_DBL_ROUNDS</code> <code>_LDBL_ROUNDS</code>	用於浮點添加的舍入模式。	1 (接近) 1 (接近) 1 (接近)

#### NOTE

上表中的資訊在未來的產品版本中可能有所不同。

結束微軟的

另請參閱

[整數限制](#)

# 宣告和定義 ( C + + )

2020/11/2 • [Edit Online](#)

C + + 套裝程式含各種實體，例如變數、函式、類型和命名空間。這些實體中的每個都必須先行宣告才能使用。宣告會指定實體的唯一名稱，以及其類型和其他特性的相關資訊。在 c + + 中，宣告名稱的點是編譯器可看見的位置點。您不能參考在編譯單位稍後的某個時間點所宣告的函式或類別。變數應該在使用的點之前，盡可能地宣告為接近。

下列範例顯示一些宣告：

```
#include <string>

void f(); // forward declaration

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

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

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

在第5行上，已宣告函式 `main`。在第7行上，`const pi` 會宣告並初始化名為的變數。在第8行上，`i` 會使用函式所產生的值來宣告和初始化整數 `f`。`f` 因為第3行的向前宣告，編譯器可以看見名稱。

在第9行中，宣告了型別為 `obj` 的變數 `c`。不過，此宣告會引發錯誤，因為在 `c` 稍後的程式中不會宣告，而且不會向前宣告。若要修正錯誤，您可以將的整個定義移到 `c` 之前 `main` 或其他地方，為其新增正向宣告。這種行為與其他語言(例如 c #)不同，其中函式和類別可以在來源檔案中的宣告點之前使用。

在第10行中，宣告了型別為 `str` 的變數 `std::string`。名稱 `std::string` 是可見的，因為它是在 `string` 標頭檔中導入，而該檔案會合並到第1行的原始程式檔中。`std` 是用來宣告類別的命名空間 `string`。

在第11行中，因為尚未宣告名稱，所以會引發錯誤 `j`。宣告必須提供類型，與其他語言(例如 javaScript)不同。

在第12行中，`auto` 會使用關鍵字，這會指示編譯器根據其初始化的值推斷的類型 `k`。在此情況下，編譯器會選擇 `int` 類型的。

## 宣告範圍

宣告引進的名稱在宣告發生的範圍內是有效的。在上述範例中，在函式內宣告的變數 `main` 是區域變數。您可以 `i` 在全域範圍的 `main` 外部宣告另一個名為的變數，它會是完全不同的實體。不過，這類名稱的重複可能會導致程式設計人員混淆和錯誤，應予以避免。在第21行中，類別 `c` 是在命名空間的範圍內宣告 `N`。使用命名空間有助於避免名稱衝突。大部分的 c + + 標準程式庫名稱會在 `std` 命名空間中宣告。如需範圍規則如何與宣告互動的詳細資訊，請參閱範圍。

## 定義

某些實體(包括函式、類別、列舉和常數變數)除了要宣告外，還必須定義。定義會在程式中稍後使用實體時，為編譯器提供產生機器碼所需的所有資訊。在上述範例中，第3行包含函式的宣告，但函式 `f` 的定義在第15到18行中提供。在第21行，類別 `c` 是宣告和定義的(雖然定義的類別不會執行任何動作)。常數變數必須在其宣告所在的相同語句中，以其他單字指派值的方式定義。內建類型的宣告(例如)會 `int` 自動定義，因為編譯器會知道要為它配置多少空間。

下列範例會顯示也是定義的宣告：

```
// 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;  
};
```

以下是一些不是定義的宣告：

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

## Typedef 和 using 語句

在舊版的 C++ 中，`typedef` 關鍵字是用來宣告新名稱，這是另一個名稱的別名。例如，類型 `std::string` 是的另一個名稱 `std::basic_string<char>`。程式設計人員為何要使用 `typedef` 名稱，而不是實際名稱，這應該很明顯。在現代 C++ 中，`using` 關鍵字優先于 `typedef`，但其概念相同：已宣告並定義實體的新名稱。

## 靜態類別成員

因為靜態類別資料成員是由類別的所有物件共用的離散變數，所以必須在類別定義之外定義和初始化。(如需詳細資訊，請參閱 [類別](#))。

## extern 宣告

C++ 程式可能包含一個以上的編譯單位。若要宣告在不同編譯單位中定義的實體，請使用 `extern` 關鍵字。宣告中的資訊足以用於編譯器，但如果在連結步驟中找不到實體的定義，則連結器會引發錯誤。

## 本節內容

### 儲存類別

`const`  
`constexpr`  
`extern`

### 初始設定式

### 別名和 typedef

`using` 清點

`volatile`

`decltype`

C++ 中的屬性

另請參閱

基本概念

# 儲存類別

2020/11/2 • [Edit Online](#)

C + + 變數宣告內容中的儲存類別是一種類型規範，可控制物件的存留期、連結和記憶體位置。指定的物件只能有一個儲存類別。除非使用、或規範來指定，否則區塊內定義的變數會具有自動儲存區 `extern static` `thread_local`。自動物件和變數沒有連結；區塊外部的程式碼看不到它們。當執行進入區塊時，系統會自動為它們配置記憶體，並在結束區塊時解除配置。

## 備註

1. 可以將可變關鍵字視為儲存類別規範。不過，它只能在類別定義的成員清單中使用。
2. Visual Studio 2010 和更新版本：關鍵字不再是 `auto` C + + 儲存類別規範，而且 `register` 關鍵字已被取代。Visual Studio 2017 15.7 版和更新版本：(適用於 `/std:c++17`)：已 `register` 從 C + + 語言移除關鍵字。

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

## static

`static` 關鍵字可以用來宣告全域範圍、命名空間範圍和類別範圍的變數和函式。靜態變數也可以在區域範圍內進行宣告。

靜態持續期間是指，物件或變數會在程式啟動時配置，並在程式結束時解除配置。外部連結是指，變數名稱在變數宣告所在的檔案外部可見。相反地，內部連結是指，名稱在變數宣告所在的檔案外部不可見。根據預設，全域命名空間中所定義的物件或變數具有靜態持續期間和外部連結。`static` 關鍵字可用於下列情況。

1. 當您在檔案範圍（全域和/或命名空間範圍）宣告變數或函式時，`static` 關鍵字會指定變數或函數具有內部連結。當您宣告變數時，變數會擁有靜態持續期間，且編譯器會將其初始化為 0（除非您指定其他值）。
2. 當您在函式中宣告變數時，`static` 關鍵字會指定變數在呼叫該函式之間保留其狀態。
3. 當您在類別宣告中宣告資料成員時，`static` 關鍵字會指定該成員的一個複本由類別的所有實例共用。靜態資料成員必須以檔案範圍定義。您宣告為的整數資料成員 `const static` 可以有初始化運算式。
4. 當您在類別宣告中宣告成員函式時，`static` 關鍵字會指定該函式是由類別的所有實例共用。靜態成員函式無法存取實例成員，因為函數沒有隱含的 `this` 指標。若要存取執行個體成員，請使用本身為執行個體指標或參考的參數宣告函式。
5. 您無法將等位的成員宣告為靜態。不過，全域宣告的匿名等位必須明確宣告 `static`。

這個範例顯示在函式中宣告的變數如何 `static` 在呼叫該函式之間保留其狀態。

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

using namespace std;
void showstat( int curr ) {
    static int nStatic;      // Value of nStatic is retained
                            // between each function call
    nStatic += curr;
    cout << "nStatic is " << nStatic << endl;
}

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

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

這個範例示範如何 `static` 在類別中使用。

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

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

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

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

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

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

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

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

這個範例會顯示在成員函式中宣告的本機變數 `static`。`static` 變數可供整個程式使用；類型的所有實例會共用相同的 `static` 變數複本。

```
// static3.cpp  
// compile with: /EHsc  
#include <iostream>  
using namespace std;  
struct C {  
    void Test(int value) {  
        static int var = 0;  
        if (var == value)  
            cout << "var == value" << endl;  
        else  
            cout << "var != value" << endl;  
  
        var = value;  
    }  
};  
  
int main() {  
    C c1;  
    C c2;  
    c1.Test(100);  
    c2.Test(100);  
}
```

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

從 C++11 開始，`static` 本機變數初始化保證是安全線程。這項功能有時稱為魔術靜態。不過，在多執行緒應用程式中，必須同步處理所有後續指派。您可以使用旗標來停用安全線程靜態初始化功能 [/Zc:threadSafeInit-](#)，以避免依賴 CRT。

## extern

宣告為的物件和變數 `extern` 會宣告在另一個轉譯單位或封閉範圍中定義為具有外部連結的物件。如需詳細資訊，請參閱 [extern](#) 和 [轉譯單位和連結](#)。

## thread\_local (C++11)

使用指定名稱宣告的變數 `thread_local` 只能在其建立所在的執行緒上存取。變數會在建立執行緒時建立，並在終結執行緒時終結。每個執行緒都有它自己的變數複本。在 Windows 上，的 `thread_local` 功能相當於 Microsoft 特有的 [\\_\\_declspec\(thread\)](#) 屬性。

```

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;
}

```

規範的注意事項 `thread_local` :

- Dll 中的動態初始化執行緒區域變數可能無法在所有呼叫執行緒上正確初始化。如需詳細資訊，請參閱 [thread](#)。
- `thread_local` 規範可以與 `static` 或結合 `extern`。
- 您只能套用 `thread_local` 至資料宣告和定義；`thread_local` 不能用在函式宣告或定義上。
- 您只能 `thread_local` 在具有靜態儲存期的資料項目上指定。這包括全域資料物件（`static` 和 `extern`）、本機靜態物件，以及類別的靜態資料成員。`thread_local` 如果未提供其他儲存類別，則宣告的任何區域變數都會隱含為靜態，換言之，在區塊範圍 `thread_local` 相當於 `thread_local static`。
- 不論宣告和 `thread_local` 定義發生在相同的檔案還是不同的檔案中，您都必須針對執行緒區域物件的宣告和定義指定。

在 Windows 上，的 `thread_local` 功能相當於 `__declspec(thread)` 不同之處在於 `* __declspec(thread) *` 可以套用至類型定義，而且在 C 程式碼中有效。請盡可能使用 `thread_local` 因為它是 C++ 標準的一部分，因此更具可攜性。

## 參加

Visual Studio 2017 15.3 版和更新版本（適用於 `/std:c++17`）：`register` 關鍵字不再是支援的儲存類別。關鍵字仍然保留在標準中，供日後使用。

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

## 範例：自動與靜態初始化

區域自動物件或變數會在每次控制流程到達其定義時初始化。區域靜態物件或變數會在控制流程第一次到達其定義時初始化。

請考慮下列範例，範例中會定義記錄物件初始化和解構的類別，然後再定義三個物件 `I1`、`I2` 和 `I3`：

```

// initialization_of_objects.cpp
// compile with: /EHsc
#include <iostream>
#include <string.h>
using namespace std;

// Define a class that logs initializations and destructions.
class InitDemo {
public:
    InitDemo( const char *szWhat );
    ~InitDemo();

private:
    char *szObjName;
    size_t sizeofObjName;
};

// Constructor for class InitDemo
InitDemo::InitDemo( const char *szWhat ) :
szObjName(NULL), sizeofObjName(0) {
if ( szWhat != 0 && strlen( szWhat ) > 0 ) {
    // Allocate storage for szObjName, then copy
    // initializer szWhat into szObjName, using
    // secured CRT functions.
    sizeofObjName = strlen( szWhat ) + 1;

    szObjName = new char[ sizeofObjName ];
    strcpy_s( szObjName, sizeofObjName, szWhat );

    cout << "Initializing: " << szObjName << "\n";
}
else {
    szObjName = 0;
}
}

// Destructor for InitDemo
InitDemo::~InitDemo() {
if( szObjName != 0 ) {
    cout << "Destroying: " << szObjName << "\n";
    delete szObjName;
}
}

// Enter main function
int main() {
    InitDemo I1( "Auto I1" );
    cout << "In block.\n";
    InitDemo I2( "Auto I2" );
    static InitDemo I3( "Static I3" );
}
cout << "Exited block.\n";
}

```

```

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

```

這個範例示範如何和何時初始化物件 `I1`、`I2` 和 `I3` 以及何時終結它們。

關於此程式，有幾點需要注意：

- 首先，`I1` 和 `I2` 會在控制流程離開其定義所在的區塊時自動終結。
- 其次，在 C++ 中，不需要在區塊開頭宣告物件或變數。再者，只有在控制流程到達其定義時，這些物件才會初始化（`I2` 和 `I3` 是這類定義的範例）。輸出會顯示其初始化的確切時間。
- 最後，靜態區域變數（例如 `I3`）會在程式執行期間保留其值，不過會在程式終止時終結。

## 另請參閱

[宣告和定義](#)

# auto (C++)

2020/11/2 • [Edit Online](#)

從所宣告變數的初始化運算式，來推斷其類型。

## NOTE

C++ 標準定義此關鍵字的原始和修訂的意義。在 Visual Studio 2010 之前，`auto` 關鍵字會在 *自動儲存類別* 中宣告變數，也就是具有本機存留期的變數。從 Visual Studio 2010 開始，`auto` 關鍵字會宣告一個變數，該變數的型別是從其宣告中的初始化運算式推斷而來。[/Zc:auto](#) [ ] 編譯器選項控制關鍵字的意義 `auto`。

## 語法

```
auto declarator 告訴子 初始化運算式 ;
```

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

## 備註

`auto` 關鍵字會指示編譯器使用所宣告變數或 lambda 運算式參數的初始化運算式來推算其類型。

我們建議您在 `auto` 大部分情況下使用關鍵字，除非您真的想要轉換，因為它提供下列優點：

- **穩定性**：如果運算式的型別已變更（這包括當函式傳回型別變更時），就會運作。
- **效能**：您保證不會進行任何轉換。
- **可用性**：您不必擔心型別名稱拼寫問題和打字錯誤。
- **效率**：您的程式碼可能更有效率。

您可能不想使用的轉換案例 `auto`：

- 想要特定類型但不執行任何動作時。
- 運算式範本 helper 類型，例如 `(valarray+valarray)`。

若要使用 `auto` 關鍵字，請使用它來取代類型來宣告變數，並指定初始化運算式。此外，您也可以使用規範和宣告子來修改 `auto` 關鍵字，例如 `const`、`volatile`、指標 `(*)`、參考 `(&)` 和右值參考 `(&&)`。編譯器會評估初始化運算式，然後使用該資訊來推斷變數類型。

初始化運算式可以是指派（等號語法）、直接初始化（函式樣式語法）、`operator new` 運算式或初始化運算式可以是範圍架構的 `for` 語句 (C++) 語句中的範圍宣告參數。如需詳細資訊，請參閱此檔稍後的 [初始化運算式](#) 和程式碼範例。

`auto` 關鍵字是類型的預留位置，但本身不是類型。因此，`auto` 關鍵字不能用於 `sizeof` C++/cli) 的轉換或運算子，例如和 (`typeid`)。

## 實用性

`auto` 關鍵字是宣告具有複雜類型之變數的簡單方式。例如，您可以使用 `auto` 來宣告初始化運算式牽涉到範本、函式指標或成員指標的變數。

您也可以使用 `auto` 將變數宣告和初始化為 lambda 運算式。您無法自行宣告變數類型，因為只有編譯器才知道 Lambda 運算式的類型。如需詳細資訊，請參閱 [Lambda 運算式的範例](#)。

## 尾端傳回型別

您可以 `auto` 搭配類型規範使用，`decltype` 以協助撰寫範本程式庫。使用 `auto` 和宣告樣板函式，`decltype` 其傳回型別取決於其樣板引數的類型。或者，使用和來宣告包裝對另一個函式之呼叫的樣板函式 `auto`，然後傳回該其他函式的傳回 `decltype` 型別。如需詳細資訊，請參閱 [decltype](#)。

## 參考和 CV 限定詞

請注意，使用會捨棄 `auto` 參考、`const` 限定詞和 `volatile` 限定詞。請考慮下列範例：

```
// 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;
}
```

在上述範例中，`myAuto` 是 `int`，而不是 `int` 參考，因此輸出為，而不是在參考辨識符號尚未卸載的 `11 11  
11 12` 情況下 `auto`。

## 括弧初始化運算式的類型推斷 (C++ 14)

下列程式碼範例示範如何 `auto` 使用大括弧初始化變數。請注意 B 與 C 之間的差異，以及 A 與 E 之間的差異。

```
#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;
}
```

## 限制和錯誤訊息

下表列出使用關鍵字的限制 `auto`，以及編譯器發出的對應診斷錯誤訊息。

錯誤	說明
C3530	<code>auto</code> 關鍵字無法與任何其他類型規範結合。
C3531	使用關鍵字宣告的符號 <code>auto</code> 必須有初始化運算式。
C3532	您不正確地使用 <code>auto</code> 關鍵字來宣告型別。例如，您已宣告方法傳回類型或陣列。
C3533、C3539	無法使用關鍵字宣告參數或樣板引數 <code>auto</code> 。
C3535	方法或範本參數不可使用 <code>auto</code> 關鍵字宣告。
C3536	符號無法在初始化之前使用。實際上，這表示不能使用變數來初始化它自己。
C3537	您無法轉換成使用關鍵字宣告的型別 <code>auto</code> 。
C3538	宣告子清單中使用關鍵字宣告的所有符號都 <code>auto</code> 必須解析成相同的類型。如需詳細資訊，請參閱宣告 <a href="#">和定義</a> 。
C3540、C3541	<code>Sizeof</code> 和 <code>typeid</code> 運算子無法套用至使用關鍵字宣告的符號 <code>auto</code> 。

## 範例

這些程式碼片段說明 `auto` 可以使用關鍵字的一些方式。

下列宣告相同。在第一個語句中，變數宣告 `j` 為類型 `int`。在第二個語句中，會將變數推算 `k` 為類型 `int`，因為初始化運算式 (0) 是整數。

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

下列宣告相同，但第二個宣告比第一個宣告簡單。使用關鍵字最具說服力的原因之一 `auto`，是簡單的。

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

下列程式碼片段會宣告變數的類型 `iter`，以及 `elem` `for` 和範圍 `for` 迴圈開始時。

```

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

```

下列程式碼片段使用 `new` 運算子和指標宣告來宣告指標。

```

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

```

下一個程式碼片段會宣告每個宣告陳述式中的多個符號。請注意，每個陳述式中的所有符號都會解析成相同類型。

```

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.

```

此程式碼片段使用條件運算子 (`?:`) 將變數 `x` 宣告為值為 200 的整數：

```

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

```

下列程式碼片段會將變數初始化為類型、將變數初始化為 `x int y` 類型的參考，並將變數初始化為傳回類型之函式的 `const int fp` 指標 `int`。

```

int f(int x) { return x; }
int main()
{
    auto x = f(0);
    const auto& y = f(1);
    int (*p)(int x);
    p = f;
    auto fp = p;
    //...
}

```

[另請參閱](#)

## 關鍵字

/Zc:auto (推算變數類型)

sizeof 運算元

typeid

operator new

## 宣告和定義

Lambda 運算式的範例

## 初始設定式

decltype

# const (C++)

2020/11/2 • [Edit Online](#)

修改資料宣告時，`const` 關鍵字會指定物件或變數無法修改。

## 語法

```
const declaration ;
member-function const ;
```

## const 值

`const` 關鍵字會指定變數的值為常數，並告知編譯器防止程式設計人員修改它。

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

在 C++ 中，您可以使用 `const` 關鍵字，而不是`#define`預處理器指示詞來定義常數值。定義的值 `const` 會受到類型檢查的限制，而且可以用來取代常數運算式。在 C++ 中，您可以使用變數來指定陣列的大小，如下所示  
`const` :

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

在 C 中，常數值預設為外部連結，因此只能出現在原始程式檔中。在 C++ 中，常數值預設為內部連結，所以可以出現在標頭檔中。

`const` 關鍵字也可以用在指標宣告中。

```
// constant_values3.cpp
int main() {
    char *mybuf = 0, *yourbuf;
    char *const aptr = mybuf;
    *aptr = 'a'; // OK
    aptr = yourbuf; // C3892
}
```

宣告為之變數的指標 `const`，只能指派給也宣告為的指標 `const`。

```
// constant_values4.cpp
#include <stdio.h>
int main() {
    const char *mybuf = "test";
    char *yourbuf = "test2";
    printf_s("%s\n", mybuf);

    const char *bptr = mybuf; // Pointer to constant data
    printf_s("%s\n", bptr);

    // *bptr = 'a'; // Error
}
```

您可以將常數資料指標當做函式參數使用，以防止函式修改透過指標傳遞的參數。

對於宣告為的物件 `const`，您只能呼叫常數成員函式。這樣可以確保該常數物件永不會遭到修改。

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

您可以呼叫非常數物件的常數或非常數成員函式。您也可以使用關鍵字多載成員函式 `const`；這可針對常數和非常數物件呼叫不同版本的函式。

您無法使用關鍵字宣告函數或析構函式 `const`。

## const 成員函式

使用關鍵字宣告成員函式 `const` 會指定函式為「唯讀」函式，而不會修改呼叫它的物件。常數成員函式無法修改任何非靜態資料成員，或呼叫不是常數的任何成員函式。若要宣告常數成員函式，請將關鍵字放在 `const` 引數清單的右括弧後面。宣告 `const` 和定義中都需要關鍵字。

```
// 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 差異

當您在 C 原始程式碼檔案中將變數宣告為時 `const`，您可以執行下列動作：

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

然後，您可以在其他模組中使用此變數，例如：

```
extern const int i;
```

但是，若要在 C++ 中取得相同的行為，您必須將 `const` 變數宣告為：

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

如果您想要在 C `extern` ++ 原始程式碼檔案中宣告變數，以用於 C 原始程式碼檔，請使用：

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

以避免 C++ 編譯器改變名稱。

## 備註

當您遵循成員函式的參數清單時，`const` 關鍵字會指定函式不會修改叫用它的物件。

如需的詳細資訊 `const`，請參閱下列主題：

- [const 和 volatile 指標](#)
- [類型限定詞 \(C 語言參考\)](#)
- [volatile](#)
- [#define](#)

## 另請參閱

[關鍵字](#)

# constexpr (C++)

2020/11/2 • [Edit Online](#)

關鍵字 `constexpr` 是在 C++ 11 中引進，並在 C++ 14 中改善。這表示<sup>\*</sup> constant 運算式\*。就像 `const`，它可以套用至變數：當任何程式碼嘗試 `mod y` 值時，就會引發編譯器錯誤 if。不同 `const` `constexpr` 干，也可以套用至函式和類別 constructors。`constexpr` 表示值或傳回值為 constant，而且可能的話，會在編譯時期計算。

`constexpr` 整數值可以在需要整數的任何地方使用 `const`，例如在樣板引數和陣列宣告中。而且，在編譯時期而不是執行時間計算值時，它可協助您的程式執行速度更快且使用較少的記憶體。

為了限制編譯時期 ant 計算的複雜度 `const`，以及其對編譯時間的潛在影響，C++ 14 標準要求 ant 運算式中的類型 `const` 必須是 [常數值型別](#)。

## 語法

```
constexpr ***常數值型別* ident if ier = * constant 運算式* ;
constexpr ***常數值型別* ident if ier { * constant-expression* } ;
constexpr ***常數值型別* ident if ier ** (** 參數**) ** ;
constexpr ***ctor* ** (** 參數**) ** ;
```

## 參數

*Params*

一個或多個參數，每個參數都必須是常數值型別，而且本身必須是 constant 運算式。

## 傳回值

`constexpr` 變數或函數必須傳回常值類型。

## constexpr 變數

if 和變數之間的主要 difference `const` `constexpr` 是 `const` 可以延後到執行時間的變數初始化。`constexpr` 變數必須在編譯時期初始化。所有 `constexpr` 變數都是 `const`。

- `constexpr` 當變數具有常值型別且已初始化時，可使用宣告。如果是 ructor 中的每個 for med-v 初始化 `const`，則 constructor 必須宣告為 `constexpr`。
- `constexpr` 當符合這兩個條件時，可以宣告參考：參考的物件是由 constant 運算式初始化，而在初始化期間叫用的任何隱含轉換也都是 constant 運算式。
- 變數或函式的所有宣告都 `constexpr` 必須有 `constexpr` 規格 if ier。

```
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函數

函式 `constexpr` 是一種函式，其傳回值會在使用程式碼需要時于編譯時期可計算。使用程式碼需要在編譯時傳回值來初始化 `constexpr` 變數，或是提供非類型樣板引數。當其引數為 `constexpr` 值時，函式會 `constexpr` 產生編譯時期 `constant`。使用非 `constexpr` 引數呼叫，或在編譯時期不需要其值時，它會在執行時間產生值，就像一般函數一樣。(這個雙重行為可讓您不需要撰寫 `constexpr` 相同函式 `constexpr` 的和非版本。)

`constexpr` 函數或 `const ructor` 是隱含的 `inline`。

下列規則適用於 `constexpr` 函數：

- `constexpr` 函數必須接受並只傳回常數值型別。
- `constexpr` 函數可以是遞迴的。
- 它不能是虛擬的。當封入 `const` `constexpr` 類別具有任何虛擬基類時，無法定義 `ructor`。
- 主體可以定義為 `= default` 或 `= delete`。
- 主體可以不包含任何 `goto` 語句或 `try` 區塊。
- 非範本的明確特製化 `constexpr` 可以宣告為 `constexpr`：
- 範本的明確特製化 `constexpr` 也不一定要 `constexpr`：

下列規則適用於 `constexpr` Visual Studio 2017 和更新版本中的函數：

- 它可能包含 `if` 和 `switch` 語句，以及所有迴圈語句，包括 `for`、以範圍為基礎的 `for`、`while` 和\*\* `while` 等\*\*。
- 它可能包含區域變數宣告，但變數必須初始化。它必須是常數值型別，而且不能是 `static` 或執行緒區域。本機宣告的變數不一定要是 `const`，而且可能會改變。
- `constexpr` 非成員函 `static` 式不需要隱含 `const`。

```
constexpr float exp(float x, int n)
{
    return n == 0 ? 1 :
        n % 2 == 0 ? exp(x * x, n / 2) :
        exp(x * x, (n - 1) / 2) * x;
}
```

#### TIP

在 Visual Studio 偵錯工具中，當您 在偵錯工具中偵測到未優化的偵錯工具時，您可以藉 `constexpr` 由將中斷點放在編譯時期，判斷是否正在評估函式。如果叫用中斷點，便已在執行階段呼叫此函式。否則便是在編譯時期呼叫函式。

## Extern `constexpr`

`/Zc: externConstexpr` 編譯器選項會導致編譯器將外部連結套用至使用\*\*`extern constexpr` \*\*宣告的變數。在舊版 Visual Studio (預設值) 或 `/zc: externConstexpr-` 為 specified 時，if Visual Studio 會將內部連結套用至 `constexpr` 變數(即使 `extern` 使用關鍵字)。您可以從 Visual Studio 2017 更新15.6 開始取得 `/zc: externConstexpr` 選項，而且預設為關閉。`/Permissive-`選項不會啟用 `/zc: externConstexpr`。

## 範例

下列範例顯示 `constexpr` 變數、函式和使用者定義型別。在的最後一個語句中 `main()`，`constexpr` 成員函 `GetValue()` 式是執行時間呼叫，因為在編譯時期不需要知道此值。

```

// 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;
}

```

# 需求

Visual Studio 2015 或更新版本。

## 另請參閱

[宣告和定義](#)

`const`

# externC ++

2020/12/10 • [Edit Online](#)

`extern` 關鍵字可以套用至全域變數、函式或範本宣告。它會指定符號具有\* `extern al` 連結\*。如需連結的背景資訊，以及為什麼不鼓勵使用全域變數，請參閱[轉譯單位和連結](#)。

根據內容而定，`extern` 關鍵字具有四個意義：

- 在非 `const` 全域變數宣告中，`extern` 指定變數或函式定義于另一個轉譯單位中。`extern` 除了定義變數的檔案以外，必須套用在所有檔案中。
- 在變數宣告中 `const`，它會指定變數具有 `extern al` 連結。`extern` 必須套用至所有檔案中的所有宣告。（全域 `const` 變數預設具有內部連結）。
- \*\* `extern "C"** 指定函式定義于其他地方，並使用 C 語言呼叫慣例。extern "C" 修飾詞也可以套用至區塊中的多個函式宣告。`
- 在範本宣告中，`extern` 指定範本已在其他位置具現化。`extern` 告訴編譯器它可以重複使用另一個具現化，而不是在目前的位置建立一個新的實例。如需有關此使用的詳細資訊 `extern`，請參閱[明確具現化](#)。

## extern 非 globals 的連結 const

當連結器在 `extern` 全域變數宣告之前看到時，它會在另一個轉譯單位中尋找定義。全域範圍中非 `const` 變數的宣告預設為 `extern al`。僅適用於 `extern` 未提供定義的宣告。

```
//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)
```

## externglobals 的連結 const

`const` 全域變數預設具有內部連結。如果您想要讓變數具有 `extern al` 連結，請將 `extern` 關鍵字套用至定義，並套用至其他檔案中的所有其他宣告：

```
//fileA.cpp
extern const int i = 42; // extern const definition

//fileB.cpp
extern const int i; // declaration only. same as i in FileA
```

## externconstexpr 連結

在 Visual Studio 2017 15.3 版和更早版本中，編譯器一律會提供 `constexpr` 變數內部連結，即使變數已標記也一

樣 `extern`。在 Visual Studio 2017 15.5 版和更新版本中，[/zc: extern Constexpr](#) 編譯器參數會啟用符合標準的正確行為。最後，選項會變成預設值。此 [/permissive-](#) 選項不會啟用 `/Zc: extern Constexpr`。

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

如果標頭檔包含宣告的變數 `extern constexpr`，則必須將它標記 `__declspec(selectany)` 為正確地結合其重複宣告：

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

## extern"C" 和 extern "c + +" 函式宣告

在 c + + 中，搭配字串使用時，`extern` 會指定將另一種語言的連結慣例用於宣告子。只有當 c 函式和資料先前宣告為具有 C 連結時，才能加以存取。不過，您必須在另行編譯的轉譯單位中定義它們。

Microsoft c + + 支援字串常值欄位中的字串 "C" 和 "c + +"。所有標準 include 檔都使用\*\* `extern "c"**` 語法，以允許在 c + + 程式中使用執行時間程式庫函數。

## 範例

下列範例顯示如何宣告具有 C 連結的名稱：

```
// 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;
```

如果函式有一個以上的連結規格，則必須同意。將函式宣告為同時具有 C 和 c + + 連結是錯誤的。此外，如果程式中的一個函式發生兩次宣告（一個使用連結規格，另一個沒有），則使用連結規格的宣告必須是第一個。第一個宣告會針對任何已經有連結規格的其餘函式宣告指定連結。例如：

```
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.
```

## 另請參閱

[字](#)

[轉譯單位和連結](#)

[externC 中的儲存類別規範](#)

[C 中的識別碼行為](#)

[C 中的連結](#)

# 初始設定式

2020/11/2 • [Edit Online](#)

初始設定式指定變數的初始值。您可以初始化下列內容中的變數：

- 變數的定義：

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

- 函式的其中一個參數：

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

- 函式的傳回值：

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

初始設定式可能會有下列格式：

- 以括號刮住的運算式 (或逗號分隔的運算式清單)：

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

- 運算式後面接著等號：

```
string s = "hello";
```

- 以大括號括住的初始設定式清單。該清單可能是空的，也可能包含一組清單 (如下列範例所示)：

```
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({});
```

## 初始化的類型

初始化分為數種類型，可能會在程式執行的不同時期發生。不同類型的初始化不會互斥，例如清單初始化可以觸

發值初始化，而在其他情況下，則可觸發彙總初始化。

## 零初始化

零初始化是將變數設為隱含轉換為類型的零值：

- 數值變數初始化為 0 (或 0.0、0.000000000 等)。
- Char 變數會初始化為 `'\0'`。
- 指標會初始化為 `nullptr`。
- 陣列、POD 類別、結構和等位，其成員會初始化為零值。

零初始化在不同時期執行：

- 在程式啟動時，對象是具有靜態期間的所有具名變數。之後可以再次初始化這些變數。
- 在初始化值期間，對象是純量類型和使用空括號初始化的 POD 類別類型。
- 只初始化其成員子集的陣列。

以下舉例說明零初始化：

```
struct my_struct{
    int i;
    char c;
};

int i0;           // zero-initialized to 0
int main() {
    static float f1; // zero-initialized to 0.00000000
    double d{};     // zero-initialized to 0.0000000000000000
    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'
}
```

## 預設初始化

類別、結構和等位的預設初始化是使用預設建構函式的初始化。預設的函式可以使用沒有初始化運算式或使用關鍵字來呼叫 `new`：

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

如果類別、結構或等位沒有預設建構函式，則編譯器會發出錯誤。

定義純量變數時若未使用初始化運算式，則預設會初始化。這些變數都具有不定值。

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

定義陣列時若未使用初始化運算式，則預設會初始化。預設初始化陣列時，預設會初始化其成員，且其成員具有下列不定值 (如下列範例所示)：

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

如果陣列成員沒有預設建構函式，則編譯器會發出錯誤。

#### 常數變數的預設初始化

宣告常數變數時必須搭配使用初始設定式。如果常數變數是純量類型，則會引發編譯器錯誤，如果是具有預設建構函式的類別類型，則會引發警告：

```
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  
}
```

#### 靜態變數的預設初始化

未使用初始設定式宣告的靜態變數會初始化為 0 (隱含地轉換為類型)。

```
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'}  
}
```

如需全域靜態物件初始化的詳細資訊，請參閱[main 函數和命令列引數](#)。

#### 值初始化

在下列情況下，會發生值初始化：

- 使用空大括號初始設定初始化具名的值。
- 使用空括號或大括號初始化匿名暫存物件。
- 使用 `new` 關鍵字加上空括弧或大括弧初始化物件

值初始化會執行下列作業：

- 對於至少有一個公用建構函式的類別，會呼叫預設建構函式。
- 對於沒有宣告建構函式的非等位類別，會將該物件初始化為零並呼叫預設建構函式。
- 對於陣列，會將每個元素初始化為值。
- 在所有其他情況下，會將變數初始化為零。

```

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 0.0000000000000000
}

```

## 複製初始化

複製初始化是使用不同的物件初始化同一個物件。在下列情況下，會發生這類初始化：

- 使用等號初始化變數
- 將引數傳遞至函式
- 從函式傳回物件
- 擲回或攔截例外狀況
- 使用等號初始化非靜態資料成員
- 複製初始化會在彙總初始化期間初始化類別、結構和等位成員。如需範例，請參閱[匯總初始化](#)。

下列程式碼示範數個複製初始化範例：

```

#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();
    }
}

```

複製初始化無法叫用明確建構函式。

```
vector<int> v = 10; // the constructor is explicit; compiler error C2440: cannot convert from 'int' to  
'std::vector<int,std::allocator<_Ty>>'  
regex r = "a.*b"; // the constructor is explicit; same error  
shared_ptr<int> sp = new int(1729); // the constructor is explicit; same error
```

在某些情況下，若刪除或無法存取類別的複製建構函式，複製初始化會引發編譯器錯誤。

## 直接初始化

直接初始化是使用 (非空白) 大括號或括號的初始化。直接初始化可以叫用明確建構函式，這點與複製初始化不同。在下列情況下，會發生這類初始化：

- 使用非空白大括號或括號初始化變數。
- 使用 `new` 關鍵字加上非空白大括弧或括弧初始化變數
- 使用初始化變數\*\* `static_cast` \*\*
- 在建構函式中，會使用初始設定式清單初始化基底類別和非靜態成員。
- 在 Lambda 運算式之所擷取變數的複本中。

下列程式碼示範一些直接初始化範例：

```
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();  
}
```

## 清單初始化

使用以大括號括住的初始設定式清單初始化變數時，會發生清單初始化。在下列情況下，可以使用以大括號括住的初始設定式清單：

- 初始化變數
- 使用關鍵字初始化類別 `new`
- 從函式傳回物件
- 將引數傳遞至函式
- 直接初始化中的其中一個引數

- 在非靜態資料成員初始設定式中
- 在建構函式初始設定式清單中

下列程式碼示範一些清單初始化範例：

```

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' } };
}

```

## 匯總初始化

彙總初始化是陣列或類別類型 (通常是結構或等位) 的清單初始化表單，這些陣列或類別類型具有：

- 非 private 成員或 protected 成員
- 沒有使用者提供的建構函式 (明確預設或已刪除的建構函式除外)
- 沒有基底類別
- 沒有虛擬成員函式

### NOTE

在 Visual Studio 2015 和更早版本中，匯總不允許非靜態成員有大括弧或相等的初始化運算式。這項限制已在 C++ 14 標準中移除，並在 Visual Studio 2017 中執行。

彙總初始設定式包含以大括號括住的初始化清單 (包含或不含等號) (如下列範例所示)：

```

#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;
}

```

您應該會看見下列輸出：

```

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

```

#### IMPORTANT

在匯總初始化期間宣告但未明確初始化的陣列成員，會以零初始化，如上所示 `myArr3`。

#### 初始化等位和結構

如果等位沒有建構函式，則可以使用單一值（或使用等位的另一個執行個體）將它初始化。該值用於初始化第一個非靜態欄位。此初始化與結構初始化不同，後者會使用初始化設定式中的第一個值初始化第一個欄位、第二個值初始化第二個欄位，依此類推。比較下列範例中的等位初始化和結構初始化：

```

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'
}

```

### 初始化包含彙總的彙總

彙總類型可以包含其他彙總類型 (例如陣列、結構陣列等)。這些類型均使用巢狀大括號集進行初始化，例如：

```

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' } };
}

```

### 參考初始化

若要初始化參考類型的變數，必須使用衍生該參考類型之類型的物件，或者其類型可轉換成衍生該參考類型之類型的物件。例如：

```

// 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
}

```

使用暫存物件初始化參考的唯一方式是初始化常數暫存物件。初始化之後，參考類型變數一律會指向相同物件；不能將此變數修改為指向另一個物件。

雖然語法可能相同，但初始化參考類型變數和指派參考類型變數，在語意上是不同的。在上述範例中，變更 `iVar`

和 `lvar` 的指派就初始化而言看似相似，但兩者的作用不同。初始化會指定參考類型變數所指向的物件；指派則會透過參考指派被參考的物件。

由於將參考類型的引數傳遞至函式以及從函式傳回參考類型的值都是初始化，因此可以正確地初始化參考所傳回的函式型式引數。

只有在下列情況下，可以不使用初始設定式宣告參考類型變數：

- 函式宣告 (原型)。例如：

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

- 函式傳回類型宣告。例如：

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

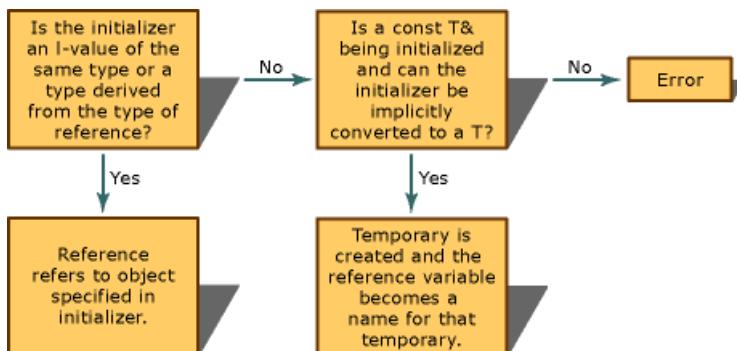
- 宣告函式類型類別成員。例如：

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

- 明確指定為之變數的宣告 `extern`。例如：

```
extern int& iVal;
```

初始化參考類型變數時，編譯器會使用下圖所示的決策圖，在建立物件參考或建立參考指向之暫存物件之間做選擇。



初始化參考型別的決策圖表

類型的參考 `volatile` (宣告為 `volatile typename & identifier`) 可以使用 `volatile` 相同類型的物件或尚未宣告為的物件來初始化 `volatile`。不過，它們無法使用 `const` 該類型的物件進行初始化。同樣地，類型的參考 `const` (宣告為 `const typename & identifier`) 可以使用 `const` 相同類型的物件 (或是任何轉換成該類型的專案，或是未宣告為的物件) 來初始化 `const`。不過，它們無法使用 `volatile` 該類型的物件進行初始化。

不是以或關鍵字限定的參考 `const` 只能 `volatile` 使用宣告為或的物件來初始化 `const volatile`。

### 外部變數的初始化

自動、靜態和外部變數的宣告可以包含初始化運算式。不過，只有當變數未宣告為時，外部變數的宣告才會包含初始化運算式 `extern`。

# 別名和 `typedef` (C++)

2020/11/2 • [Edit Online](#)

您可以使用別名宣告來宣告名稱，以當做先前宣告之類型的同義字使用。（這種機制也稱為「非正式」類型別名）。您也可以使用此機制來建立別名範本，這對自訂配置器特別有用。

## 語法

```
using identifier = type;
```

## 備註

### 標識

別名的名稱。

### *type*

您要建立別名的類型識別項。

別名不會引入新的類型，而且無法變更現有類型名稱的意義。

別名的最簡單形式相當於 `typedef` C++03 中的機制：

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

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

這兩個都啟用 "counter" 類型變數的建立。更有用的是像 `std::ios_base::fmtflags` 的這個類類型名：

```
// 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);
```

別名也適用於函式指標，但比對等的 `typedef` 更容易閱讀：

```
// 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;
```

此機制的限制 `typedef` 是它不會與範本搭配使用。不過, C++11 的類類型名語法啟用別名樣板的建立：

```
template<typename T> using ptr = T*;  
  
// the name 'ptr<T>' is now an alias for pointer to T  
ptr<int> ptr_int;
```

## 範例

下列範例示範如何搭配使用別名樣板與自訂配置器 (在這個案例中為整數向量類型)。您可以用任何類型替代 `int` 來建立方便的別名，以隱藏主要功能程式碼中的複雜參數清單。在您的程式碼中使用自訂配置器，可以改善可讀性和減少引入錯字所導致 Bug 的風險。

```
#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;  
}
```

## TypeDefs

宣告 `typedef` 會引進一個名稱，在其範圍內，會變成宣告的類型宣告部分所指定之類型的同義字。

您可以使用 `typedef` 宣告，針對語言已經定義的類型或您已經宣告的類型建構較短或更有意義的名稱。Typedef 名稱可讓您封裝可能變更的實作詳細資料。

相較于 `class`、`struct`、和宣告，宣告不 `union`、`enum`、`typedef` 會引進新的類型，而是引進現有類型的新名稱。

使用宣告的名稱會 `typedef` 佔用與其他識別碼相同的命名空間(除了語句標籤以外)。因此，除非是在類別類型宣告中，否則這類名稱不能使用與先前宣告的名稱相同的識別項。請考慮下列範例：

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

與其他識別碼相關的名稱隱藏規則也會控制使用宣告之名稱的可見度 `typedef`。因此，下列範例在 C++ 中是合法的：

```
// typedef_names2.cpp
typedef unsigned long UL;    // Declare a typedef name, UL
int main()
{
    unsigned int UL;    // Redeclaration hides typedef name
}

// typedef UL back in scope
```

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

int main()
{
}

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

將相同名稱的區域範圍識別項宣告為 `typedef` 時，或者宣告相同範圍或內部範圍之結構或等位的成員時，必須指定類型指定名稱。例如：

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

若要重複使用 `FlagType` 名稱做為識別項、結構成員或等位成員的名稱，必須提供類型：

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

只有下列做法是不夠的：

```
const FlagType;      // Incomplete specification
```

因為 `FlagType` 會被當做類型的一部分，而非將重新宣告的識別項。這個宣告會被視為不合法的宣告，如同

```
int; // Illegal declaration
```

您可以使用 `typedef` 宣告任何類型，包括指標、函式和陣列類型。您可以在定義結構或等位類型之前宣告結構或等位類型指標的 `typedef` 名稱，只要定義的可見度和宣告相同即可。

### 範例

宣告的其中一種用法 `typedef` 是讓宣告更為一致和精簡。例如：

```
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` 在相同的宣告中指定基本和衍生類型，可以用逗號分隔宣告子。例如：

```
typedef char CHAR, *PSTR;
```

下列範例提供類型 `DRAWF` 給未傳回任何值的函式，並且接受兩個 `int` 引數：

```
typedef void DRAWF( int, int );
```

在上述 `typedef` 語句之後，宣告

```
DRAWF box;
```

相當於下列宣告：

```
void box( int, int );
```

`typedef` 通常會與結合 `struct` 以宣告和命名使用者定義的類型：

```
// typedefSpecifier2.cpp
#include <stdio.h>

typedef struct mystructtag
{
    int i;
    double f;
} mystruct;

int main()
{
    mystruct ms;
    ms.i = 10;
    ms.f = 0.99;
    printf_s("%d %f\n", ms.i, ms.f);
}
```

10 0.990000

### typedef 的重新宣告

宣告 `typedef` 可以用來重新宣告相同的名稱，以參考相同的類型。例如：

```
// FILE1.H
typedef char CHAR;

// FILE2.H
typedef char CHAR;

// PROG.CPP
#include "file1.h"
#include "file2.h" // OK
```

程式進程。CPP包含兩個標頭檔，兩者都包含 `typedef` 名稱的宣告 `CHAR`。只要兩個宣告都參考相同的類型，就可以接受此類重新宣告。

`typedef` 無法重新定義先前宣告為不同類型的名稱。因此，如果FILE2.H包含

```
// FILE2.H
typedef int CHAR; // Error
```

由於嘗試將名稱 `CHAR` 重新定義為參考不同類型，因此編譯器會發出錯誤。此項重新定義可延伸到如下的建構：

```
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.
    structbyteregs h;
} REGS;
```

### c++ 與 C 中的 typedef

由於在宣告 `typedef` 中宣告未命名結構的 ANSI C 實務，因此支援使用具有類別類型的規範 `typedef`。例如，許多 C 程式設計人員採用下列作法：

```
// 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;
```

這類宣告的優點是可以執行類似以下的宣告：

```
POINT ptOrigin;
```

而非：

```
struct point_t ptOrigin;
```

在 C++ 中，`typedef` 名稱和實數類型（使用 `class`、`struct`、和關鍵字宣告）之間的差異 `union`、`enum` 比較相異。雖然在語句中宣告無編號結構的 C 作法 `typedef` 仍然可以運作，但它不會提供任何標記的優點，如同在 C 中所做的一樣。

```
// typedef_with_class_types2.cpp
// compile with: /c /W1
typedef struct {
    int POINT();
    unsigned x;
    unsigned y;
} POINT;
```

上述範例會 `POINT` 使用未命名的類別語法來宣告名為的類別 `typedef`。`POINT` 是類別名稱；不過，以此方式產生的名稱會受到下列限制：

- 名稱（同義字）不能出現在 `class`、`struct` 或前置詞之後 `union`。
- 不可在類別宣告中使用此名稱做為建構函式或解構函式名稱。

總而言之，此語法不提供任何繼承、建構或解構機制。

# using 宣告

2020/11/2 • [Edit Online](#)

`using` 宣告會將名稱引入宣告式區域，其中會出現 `using` 宣告。

## 語法

```
using [typename] nested-name-specifier unqualified-id ;  
using declarator-list ;
```

### 參數

嵌套名稱規範命名空間、類別或列舉名稱和範圍解析運算子的序列，`(::)`，由範圍解析運算子終止。您可以使用單一範圍解析運算子來引入全域命名空間中的名稱。關鍵字 `typename` 是選擇性的，可用來解析從基類引入類別樣板的相依名稱。

未限定識別碼不合格的識別碼運算式，可能是識別碼、多載的運算子名稱、使用者定義的常值運算子或轉換函式名稱、類別的函式名稱，或範本名稱和引數清單。

宣告子清單以逗號分隔的 `[ typename ]` 嵌套名稱規範不合格識別碼宣告子清單，後面接著省略號。

## 備註

`Using` 宣告導入了不合格的名稱，做為在其他地方宣告之實體的同義字。它允許使用特定命名空間中的單一名稱，而不需要在其出現的宣告區域中明確限定。這與 [使用](#) 指示詞相較之下，其允許在命名空間中使用 [所有](#) 名稱，而不需限定。`using` 關鍵字也可用於[類型別名](#)。

## 範例： `using` 類別欄位中的宣告

`Using` 宣告可用於類別定義中。

```

// using_declaration1.cpp
#include <stdio.h>
class B {
public:
    void f(char) {
        printf_s("In B::f()\n");
    }

    void g(char) {
        printf_s("In B::g()\n");
    }
};

class D : B {
public:
    using B::f;      // B::f(char) is now visible as D::f(char)
    using B::g;      // B::g(char) is now visible as D::g(char)
    void f(int) {
        printf_s("In D::f()\n");
        f('c');      // Invokes B::f(char) instead of recursing
    }

    void g(int) {
        printf_s("In D::g()\n");
        g('c');      // Invokes B::g(char) instead of recursing
    }
};

int main() {
    D myD;
    myD.f(1);
    myD.g('a');
}

```

```

In D::f()
In B::f()
In B::g()

```

## 範例:宣告 using 成員的宣告

當用來宣告成員時, using 宣告必須參考基類的成員。

```
// using_declaration2.cpp
#include <stdio.h>

class B {
public:
    void f(char) {
        printf_s("In B::f()\n");
    }

    void g(char) {
        printf_s("In B::g()\n");
    }
};

class C {
public:
    int g();
};

class D2 : public B {
public:
    using B::f; // ok: B is a base of D2
    // using C::g; // error: C isn't a base of D2
};

int main() {
    D2 MyD2;
    MyD2.f('a');
}
```

```
In B::f()
```

## 範例： `using` 具有明確限定性的宣告

使用 `using` 宣告宣告的成員可以使用明確限定性來參考。`::` 前置詞是指全域命名空間。

```

// using_declaration3.cpp
#include <stdio.h>

void f() {
    printf_s("In f\n");
}

namespace A {
    void g() {
        printf_s("In A::g\n");
    }
}

namespace X {
    using ::f;    // global f is also visible as X::f
    using A::g;   // A's g is now visible as X::g
}

void h() {
    printf_s("In h\n");
    X::f();      // calls ::f
    X::g();      // calls A::g
}

int main() {
    h();
}

```

```

In h
In f
In A::g

```

## 範例: 宣告 `using` 同義字和別名

當使用宣告時，宣告所建立的同義字只會參考在 `using` 宣告的點有效的定義。使用宣告之後新增至命名空間的定義，不是有效的同義字。

宣告所定義的名稱 `using` 是其原始名稱的別名。它不會影響原始宣告的類型、連結或其他屬性。

```

// post_declarator_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);
}

```

## 範例: 區域 `using` 宣告和宣告

關於命名空間中的函式，如果宣告式區域中有提供單一名稱的一組區域宣告和使用宣告，則它們必須參考相同的實體，或它們都必須參考函數。

```
// 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
}
```

在上述範例中，`using B::i` 語句會導致 `int i` 在函數中宣告第二個 `g()`。`using B::f` 語句不會與函數衝突，`f(char)` 因為引進的函式名稱 `B::f` 具有不同的參數類型。

## 範例: 區域函數 `using` 單明和宣告

區域函式宣告的名稱和類型不能與使用宣告所引進的函式相同。例如：

```
// 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)
}
```

## 範例: 宣告 `using` 和繼承

就繼承而言，當 `using` 宣告將名稱從基類引入衍生類別範圍時，衍生類別中的成員函式會覆寫基類中具有相同名稱和引數類型的虛擬成員函數。

```

// using_declaration_inheritance1.cpp
#include <stdio.h>
struct B {
    virtual void f(int) {
        printf_s("In B::f(int)\n");
    }

    virtual void f(char) {
        printf_s("In B::f(char)\n");
    }

    void g(int) {
        printf_s("In B::g\n");
    }

    void h(int);
};

struct D : B {
    using B::f;
    void f(int) { // ok: D::f(int) overrides B::f(int)
        printf_s("In D::f(int)\n");
    }

    using B::g;
    void g(char) { // ok: there is no B::g(char)
        printf_s("In D::g(char)\n");
    }

    using B::h;
    void h(int) {} // Note: D::h(int) hides non-virtual B::h(int)
};

void f(D* pd) {
    pd->f(1); // calls D::f(int)
    pd->f('a'); // calls B::f(char)
    pd->g(1); // calls B::g(int)
    pd->g('a'); // calls D::g(char)
}

int main() {
    D * myd = new D();
    f(myd);
}

```

```

In D::f(int)
In B::f(char)
In B::g
In D::g(char)

```

## 範例：宣告 `using` 存取範圍

使用宣告中提及之名稱的所有實例都必須可以存取。尤其是，如果衍生類別使用 `using` 告知來存取基類的成員，則必須可存取成員名稱。如果名稱是多載成員函式的名稱，則所有名為的函式都必須是可存取的。

如需成員存取範圍的詳細資訊，請參閱 [成員存取控制](#)。

```
// 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
};
```

## 另請參閱

[命名空間](#)

[關鍵字](#)

# volatile (C++)

2020/11/2 • [Edit Online](#)

類型限定詞，可以用來宣告程式中的物件可以由硬體修改。

## 語法

```
volatile declarator ;
```

## 備註

您可以使用/volatile編譯器參數來修改編譯器如何解讀此關鍵字。

Visual Studio 會根據 volatile 目標架構，以不同的方式來解讀關鍵字。針對 ARM，如果未指定 /volatile 編譯器選項，則編譯器會執行，如同已指定 /volatile: iso。針對 ARM 以外的架構，如果未指定 /volatile 編譯器選項，編譯器會執行，如同已指定 /volatile: ms 一樣。因此，對於 ARM 以外的架構，我們強烈建議您指定 /volatile: iso，並在處理跨執行緒共用的記憶體時，使用明確的同步處理原始物件和編譯器內建函式。

您可以使用 volatile 限定詞來提供非同步處理常式所使用之記憶體位置的存取權，例如中斷處理常式。

當 volatile 用於也具有 `_restrict` 關鍵字的變數時，volatile 會優先使用。

如果 struct 成員已標記為 volatile，則 volatile 會傳播到整個結構。如果結構沒有可使用一個指令複製到目前架構上的長度，volatile 可能會在該結構上完全遺失。

volatile 如果下列其中一個條件成立，關鍵字可能不會影響欄位：

- volatile 欄位長度超過可透過單一指令複製到目前架構的大小上限。
- 最外層包含的長度 struct (如果它是可能嵌套的成員 struct) 超過可使用一個指令複製到目前架構上的大小上限。

雖然處理器不會重新排列無法快取的記憶體存取，但是無法快取的變數必須標示為，volatile 以確保編譯器不會重新排序記憶體存取。

宣告為的物件 volatile 不會在特定優化中使用，因為它們的值可以隨時變更。即使先前指令要求相同物件的值，再次被要求時，系統一定會讀取暫時性物件目前的值。此外，物件的值會在指派時立即被寫入。

## 符合 ISO 標準

如果您熟悉 c # volatile 關鍵字，或熟悉 volatile 舊版 Microsoft C++ 編譯器 (MSVC) 中的行為，請注意 C++ 11 ISO Standard volatile 關鍵字是不同的，而且當指定 /volatile: ISO 編譯器選項時，會在 MSVC 中受到支援。(對於 ARM 系統來說，預設為指定)。volatile C++ 11 ISO 標準程式碼中的關鍵字僅用於硬體存取；請勿將它用於執行緒間的通訊。對於執行緒間通訊，請使用 C++ 標準程式庫 中的機制(例如 std:: < T > 不可部分完成)。

## 符合 ISO 標準結尾

Microsoft 特定的

使用 /volatile: ms 編譯器選項時—根據預設，當 ARM 以外的架構為目標時，編譯器會產生額外的程式碼來維護對 volatile 物件之參考的順序，以及維護其他全域物件參考的順序。尤其是：

- 暫時性物件的寫入 (也稱為暫時性寫入) 有 Release 語義，也就是說，在指令序列中暫時性物件寫入之前的全域或靜態物件參考，會在已編譯二進位檔中的暫時性寫入之前發生。
- 暫時性物件的讀取 (也稱為暫時性讀取) 有 Acquire 語義，也就是說，在指令序列中揮發性記憶體讀取之後的全域或靜態物件參考，會在已編譯二進位檔中的暫時性讀取之後發生。

這可讓暫時性物件用於多執行緒應用程式的記憶體鎖定和記憶體釋放。

#### NOTE

當使用 /volatile: ms 編譯器選項時所提供的增強保證時，程式碼是不可移植的。

結束 Microsoft 專有

## 另請參閱

關鍵字

const

[const 和 volatile 指標](#)

# decltype (C++)

2020/11/2 • [Edit Online](#)

`decltype` 類型規範會產生指定運算式的類型。`decltype` 類型規範與 `auto` 關鍵字搭配，主要適用於撰寫範本程式庫的開發人員。使用 `auto` 和宣告樣板函式，`decltype` 其傳回型別取決於其樣板引數的類型。或者，使用和來宣告包裝對另一個函式之呼叫的樣板函式 `auto decltype`，然後傳回包裝函式的傳回型別。

## 語法

```
decltype( 運算式 )
```

### 參數

#### 表達

運算式。如需詳細資訊，請參閱 [運算式](#)。

## 傳回值

運算式參數的型別。

## 備註

`decltype` 類型規範在 Visual Studio 2010 或更新版本中受到支援，而且可以搭配原生或 managed 程式碼使用。Visual Studio 2015 和更新版本支援 `decltype(auto)` (C++14)。

編譯器會使用下列規則來判斷 運算式 參數的類型。

- 如果 運算式 參數是識別碼或 類別成員存取，`decltype(expression)` 就是以 運算式 命名的實體類型。如果沒有這類實體或 運算式 參數命名一組多載函式，編譯器會產生錯誤訊息。
- 如果 *expression* 參數是對函式或多載運算子函式的呼叫，`decltype(expression)` 就是函式的傳回型別。在多載運算子周圍的括號會被忽略。
- 如果 運算式 參數是 **右值**，`decltype(expression)` 就是 運算式的類型。如果 *expression* 參數是 **左值**，`decltype(expression)` 則是 運算式 類型的 **左值參考**。

下列程式碼範例示範類型規範的一些用法 `decltype`。首先，假設您撰寫了下列陳述式。

```
int var;
const int&& fx();
struct A { double x; }
const A* a = new A();
```

接下來，請檢查下表中四個語句所傳回的類型 `decltype`。

III	II	I
<code>decltype(fx());</code>	<code>const int&amp;&amp;</code>	的 <b>右值參考</b> <code>const int</code> 。
<code>decltype(var);</code>	<code>int</code>	變數 <code>var</code> 的類型。

<code>decltype(a-&gt;x);</code>	<code>double</code>	成員存取的類型。
<code>decltype((a-&gt;x));</code>	<code>const double&amp;</code>	括號內的陳述式會評估為運算式而不是成員存取。因為 <code>a</code> 是宣告為 <code>const</code> 指標，所以型別是的參考 <code>const double</code> 。

## Decltype 和 Auto

在 C++14 中，您可以使用 `decltype(auto)` with no 尾端傳回型別來宣告樣板函式，其傳回型別取決於其樣板引數的類型。

在 C++11 中，您可以使用 `decltype` 尾端傳回類型上的類型規範與 `auto` 關鍵字，來宣告其傳回類型取決於其樣板引數類型的樣板函式。例如，請考慮下列程式碼範例，其中樣板函式的傳回型別取決於樣板引數的類型。在程式碼範例中，未知的預留位置表示無法指定傳回型別。

```
template<typename T, typename U>
UNKNOWN func(T&& t, U&& u){ return t + u; };
```

類型規範的引入可 `decltype` 讓開發人員取得範本函式所傳回之運算式的型別。使用稍後所示的 替代函式宣告語法、`auto` 關鍵字和 `decltype` 類型規範來宣告 晚期指定 的傳回型別。晚期指定的傳回類型是在編譯宣告時決定，而不是在撰寫程式時決定。

下列原型說明替代函式宣告的語法。請注意，`const` 和 `volatile` 限定詞，以及 `throw` 例外狀況規格 是選擇性的。*Function\_body* 預留位置代表指定函式功能的複合陳述式。作為最佳編碼作法，語句中的 運算式 預留位置 `decltype` 應符合 `return function_body` 中的語句(如果有的話)所指定的運算式。

```
auto ***function_name* [ * opt opt ] const opt volatile opt -> decltype( 運算式 ) noexcept opt
{ function_body 的參數 function_body** }; *
```

下列程式碼範例中，`myFunc` 樣板函式的晚期指定傳回類型由 `t` 和 `u` 樣板引數的類型決定。作為最佳編碼作法，程式碼範例也會使用右值參考和函式 `forward` 範本，以支援 完美轉送。如需詳細資訊，請參閱[右值參考宣告子:&&](#)。

```
//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 和轉送函式 (C++11)

轉送函式會包裝對其他函式的呼叫。試想一個轉送其引數，或者包含這些引數的運算式結果到其他函式的函式樣板。此外，轉送函式會傳回呼叫其他函式的結果。在這個案例中，轉送函式的傳回類型應該會與包裝函式的傳回類型相同。

在此案例中，您無法在沒有類型規範的情況下寫入適當的類型運算式 `decltype`。`decltype` 類型規範會啟用泛型轉送函式，因為它不會遺失有關函數是否傳回參考型別的必要資訊。如需轉送函式的程式碼範例，請參閱先前的

`myFunc` 樣板函式範例。

## 範例

下列程式碼範例會宣告樣板函式 `Plus()` 的晚期指定傳回型別。函數會使用多載來 `Plus` 處理其兩個運算元 `operator+`。因此，加號運算子的解讀 ()，而函式的傳回型別取決於函式 `+` `Plus` 引數的類型。

```

// decltype_1.cpp
// compile with: cl /EHsc decltype_1.cpp

#include <iostream>
#include <string>
#include <utility>
#include <iomanip>

using namespace std;

template<typename T1, typename T2>
auto Plus(T1&& t1, T2&& t2) ->
    decltype(forward<T1>(t1) + forward<T2>(t2))
{
    return forward<T1>(t1) + forward<T2>(t2);
}

class X
{
    friend X operator+(const X& x1, const X& x2)
    {
        return X(x1.m_data + x2.m_data);
    }

public:
    X(int data) : m_data(data) {}
    int Dump() const { return m_data; }
private:
    int m_data;
};

int main()
{
    // Integer
    int i = 4;
    cout <<
        "Plus(i, 9) = " <<
        Plus(i, 9) << endl;

    // Floating point
    float dx = 4.0;
    float dy = 9.5;
    cout <<
        setprecision(3) <<
        "Plus(dx, dy) = " <<
        Plus(dx, dy) << endl;

    // String
    string hello = "Hello, ";
    string world = "world!";
    cout << Plus(hello, world) << endl;

    // Custom type
    X x1(20);
    X x2(22);
    X x3 = Plus(x1, x2);
    cout <<
        "x3.Dump() = " <<
        x3.Dump() << endl;
}

```

```

Plus(i, 9) = 13
Plus(dx, dy) = 13.5
Hello, world!
x3.Dump() = 42

```

**Visual Studio 2017 和更新版本：**編譯器會在宣告 `decltype` 範本時剖析引數，而非具現化。因此，如果在引數中找到非相依特製化 `decltype`，則不會延後到具現化時間，而且會立即處理，而且會在該時間診斷任何產生的錯誤。

下列範例顯示在宣告時引發的這類編譯器錯誤：

```
#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");
```

## 需求

Visual Studio 2010 或更新版本。

`decltype(auto)` 需要 Visual Studio 2015 或更新版本。

# C++ 中的屬性

2020/11/2 • [Edit Online](#)

C++ 標準會定義一組屬性，也可讓編譯器廠商定義自己的屬性（在廠商專屬的命名空間內），但編譯器只需要辨識標準中定義的屬性。

在某些情況下，標準屬性會與編譯器特定的 `declspec` 參數重迭。在 Visual C++ 中，您可以使用 `[[deprecated]]` 屬性，而不是使用 `declspec(deprecated)`，而且任何符合的編譯器都會辨識屬性。對於所有其他的 `declspec` 參數（例如 `dllimport` 和 `dllexport`），除了屬性對等，您還必須繼續使用 `declspec` 語法。屬性不會影響類型系統，而且不會變更程式的意義。編譯器會忽略無法辨識的屬性值。

**Visual Studio 2017 15.3 版和更新版本**（適用於 [/std: c++17](#)）：在屬性清單的範圍內，您可以使用單一 `introducer` 來指定所有名稱的命名空間 `using`：

```
void g() {
    [[using rpr: kernel, target(cpu,gpu)]] // equivalent to [[ rpr::kernel, rpr::target(cpu,gpu) ]]
    do task();
}
```

## C++ 標準屬性

在 C++11 中，屬性提供了一種標準化的方式，可將 C++ 結構（包括但不限於類別、函式、變數和區塊）標注為其他不一定是廠商特有的資訊。編譯器可以使用此資訊來產生參考用訊息，或在編譯屬性化程式碼時套用特殊邏輯。編譯器會忽略無法辨識的任何屬性，這表示您無法使用此語法來定義您自己的自訂屬性。屬性會以雙方括弧括住：

```
[[deprecated]]
void Foo(int);
```

屬性代表特定廠商擴充功能的標準化替代方式，例如 `#pragma` 指示詞、`__declspec()`（Visual C++），或 `__attribute__`（GNU）。不過，您仍然需要使用廠商專屬的結構來進行大部分的工作。標準目前指定了下列屬性，這些是符合的編譯器應該辨識的屬性：

- `[[noreturn]]` 指定函數永遠不會傳回；換句話說，它一律會擲回例外狀況。編譯器可以調整其實體的編譯規則 `[[noreturn]]`。
- `[[carries_dependency]]` 指定函式會傳播與執行緒同步處理有關的資料相依性排序。屬性可以套用至一個或多個參數，以指定傳入的引數會在函式主體中攜帶相依性。屬性可以套用至函式本身，以指定傳回值會從函式中承擔相依性。編譯器可以使用此資訊來產生更有效率的程式碼。
- `[[deprecated]]` **Visual Studio 2015 和更新版本**：指定函式不打算使用，而且可能不存在於程式庫介面的未來版本中。當用戶端程式代碼嘗試呼叫函式時，編譯器可以使用此函數來產生參考用訊息。可以套用至類別、`typedef` 名稱、變數、非靜態資料成員、函式、命名空間、列舉、枚舉器或範本特製化的宣告。
- `[[fallthrough]]` **Visual Studio 2017 和更新版本**：（適用於 [/std: c++17](#)），`[[fallthrough]]` 屬性可以在 `switch` 語句的內容中用來當做 `fallthrough` 行為所預期的編譯器（或讀取程式碼的任何人）的提示。Microsoft C++ 編譯器目前不會警告 `fallthrough` 行為，因此這個屬性不會影響編譯器行為。
- `[[nodiscard]]` **Visual Studio 2017 15.3 版和更新版本**：（適用於 [/std: c++17](#)）指定函式的傳回值不會被捨棄。引發警告 C4834，如下列範例所示：

```
[[nodiscard]]  
int foo(int i) { return i * i; }  
  
int main()  
{  
    foo(42); //warning C4834: discarding return value of function with 'nodiscard' attribute  
    return 0;  
}
```

- **[[maybe\_unused]] Visual Studio 2017 15.3 版和更新版本：**(可與[/std: c++17](#)搭配使用)指定不使用變數、函式、類別、typedef、非靜態資料成員、列舉或樣板特製化。未使用標示的實體時，編譯器不會發出警告  
[[maybe\_unused]]。不含屬性宣告的實體之後可以使用屬性重新宣告，反之亦然。實體會在其第一個標記為已分析的宣告之後，以及目前轉譯單位的其餘部分，被視為標示。

## Microsoft 特有的屬性

- **[[gsl::suppress(rules)]]**這個 Microsoft 特有的屬性是用來隱藏在程式碼中強制執行[方針支援程式庫 \(GSL\)](#)規則之檢查的警告。例如，請考慮下列程式碼片段：

```
int main()  
{  
    int arr[10]; // GSL warning C26494 will be fired  
    int* p = arr; // GSL warning C26485 will be fired  
    [[gsl::suppress(bounds.1)]] // This attribute suppresses Bounds rule #1  
    {  
        int* q = p + 1; // GSL warning C26481 suppressed  
        p = q--; // GSL warning C26481 suppressed  
    }  
}
```

此範例會引發下列警告：

- 26494(類型規則5:一律初始化物件)。
- 26485(界限規則3:沒有陣列到指標衰減)。
- 26481(界限規則1:不要使用指標算術。請改用 span)。

當您使用已安裝並啟用的 CppCoreCheck 程式碼分析工具來編譯此程式碼時，就會引發前兩個警告。但是第三個警告並不會因為屬性而引發。您可以藉由撰寫 [[gsl::抑制(界限)]] 來隱藏整個界限設定檔，而不包含特定的規則編號。C++ Core Guidelines 的設計可協助您撰寫更好且更安全的程式碼。[隱藏] 屬性可讓您在不想要的時候，輕鬆關閉警告。

# C++ 內建運算子、優先順序和關聯性

2020/11/2 • [Edit Online](#)

C++ 語言包含所有 C 運算子，並且新增了數個新的運算子。運算子指定要對一個或多個運算元執行的評估：

## 優先順序和關聯性

運算子 **優先順序**: 指定運算式中包含多個運算子的作業順序。運算子 **關聯性**: 指定在包含多個具有相同優先順序之運算子的運算式中，運算元會群組在其左邊或其右邊的一個。

## 替代拼寫

C++ 為某些運算子指定替代的拼寫。在 C 中，替代的拼寫是以宏的形式在 `<iso646.h>` 標頭中提供。在 C++ 中，這些替代專案是關鍵字，使用 `<iso646.h>` 或 C++ 對等專案 `<ciso646>` 已被取代。在 Microsoft C++ 中，`/permissive-` `/Za` 需要或編譯器選項才能啟用替代的拼寫。

## C++ 運算子優先順序和關聯性資料表

下表顯示 C++ 運算子的優先順序和順序關聯性（從最高到最低優先順序）。除非以括號明確強制其他關聯性，否則運算子的優先順序數字若相同，其優先順序即相等。

運算子	優先順序	關聯性
範圍解析	1	<code>::</code>
成員選取 (物件或指標)	2	<code>.</code> 或 <code>-&gt;</code>
陣列註標	3	<code>[]</code>
函式呼叫	4	<code>()</code>
後置遞增	5	<code>++</code>
後置遞減	6	<code>--</code>
類型名稱	7	<code>typeid</code>
常數類型轉換	8	<code>const_cast</code>
動態類型轉換	9	<code>dynamic_cast</code>
重新轉譯的類型轉換	10	<code>reinterpret_cast</code>
靜態類型轉換	11	<code>static_cast</code>

■■■■■	■■■	■■■■■
■3 ■■, ■■■		
物件或類型的大小	<code>sizeof</code>	
前置遞增	<code>++</code>	
前置遞減	<code>--</code>	
一補數	<code>~</code>	<code>compl</code>
邏輯 not	<code>!</code>	<code>not</code>
一元負運算	<code>-</code>	
一元加號	<code>+</code>	
傳址	<code>&amp;</code>	
間接	<code>*</code>	
建立物件	<code>new</code>	
終結物件	<code>delete</code>	
鑄造	<code>()</code>	
■4 ■■, ■■■		
成員指標 (物件或指標)	<code>.*</code> 或 <code>-&gt;*</code>	
■5 ■■, ■■■		
乘	<code>*</code>	
部門	<code>/</code>	
模組	<code>%</code>	
■6 ■■, ■■■		
加法	<code>+</code>	
減	<code>-</code>	
■7 ■■, ■■■		
左移	<code>&lt;&lt;</code>	

■■■■■	■■■	■■■■■
右移	>>	
■8 ■, ■■■		
小於	<	
大於	>	
小於或等於	<=	
大於或等於	>=	
■9 ■, ■■■		
等式	==	
不等	!=	not_eq
■10 ■■■		
位元 AND	&	bitand
■11 ■, ■■■		
位元排除 OR	^	xor
■12 ■, ■■■		
位元包含 OR		bitor
■13 ■, ■■■		
邏輯 AND	&&	and
■14 ■, ■■■		
邏輯 OR		or
■15 ■, ■■■		
條件式	? :	
■16 ■, ■■■		
指派	=	
乘法指派	*=	
除法指派	/=	

█████	████	████
模數指派	<code>%=</code>	
加法指派	<code>+=</code>	
減法指派	<code>-=</code>	
左移指派	<code>&lt;&lt;=</code>	
右移指派	<code>&gt;&gt;=</code>	
位元 AND 指派	<code>&amp;=</code>	<code>and_eq</code>
位元包含 OR 指派	<code> =</code>	<code>or_eq</code>
位元互斥 OR 指派	<code>^=</code>	<code>xor_eq</code>
■17■, ■■■		
擲回運算式	<code>throw</code>	
■18■, ■■■		
千	,	

## 另請參閱

[運算子多載](#)

# alignof 運算子

2020/11/2 • [Edit Online](#)

`alignof` 運算子會以位元組作為類型的值，傳回指定之類型的對齊方式 `size_t`。

## 語法

```
alignof( type )
```

## 備註

例如：

<code>alignof( char )</code>	1
<code>alignof( short )</code>	2
<code>alignof( int )</code>	4
<code>alignof( long long )</code>	8
<code>alignof( float )</code>	4
<code>alignof( double )</code>	8

`alignof` 值與 `sizeof` 適用于基本類型的值相同。不過，請考慮這個範例：

```
typedef struct { int a; double b; } S;
// alignof(S) == 8
```

在此情況下，此 `alignof` 值是結構中最大元素的對齊需求。

同樣地，針對

```
typedef __declspec(align(32)) struct { int a; } S;
```

`alignof(S)` 等於 `32`。

一種用途 `alignof` 是做為您自己的記憶體配置常式之一的參數。例如，假設有下列定義的結構 `S`，您可以呼叫名為 `aligned_malloc` 的記憶體配置常式，至特定對齊界限上配置記憶體。

```
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));
```

如需修改對齊的詳細資訊，請參閱：

- [pack](#)
- [對齊](#)
- [\\_\\_unaligned](#)
- [/Zp \(結構成員對齊\)](#)
- [結構對齊範例 \(x64 專用\)](#)

如需適用於 x86 和 x64 之程式碼中的對齊差異的詳細資訊，請參閱：

- [與 x86 編譯器衝突](#)

## Microsoft 專有

`alignof` 和 `__alignof` 是 Microsoft 編譯器中的同義字。Microsoft 專屬的操作員在成為 C++ 11 標準的一部分之前，`__alignof` 提供了這項功能。若要獲得最大的可攜性，您應該使用 `alignof` 運算子，而不是 Microsoft 特有的 `__alignof` 運算子。

為了與舊版相容，`_alignof` `__alignof` 除非指定了編譯器選項 [停用 [/Za \(語言擴充功能\)](#)]，否則會是同義字。

## 另請參閱

[具有一元運算子的運算式](#)

[關鍵字](#)

`__uuidof`

# 運算子

2020/11/2 • [Edit Online](#)

## Microsoft 特定的

擷取附加至運算式的 GUID。

## 語法

`__uuidof ( 運算式 )`

## 備註

運算式可以是類型名稱、指標、參考或該類型的陣列、在這些類型上特製化的樣板，或這些類型的變數。只要編譯器可以用它來尋找附加的 GUID，則引數有效。

這個內建函式的特殊案例是當 0 或 Null 做為引數提供時。在此情況下，`__uuidof` 會傳回由零組成的 GUID。

使用這個關鍵字擷取附加至下列項目的 GUID：

- 由擴充屬性所產生的物件 `uuid`。
- 使用屬性建立的程式庫區塊 `module`。

### NOTE

在 debug 組建中，`__uuidof` 一律會動態初始化物件（在執行時間）。在發行組建中，`__uuidof` 可以靜態方式（在編譯時期）初始化物件。

為了與舊版相容，`__uuidof` `__uuidof` 除非指定了編譯器選項 [停用 `/za` (語言擴充功能)]，否則會是同義字。

## 範例

下列程式碼（以 ole32.lib 編譯）會顯示以模組屬性所建立之程式庫區塊的 uuid：

```
// 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);
}
```

## 註解

如果程式庫名稱不再在範圍內，您可以使用 `__LIBID_` 而不是 `__uuidof`。例如：

```
StringFromCLSID(__LIBID_, &lpolestr);
```

結束 Microsoft 專有

## 另請參閱

[具有一元運算子的運算式](#)

[關鍵字](#)

# 加法類運算子：+ 和 -

2020/4/15 • [Edit Online](#)

## 語法

```
expression + expression  
expression - expression
```

## 備註

加法類運算子為：

- 加法\*\*+( )
- 減法-( )

這些二進位運算子具有由左至右的順序關聯性。

加法類運算子接受算術或指標類型運算元。添加\*\*+( ) 運算子的結果是操作數的總和。減法-( ) 運算符的結果是操作數之間的差異。如果一個或兩個運算元為指標，它們必須是物件的指標，而不是函式的指標。如果兩個運算元都是指標，除非兩個運算元都是同一個陣列中的物件指標，否則結果並沒有意義。

加法運算符採用算術、積分和標量類型的操作數。其定義如下表所列。

### 搭配加法類運算子使用的類型

II	II
演演算法	整數和浮點類型統稱為「算術」類型。
積分	各種大小 (long, short) 的 char 和 int 及列舉是「整數」類型。
純量 (scalar)	純量運算元是算術或指標類型的運算元。

這些運算子的有效組合包括：

算術 + arithmetic

標量 + 積分

整體 + 標量

算術 - arithmetic

黃 - 牛標量

請注意，加法和減法並非對等的運算。

如果兩個操作數都是算術類型，則標準轉換中介紹的轉換將應用於操作數，並且結果是轉換的類型。

## 範例

```

// 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;
}

```

## 指標加法

如果其中一個加法運算的運算元是物件陣列的指標，其他運算元必須是整數類型。結果會是與原始指標相同類型且指向另一個陣列元素的指標。下列程式碼片段說明這個概念：

```

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;
}

```

雖然已將整數值 1 加入至 `pIntArray`，但並不代表「對位址加 1」，而是表示「將指標調整為指向陣列中的下一個物件」，而其距離正好是 2 個位元組 (或 `sizeof( int )`)。

### NOTE

`pIntArray = pIntArray + 1` 形式的程式碼在 C++ 程式中很少見，若要執行遞增作業，使用下列形式會更好：  
`pIntArray++` 或 `pIntArray += 1`。

## 指標減法

如果兩個運算元都是指標，則減法運算的結果為兩個運算元之間的差數 (以陣列元素為單位)。減法運算式生成類型 `ptrdiff_t` 的簽名積分結果(在標準中定義,包括檔<stddef.h>)。

兩個運算元中的第二個運算元可以是整數類資料類型。減法運算的結果與原始指標的類型相同。減法的值是指向  $(n - i)$  th 陣列元素的指標,其中  $n$ 是原始指標指向的元素 , $i$ 是第二個操作數的整數值。

## 另請參閱

[具有二元運算子的運算式](#)

[C++ 內建運算子、優先順序和順序關聯性](#)

[C 加法類運算子](#)

# 位址運算子： &

2020/11/2 • [Edit Online](#)

## 語法

& *cast-expression*

## 備註

一元傳址運算子 (`&`) 接受其運算元的位址。傳址運算子的運算元可以是函式指示項或指定非位欄位之物件的左值。

Address 運算子只能套用至基本、結構、類別或等位型別的變數，這些變數是在檔案範圍層級宣告，或是套用至下標陣列參考。在這些運算式中，不包含 address 運算子的常數運算式可以在 address 運算式中加入或減去。

套用至函式或左值時，運算式的結果會是衍生自運算元類型的指標類型 (右值)。例如，如果運算元的類型為，則 `char` 運算式的結果為的型別指標 `char`。套用至或物件的 address 運算子 `const` `volatile` 會評估為 `const type *` 或 `volatile type *`，其中 `type` 是原始物件的型別。

多載函式的位址只會在清除參考的函式版本時才會執行。如需如何取得特定多載函式之 [位址的詳細](#) 資訊，請參閱函式多載。

當 address 運算子套用至限定名稱時，結果取決於 限定名稱 是否指定靜態成員。如果是，則結果為成員宣告中所指定類型的指標。若為非靜態成員，結果會是以 限定類別名稱 表示之類別成員名稱的指標。如需有關 限定類別名稱 的詳細資訊，請參閱 [主要運算式](#)。

## 範例：靜態成員的位址

下列程式碼片段會根據類別成員是否為靜態，顯示位址的運算子結果有何不同：

```
// 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
}
```

在這個範例中，因為 `&PTM::fValue` 是靜態成員，所以運算式 `float *` 會產生 `float PTM::*` 類型而不是 `fValue` 類型。

## 範例：參考型別的位址

將傳址運算子套用至參考類型所產生的結果，與將運算子套用至參考繫結所在的物件產生的結果相同。例如：

```
// 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;
}
```

```
&d equals &rd
```

## 範例:函數位址 as 參數

下列範例會使用傳址運算子將指標引數傳遞至函式：

```
// 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
}
```

25

## 另請參閱

[具有一元運算子的運算式](#)

[C++ 內建運算子、優先順序和關聯性](#)

[左值參考宣告子:&](#)

[間接取值和傳址運算子](#)

# 指派運算子

2020/11/2 • [Edit Online](#)

## 語法

### 運算式指派-運算子運算式

*assignment-operator*: 下列其中一個

=    \*=    /=    %=    +=    -=    <<=    >>=    &=    ^=    |=

## 備註

指派運算子會將值儲存在左運算元所指定的物件中。指派作業有兩種：

- **簡單指派**, 第二個運算元的值會儲存在第一個運算元所指定的物件中。
- **複合指派**, 在儲存結果之前, 會先執行算術、移位或位運算。

下表中的所有指派運算子( = 運算子除外)都是複合指派運算子。

### 指派運算子資料表

III	II
=	將第二個運算元的值儲存在第一個運算元所指定的物件 (簡單指派)。
*=	將第一個運算元的值乘以第二個運算元的值; 將結果儲存在第一個運算元所指定的物件。
/=	將第一個運算元的值除以第二個運算元的值; 將結果儲存在第一個運算元所指定的物件。
%=	將第一個運算元以第二個運算元的值取模數; 將結果儲存在第一個運算元所指定的物件。
+=	將第二個運算元的值加至第一個運算元的值; 將結果儲存在第一個運算元所指定的物件。
-=	將第一個運算元的值減去第二個運算元的值; 將結果儲存在第一個運算元所指定的物件。
<<=	將第一個運算元的值往左移位由第二個運算元的值所指定的位元數; 將結果儲存在第一個運算元所指定的物件。
>>=	將第一個運算元的值往右移位由第二個運算元的值所指定的位元數; 將結果儲存在第一個運算元所指定的物件。
&=	取得第一和第二個運算元的位元 AND; 將結果儲存在第一個運算元所指定的物件。
^=	取得第一和第二個運算元的位元排除 OR; 將結果儲存在第一個運算元所指定的物件。

=	取得第一和第二個運算元的位元包含 OR; 將結果儲存在第一個運算元所指定的物件。
---	--

## 運算子關鍵字

有三個複合指派運算子具有關鍵詞對應專案。其中包括：

&=	and_eq
=	or_eq
^=	xor_eq

C++ 會針對複合指派運算子，將這些運算子關鍵字指定為替代的拼寫。在 C 中，替代的拼寫是以宏的形式在 `<iso646.h>` 標頭中提供。在 C++ 中，替代的拼寫是關鍵字；使用 `<iso646.h>` 或 C++ 對等用法 `<ciso646>` 已被取代。在 Microsoft C++ 中，`/permissive- /Za` 需要或編譯器選項才能啟用替代的拼寫。

## 範例

```
// 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;
}
```

## 單一指派

簡單指派運算子 (`=`) 會導致第二個運算元的值儲存在第一個運算元所指定的物件中。如果這兩個物件都是算術類型，則在儲存值之前，右運算元會轉換成左側的類型。

`const` 和類型的物件 `volatile` 可以指派給只有或不是或的類型的左值 `volatile const volatile`。

指派給類別類型 (`struct`、`union` 和 `class` 類型) 的物件是由名為的函式所執行 `operator=`。這個運算子函式的預設行為是執行位元複製；不過可以使用多載運算子修改此行為。如需詳細資訊，請參閱 [運算子多載](#)。類別類型也可以有 [複製指派](#) 和 [移動指派](#) 運算子。如需詳細資訊，請參閱 [複製函數](#) 和 [複製指派運算子](#) 和 [移動函式](#) 和 [移動指派運算子](#)。

只要物件是從指定基底類別明確衍生的任何類別，就可以指派給該基底類別的物件。相反的，因為有從衍生類別到基類的隱含轉換，而不是從基類到衍生類別。例如：

```

// exre_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
}

```

參考類別的指派作用就像指派參考所指向的物件。

對於類別類型的物件而言，指派與初始化不同。為說明指派與初始化的不同之處，假設下列程式碼

```

UserType1 A;
UserType2 B = A;

```

上述程式碼示範初始化設定式；它會呼叫建構函式 `UserType2`，此建構函式接受屬於類型 `UserType1` 的引數。假設下列程式碼

```

UserType1 A;
UserType2 B;

B = A;

```

### 指派陳述式

```
B = A;
```

可能會有下列其中一項作用：

- 呼叫的函式，並提供 `operator= UserType2 operator= UserType1` 引數給。
- 如果 `UserType1::operator UserType2` 函式存在，呼叫這類明確轉換函式。
- 呼叫建構函式 `UserType2::UserType2`，前提是這類建構函式存在，而且接受 `UserType1` 引數並複製結果。

## 複合指派

複合指派運算子會顯示在[指派運算子資料表](#)中。這些運算子的形式為  $e1 \ op = e2$ ，其中  $e1$  是不可修改的 `const` 左值，而  $e2$  是：

- 算術類型

- 如果  $op$  為或，則為指標  $\boxed{+} \ast\ast\ast \boxed{-}$

$E1 \ op = e2$  表單的行為是  $e1 \boxed{=} e1 \ op e2$ ，但是  $e1$  只會評估一次。

對列舉類型的複合指派會產生錯誤訊息。如果左運算元是指標類型，則右運算元必須是指標類型，或者必須是判斷值為0的常數運算式。當左運算元是整數類資料類型時，右運算元必須不是指標類型。

## 指派運算子的結果

指派運算子會在指派之後傳回左運算元所指定的物件值。結果的類型會是左運算元的類型。指派運算式的結果一律是左值。這些運算子具有由右到左的順序關聯性。左運算元必須是可修改的左值。

在 ANSI C 中，指派運算式的結果不是左值。這表示  $(a \ += b) \ += c$  中不允許使用合法的  $c++$  運算式。

## 另請參閱

[具有二元運算子的運算式](#)

[C++ 內建運算子、優先順序和關聯性](#)

[C 指派運算子](#)

# 位 AND 運算子 : &

2020/11/2 • [Edit Online](#)

## 語法

運算式 \* & \*\*\*運算式

## 備註

位 AND 運算子( `&` )會比較第一個運算元的每個位與第二個運算元的對應位。如果這兩個位元都是 1, 則對應的結果位元會設為 1。否則, 對應的結果位元會設為 0。

位 AND 運算子的兩個運算元都必須有整數類資料類型。標準轉換中所涵蓋的一般算術轉換適用於運算元。

## & 的 Operator 關鍵字

C ++ 會 `bitand` 將指定為的替代拼寫 `&`。在 C 中, 替代的拼寫是以宏的形式在 `<iso646.h>` 標頭中提供。在 C ++ 中, 替代的拼寫是關鍵字;使用 `<iso646.h>` 或 C ++ 對等用法 `<ciso646>` 已被取代。在 Microsoft C ++ 中, `/permissive-` /za 需要或編譯器選項才能啟用替代的拼寫。

## 範例

```
// expre_Bitwise_AND_Operator.cpp
// compile with: /EHsc
// Demonstrate bitwise AND
#include <iostream>
using namespace std;
int main() {
    unsigned short a = 0xFFFF;      // pattern 1111 ...
    unsigned short b = 0xAAAA;      // pattern 1010 ...

    cout << hex << ( a & b ) << endl;   // prints "aaaa", pattern 1010 ...
}
```

## 另請參閱

[C ++ 內建運算子、優先順序和關聯性](#)

[C 位元運算子](#)

# 位元互斥 OR 運算子：^

2020/11/2 • [Edit Online](#)

## 語法

運算式 \*  $\wedge$  \*\*\* 運算式

## 備註

位互斥 OR 運算子 () 會將  $\wedge$  其第一個運算元的每個位與第二個運算元的對應位進行比較。如果其中一個運算元的位為0，而另一個運算元中的位為1，則對應的結果位會設為1。否則，對應的結果位元會設為0。

運算子的兩個運算元都必須有整數類資料類型。[標準轉換](#)中涵蓋的一般算術轉換適用於運算元。

如需  $\wedge$  c + +/cli 和 c + +/cx 中字元替代用法的詳細資訊，請參閱 [\(^\) \(c + +/Cli 和 c + +/cx\) 的物件運算子控制碼](#)。

## ^ 的 Operator 關鍵字

C + + `xor` 將指定為的替代拼寫  $\wedge$ 。在 C 中，會在標頭中以宏的形式提供替代的拼寫 `<iso646.h>`。在 c + + 中，替代的拼寫是關鍵字；使用 `<iso646.h>` 或 c + + 對等專案 `<ciso646>` 已被取代。在 Microsoft c + + 中，`/permissive-` `/Za` 需要或編譯器選項才能啟用替代的拼寫。

## 範例

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

## 另請參閱

[C + + 內建運算子、優先順序和關聯性](#)

# 位包含 OR 運算子 : |

2020/11/2 • [Edit Online](#)

## 語法

運算式 \* | \*\*\* 運算式2

## 備註

位包含 OR 運算子 ( | ) 會比較其第一個運算元的每個位與第二個運算元的對應位。如果其中一個位元是 1，則對應的結果位元會設為 1。否則，對應的結果位元會設為 0。

運算子的兩個運算元都必須有整數類資料類型。[標準轉換](#) 中所涵蓋的一般算術轉換適用於運算元。

## | 的 Operator 關鍵字

C ++ 會 `bitor` 將指定為的替代拼寫 | 。在 C 中，替代的拼寫是以宏的形式在 `<iso646.h>` 標頭中提供。在 c + + 中，替代的拼寫是關鍵字；使用 `<iso646.h>` 或 c + + 對等用法 `<ciso646>` 已被取代。在 Microsoft c + + 中，`/permissive-` /za 需要或編譯器選項才能啟用替代的拼寫。

## 範例

```
// 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 = 0xFFFF;      // pattern 1010 ...

    cout << hex << ( a | b ) << endl;   // prints "ffff" pattern 1111 ...
}
```

## 另請參閱

[C ++ 內建運算子、優先順序和關聯性](#)

[C 位元運算子](#)

# 轉型運算子：()

2020/11/2 • [Edit Online](#)

類型轉換提供了一個明確的轉換方法，可在特定情況下轉換物件的類型。

## 語法

```
unary-expression ( type-name ) cast-expression
```

## 備註

所有一元運算式皆會視為 Cast 運算式。

在類型轉換完成後，編譯器會將 *cast-expression* 視為 *type-name* 類型。轉型可用來將任何純量類型的物件與任何其他純量類型的物件來回轉換。明確類型轉換受到與判斷隱含轉換效果的相同規則所限制。對轉型的其他限制可能來自特定類型的實際大小或表示。

## 範例

```
// 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;
}
```

```

// 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);
}

```

```

RawBytes truncated to 10 chars: Exciting!!
Casted Bytes: Exciting
Note that the cast changed the raw internal string
Raw Bytes after cast: Exciting

```

## 另請參閱

[具有一元運算子的運算式](#)

[C++ 內建運算子、優先順序和順序關聯性](#)

[明確類型轉換運算子: \(# B1\)](#)

[轉換運算子](#)

[Cast 運算子](#)

# 逗號運算子：，

2020/3/25 • [Edit Online](#)

允許將兩個陳述式設為群組，其中一個是必要項。

## 語法

```
expression , expression
```

## 備註

逗號運算子具有由左到右的順序關聯性 (Associativity)。以逗號分隔的兩個運算式會由左至右求值。左運算元一定會進行求值，而且所有副作用都會在右運算元求值之前完成。

逗號可以在某些內容中做為分隔符號使用，例如，函式引數清單。請勿混淆做為分隔符號使用的逗號，與做為運算子使用的逗號，這兩種用法完全不同。

以 `e1, e2` 運算式為例。運算式的類型和值是 `e2` 的類型和值。會捨棄評估 `e1` 的結果。如果右運算元是左值，則結果會是左值。

逗號通常做為分隔符號使用 (例如，在函式的實際引數中或彙總初始設定式中)，而逗號運算子及其運算元必須以括號括住。例如：

```
func_one( x, y + 2, z );
func_two( (x--, y + 2), z );
```

在上面 `func_one` 的函式呼叫中，會傳遞三個以逗號分隔的引數：`x`、`y + 2` 和 `z`。在 `func_two` 的函式呼叫中，括號會強制編譯器將第一個逗號解譯為循序求值運算子。這個函式呼叫會傳遞兩個引數至 `func_two`。第一個引數是循序求值運算 `(x--, y + 2)` 的結果，具有 `y + 2` 運算式的值和類型，而第二個引數為 `z`。

## 範例

```
// cpp_comma_operator.cpp
#include <stdio.h>
int main () {
    int i = 10, b = 20, c = 30;
    i = b, c;
    printf("%i\n", i);

    i = (b, c);
    printf("%i\n", i);
}
```

```
20
30
```

## 另請參閱

[具有二元運算子的運算式](#)

C++ 內建運算子、優先順序和順序關聯性  
循序評估運算子

# 條件運算子 : ? :

2020/11/2 • [Edit Online](#)

## 語法

```
expression ? expression : expression
```

## 備註

條件運算子(?)是一種三元運算子(它會採用三個運算元)。條件運算子運作方式如下：

- 第一個運算元會隱含地轉換成 `bool`。接著會對它進行評估，並且完成所有副作用，再繼續執行。
- 如果第一個運算元評估為 `true` (1)，則會評估第二個運算元。
- 如果第一個運算元評估為 `false` (0)，則會評估第三個運算元。

條件運算子的結果會是所評估運算元(也就是第二個或第三個運算元)的結果。在條件運算式中，只會評估後兩個運算元的其中一個。

條件運算式具有由右至左順序關聯性。第一個運算元必須為整數類資料類型或指標類型。下列規則適用於第二個和第三個運算元：

- 如果這兩個運算元屬於相同類型，則結果為該類型。
- 如果兩個運算元都是算術或列舉類型，則會執行一般算術轉換(涵蓋在[標準轉換](#)中)，將它們轉換成一般類型。
- 如果這兩個運算元都是指標類型，或者其中一個是指標類型，另一個是判斷值為 0 的常數運算式，則會進行指標轉換，將它們轉換成一般類型。
- 如果這兩個運算元屬於參考類型，則會進行參考轉換，將它們轉換成一般類型。
- 如果這兩個運算元屬於 `void` 類型，則一般類型會是 `void` 類型。
- 如果這兩個運算元屬於相同的使用者定義類型，則一般類型會是該類型。
- 如果運算元的類型不同，且至少其中一個運算元具有使用者定義類型，則會使用語言規則來判斷一般類型。  
(請參閱下列警告)。

未包含在上述清單中的任何第二個和第三個運算元組合都不合法。結果的類型是一般類型，而如果第二個和第三個運算元屬於相同類型且都是左值，則結果也會是左值。

### WARNING

如果第二個運算元的類型與第三個運算元的類型不同，則會叫用依照 C++ 標準指定的複雜類型轉換規則。這些轉換可能會導致非預期的行為，包括建構和解構暫存物件。因此，強烈建議您 (1) 避免使用使用者定義類型做為搭配條件運算子的運算元，或者 (2) 如果您使用使用者定義類型，則明確地將每個運算元轉換成一般類型。

## 範例

```
// expe_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;
}
```

## 另請參閱

[C++ 內建運算子、優先順序和順序關聯性](#)

[條件運算式運算子](#)

# delete 運算子 (C++)

2020/11/2 • [Edit Online](#)

取消配置記憶體區塊。

## Syntax

```
[ :: ] delete cast-運算式  
[ :: ] delete [] cast-運算式
```

## 備註

*Cast 運算式*引數必須是先前配置給以[new 運算子](#)建立之物件的記憶體區塊指標。`delete` 運算子具有類型的結果`void`，因此不會傳回值。例如：

```
CDialog* MyDialog = new CDialog;  
// use MyDialog  
delete MyDialog;
```

`delete` 在未配置的物件指標上使用，會`new` 提供無法預期的結果。不過，您可以`delete` 在值為0的指標上使用。此布建表示，當`new` 失敗時傳回0時，刪除失敗作業的結果`new` 是無害的。如需詳細資訊，請參閱[new 和 Delete 運算子](#)。

`new` 和`delete` 運算子也可以用於內建類型(包括陣列)。如果`pointer` 參考陣列，請在前面放置空的括弧`([])`  
`pointer`：

```
int* set = new int[100];  
//use set[]  
delete [] set;
```

`delete` 在物件上使用運算子會將其記憶體解除配置。在刪除物件之後將指標取值的程式可能會有無法預期的結果或損毀。

當`delete` 用來解除配置 c++ 類別物件的記憶體時，會在物件的記憶體解除配置之前呼叫物件的「函式」((如果物件具有) 的函式)。

如果運算子的運算元`delete` 是可修改的左值，則在刪除物件之後，就不會定義其值。

如果指定了[/sdl \(啟用其他安全性檢查\)](#) 編譯器選項，則在`delete` 刪除物件之後，運算子的運算元會設定為不正確值。

## 使用 delete

[Delete 運算子](#)有兩種語法變體：一個用於單一物件，另一個用於物件陣列。下列程式碼片段顯示其差異：

```
// expre_Using_delete.cpp
struct UDTtype
{
};

int main()
{
    // Allocate a user-defined object, UDOobject, and an object
    // of type double on the free store using the
    // new operator.
    UDTtype *UDOobject = new UDTtype;
    double *dObject = new double;
    // Delete the two objects.
    delete UDOobject;
    delete dObject;
    // Allocate an array of user-defined objects on the
    // free store using the new operator.
    UDTtype (*UDArr)[7] = new UDTtype[5][7];
    // Use the array syntax to delete the array of objects.
    delete [] UDArr;
}
```

下列兩種情況會產生未定義的結果：使用陣列形式的 `delete`（在 `delete []` 物件上），以及在陣列上使用 `nonarray` 形式的 `delete` 形式。

## 範例

如需使用的範例 `delete`，請參閱 [new 運算子](#)。

## delete 運作方式

Delete 運算子會叫用函數 `運算子 delete`。

對於不屬於類別類型（[類別、結構或等位](#)）的物件，則會叫用全域 `delete` 運算子。對於類別類型的物件，如果刪除運算式以一元範圍解析運算子開頭 `()`，則會在全域範圍中解析解除配置函數的名稱 `::`。否則，如果指標不是 `Null`，`delete` 運算子會在解除配置記憶體之前叫用物件的解構函式。`delete` 運算子可以依類別來定義；如果指定類別沒有這類定義，會叫用全域 `delete` 運算子。如果使用刪除運算式來解除配置靜態類型包含虛擬解構函式的類別物件，則會透過該物件之動態類型的虛擬解構函式來解析解除配置函式。

## 另請參閱

[具有一元運算子的運算式](#)

[關鍵字](#)

[new 和 delete 運算子](#)

# 等號比較運算子 : == 和 !=

2020/11/2 • [Edit Online](#)

## 語法

運算式 \* `==` \*\*\*運算式

運算式 \* `!=` \*\*\*運算式

## 備註

二進位相等運算子會比較其運算元，以進行嚴格的相等或不等比較。

等號比較運算子(等於(`==`)和不等於(`!=`))的優先順序比關聯式運算子低，但其行為類似。這些運算子的結果型別是`bool`。

`== true` 如果兩個運算元具有相同的值，則等於運算子()會傳回，否則會傳回`false`。`!= true` 如果運算元的值不相同，則不等於運算子()會傳回，否則會傳回`false`。

## != 的 Operator 關鍵字

C++ 會`not_eq` 將指定為的替代拼寫`!=`。(沒有的替代拼寫`==`)。在 C 中，替代的拼寫是以宏的形式在`<iso646.h>` 標頭中提供。在 c++ 中，替代的拼寫是關鍵字；使用`<iso646.h>` 或 c++ 對等用法`<ciso646>`已被取代。在 Microsoft c++ 中，`/permissive- /Za` 需要或編譯器選項才能啟用替代的拼寫。

## 範例

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

using namespace std;

int main() {
    cout << boolalpha
        << "The true expression 3 != 2 yields: "
        << (3 != 2) << endl
        << "The false expression 20 == 10 yields: "
        << (20 == 10) << endl;
}
```

相等運算子可以比較相同類型成員的指標。在這種比較中，會執行指標對成員的轉換。成員指標也可以與判斷值為 0 的常數運算式進行比較。

## 另請參閱

[具有二元運算子的運算式](#)

[C++ 內建運算子, 優先順序和關聯性](#)

[C 關係和等號比較運算子](#)

# 明確類型轉換運算子：()

2020/4/15 • [Edit Online](#)

C++ 允許使用類似函式呼叫的語法進行明確的類型轉換。

## 語法

```
simple-type-name ( expression-list )
```

## 備註

簡單類型名稱後跟括弧中包含的運算式清單使用指定的運算式建構指定類型的物件。下列範例顯示將類型明確地轉換為類型 int：

```
int i = int( d );
```

下面的示例顯示了一個 Point 類。

## 範例

```

// 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();
}

```

## 輸出

```

x = 20, y = 10
x = 0, y = 0

```

雖然上述範例示範的是使用常數進行明確類型轉換，相同的技巧也可以在物件上進行這些轉換。下列程式碼片段示範這項功能：

```

int i = 7;
float d;

d = float( i );

```

使用「轉換」語法也可以指定明確類型轉換。上述範例若是使用轉換語法重新撰寫則會如下所示：

```

d = (float)i;

```

從單一值進行轉換時，轉換和函式樣式轉換的結果相同。不過，在函式樣式語法中，您可以在轉換時指定一個以上

的引數。這項差異對於使用者定義的類型而言是很重要的。請考慮 `Point` 類別及其轉換：

```
struct Point
{
    Point( short x, short y ) { _x = x; _y = y; }
    ...
    short _x, _y;
};

...
Point pt = Point( 3, 10 );
```

前面的範例使用函式樣式轉換，示範如何將兩個值(一個用於 *x* 和一個用於 *y*) 轉換為 `Point` 使用者定義的類型。

**Caution**

由於其中覆寫了 C++ 編譯器的內建類型檢查，因此請小心使用明確類型轉換。

**強制轉換** 記號必須用於轉換為沒有 **簡單類型名稱** 的類型(例如指標或引用類型)。轉換為可以使用 **簡單類型名稱** 表示的類型可以以任一形式編寫。

在轉換中定義類型是不合法。

## 另請參閱

[後置運算式](#)

[C++ 內建運算子、優先順序和順序關聯性](#)

# 函式呼叫運算子 : ()

2020/11/2 • [Edit Online](#)

函式呼叫是一種 `postfix-expression`，由評估為函式或可呼叫物件後面接著函式呼叫運算子的運算式所組成 `()`。物件可以宣告函式 `operator ()`，以提供物件的函式呼叫語義。

## 語法

```
postfix-expression :  
    postfix-expression ( argument-expression-list opt )
```

## 備註

函式呼叫運算子的引數來自 `argument-expression-list`（以逗號分隔的運算式清單）。這些運算式的值會當做引數傳遞至函式。引數運算式清單可以是空的。在 C++ 17 之前，函數運算式和引數運算式的評估順序並未指定，而且可能會以任何順序發生。在 C++ 17 和更新版本中，函數運算式會在任何引數運算式或預設引數之前進行評估。引數運算式會以不確定的順序進行評估。

會 `postfix-expression` 評估為要呼叫的函式。它可以採用數種形式的其中一種：

- 函式識別碼，可在目前的範圍中看見，或在提供的任何函式引數範圍中顯示。
- 評估為函式、函式指標、可呼叫物件，或參考至其中一個的運算式
- 成員函式存取子（明確或隱含）
- 成員函式的解除引用指標。

`postfix-expression` 可能是多載函數識別碼或多載成員函式存取子。多載解析的規則會決定要呼叫的實際函數。如果成員函式是虛擬的，則會在執行時間決定要呼叫的函式。

一些範例宣告：

- 傳回類型 `T` 的函式。如以下範例宣告所示

```
T func( int i );
```

- 傳回類型 `T` 之函式的指標。如以下範例宣告所示

```
T (*func)( int i );
```

- 傳回類型 `T` 之函式的參考。如以下範例宣告所示

```
T (&func)(int i);
```

- 傳回類型 `T` 的成員指標函式取值 (Dereference)。函式呼叫的範例如下

```
(pObject->*pmf)();  
(Object.*pmf)();
```

## 範例

下列範例呼叫具有三個引數的標準程式庫函式 `strcat_s`：

```
// expre_Function_Call_Operator.cpp
// compile with: /EHsc

#include <iostream>
#include <string>

// C++ Standard Library name space
using namespace std;

int main()
{
    enum
    {
        sizeOfBuffer = 20
    };

    char s1[ sizeOfBuffer ] = "Welcome to ";
    char s2[ ] = "C++";

    strcat_s( s1, sizeOfBuffer, s2 );

    cout << s1 << endl;
}
```

```
Welcome to C++
```

## 函式呼叫結果

除非函式宣告為參考型別，否則函式呼叫會評估為右值。具有參考傳回類型的函數會評估為左值。這些函數可以在指派語句的左邊使用，如下所示：

```

// 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";
}

```

上述程式碼會定義名為的類別 `Point`，其中包含代表 *x* 和 *y* 座標的私用資料物件。這些資料物件必須經過修改，而且必須擷取其值。這個程式只是這種類別的數項設計之一，使用 `GetX` 和 `SetX` 或 `GetY` 和 `SetY` 則是另一種可能的設計方式。

傳回類別類型、類別類型的指標或類別類型的參考之函式可以做為成員選擇運算子的左運算元使用。下列程式碼是合法的：

```
// 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 );
}
```

函式可以透過遞迴方式呼叫。如需函式宣告的詳細資訊，請參閱[函數](#)。相關的資料位於轉譯單位和連結中。

## 另請參閱

[後置運算式](#)

[C++ 內建運算子、優先順序和關聯性](#)

[函式呼叫](#)

# 間接取值運算子：\*

2020/3/25 • [Edit Online](#)

## 語法

```
* cast-expression
```

## 備註

一元間接運算子(\*)會取值指標;也就是說，它會將指標值轉換為左值。間接運算子的運算元必須是類型指標。間接運算式的結果是類型，指標類型即衍生自此。在此內容中使用\*運算子，其意義與二元運算子不同，也就是乘法。

如果運算元指向某個函式，則結果為函式指示項。如果運算元指向某個儲存位置，則結果為指定儲存位置的左值。

間接運算子可以累計用於將指標取值成為指標。例如：

```
// 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;
}
```

如果指標值無效，則結果會是未定義的。下列清單包含最常見的一些使指標值無效的情況。

- 指標是一個 null 指標。
- 指標指定了在參考時不可見的本機項目位址。
- 指標指定的位址與物件指向的類型並不一致。
- 指標指定了執行中程式未使用的位址。

## 另請參閱

[具有一元運算子的運算式](#)

[C++ 內建運算子、優先順序和順序關聯性](#)

[傳址運算子:&](#)

[間接取值和傳址運算子](#)

# 左移和右移運算子 ( `>>` 和 `<<` )

2020/11/2 • [Edit Online](#)

位移位運算子是右移運算子(`>>`)，它會將`shift`運算式的位向右移動，而左移運算子()`<<`則會`<<`將`shift`運算式的位向左移動。<sup>1</sup>

## 語法

```
shift 運算式 << 加法類運算式  
shift-expression >> additive-expression
```

## 備註

### IMPORTANT

下列說明和範例在適用于 x86 和 x64 架構的 Windows 上有效。在適用于 ARM 裝置的 Windows 上，左移和右移位運算子的執行方式明顯不同。如需詳細資訊，請參閱[HELLO ARM blog](#)文章的「移位運算子」一節。

## 左移位

左移運算子會使`shift`運算式中的位向左移動加總運算式所指定的位置數目。移位作業空出的位元位置以零填補。左移是邏輯移位(會捨棄移出結尾的位元，包括正負號位元)。如需位移位類型的詳細資訊，請參閱[位移位](#)。

下列範例示範使用不帶正負號數字的左移位作業。此範例以值代表 bitset 說明位元的行為。如需詳細資訊，請參閱[Bitset 類別](#)。

```
#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; // 0b00000000'0000100  
  
    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  
}
```

如果將帶正負號的數字左移，以影響正負號位元，結果會是未定義。下列範例顯示當1位被左移至符號位位置時，會發生什麼事。

```

#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
}

```

## 右移位

右移運算子會使 *shift* 運算式中的位模式向右移位 (由加總運算式所指定的位置數目)。若是不帶正負號的數字，移位作業空出的位元位置以零填補。若是帶正負號的數字，會使用正負號位元填補空出的位元位置。換句話說，如果是正數就會使用 0，負數則使用 1。

### IMPORTANT

將帶正負號數字向右移的結果與實作相關。雖然 Microsoft C++ 編譯器會使用正負號來填補空出的位位置，但並不保證其他的實現也會這麼做。

此範例示範使用不帶正負號數字的右移位作業：

```

#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
}

```

下一個範例示範使用正號數字的右移位作業。

```

#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
}

```

下一個範例示範使用帶負號整數右移位作業。

```

#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
}

```

## 移位和提升

移位運算子兩邊的運算式都必須是整數類型。整數提升會根據[標準轉換](#)中所述的規則來執行。結果的類型與已升級之移位運算式的類型相同。

在下列範例中，類型的變數 `char` 會升級為 `int`。

```

#include <iostream>
#include <typeinfo>

using namespace std;

int main() {
    char char1 = 'a';

    auto promoted1 = char1 << 1; // 194
    cout << typeid(promoted1).name() << endl; // int

    auto promoted2 = char1 << 10; // 99328
    cout << typeid(promoted2).name() << endl; // int
}

```

## 其他詳細資料

如果加法運算式為負數，或是加總運算式大於或等於(升級)移位運算式中的位數，則移位運算的結果會是未定義的。如果加法類運算式為0，則不會執行任何移位作業。

```

#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)
}

```

## 註腳

<sup>1</sup>以下是c++ 11 ISO 規格(INCITS/ISO/IEC 14882-2011 [2012])、區段5.8.2 和5.8.3 中的shift 運算子描述。

$E_1 \ll E_2$  的值是向左移位  $E_1$  的  $E_2$  位元位置；空出的位元會以零填補。如果  $E_1$  具有不帶正負號的類型，結果的值會是  $E_1 \times 2^{E_2}$ ，比結果類型中可顯示的最大值還要少一個模數。否則，如果  $E_1$  具有帶正負號的類型和非負數值，且  $E_1 \times 2^{E_2}$  可在結果類型的對應不帶正負號類型中顯示，則轉換為結果類型的值會是產生的值；否則，行為會是未定義的。

$E_1 \gg E_2$  的值是  $E_1$  向右移位  $E_2$  個位元位置。如果  $E_1$  具有不帶正負號的類型，或如果具有帶正負號的  $E_1$  類型和非負數值，則結果的值會是  $E_1 / 2^{E_2}$  之商的整數部分。如果  $E_1$  的類型帶正負號，且具有負數值，結果產生的值由實作決定。

## 另請參閱

[具有二元運算子的運算式](#)

[C++ 內建運算子、優先順序和順序關聯性](#)

# 邏輯 AND 運算子 : &&

2020/11/2 • [Edit Online](#)

## 語法

運算式 \* && \*\*\*運算式

## 備註

**&& true** 如果兩個運算元都是，則邏輯 AND 運算子()會傳回 `true`，否則會傳回 `false`。在評估之前，運算元會隱含轉換成類型 `bool`，而結果的類型為 `bool`。邏輯 AND 具有由左至右的順序關聯性。

邏輯 AND 運算子的運算元不需要具有相同的類型，但必須有布林值、整數或指標類型。運算元通常是關聯或相等運算式。

第一個運算元會經過完整評估，而且所有副作用都會在邏輯 AND 運算式的評估繼續進行之前完成。

只有在第一個運算元評估為 `true` (非零)時，才會評估第二個運算元。當邏輯 AND 運算式為時，此評估會排除第二個運算元的不必要評估 `false`。您可以使用這個最少運算評估來預防 null 指標取值，如下列範例所示：

```
char *pch = 0;  
// ...  
(pch) && (*pch = 'a');
```

如果 `pch` 是 null (0)，則永遠不會評估運算式的右邊。這種短路評估可讓您無法透過 null 指標進行指派。

## && 的 Operator 關鍵字

C ++ 會 `and` 將指定為的替代拼寫 `&&`。在 C 中，替代的拼寫是以宏的形式在 `<iso646.h>` 標頭中提供。在 C ++ 中，替代的拼寫是關鍵字；使用 `<iso646.h>` 或 C ++ 對等用法 `<ciso646>` 已被取代。在 Microsoft C ++ 中，`/permissive- /Za` 需要或編譯器選項才能啟用替代的拼寫。

## 範例

```
// expr_Logicl_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;  
}
```

## 另請參閱

[C++ 內建運算子、優先順序和關聯性](#)

[C 邏輯運算子](#)

# 邏輯否定運算子：!

2020/11/2 • [Edit Online](#)

## 語法

\* ! \*\*\*cast-運算式

## 備註

邏輯負運算子 ( ! ) 會反轉其運算元的意義。運算元必須是算術或指標類型 (或判斷值為算術或指標類型的運算式)。運算元會隱含地轉換成類型 `bool`。`true` 如果轉換的運算元為，則結果為 `false`；`false` 如果轉換的運算元為，則結果為 `true`。結果的類型為 `bool`。

若為運算式 `e`，一元運算式 `!e` 就相當於運算式 `(e == 0)`，但牽涉到多載運算子的情況除外。

## ！的 Operator 關鍵字

C++ 會 `not` 將指定為的替代拼寫 `!`。在 C 中，替代的拼寫是以宏的形式在 `<iso646.h>` 標頭中提供。在 C++ 中，替代的拼寫是關鍵字；使用 `<iso646.h>` 或 `c++` 對等用法 `<ciso646>` 已被取代。在 Microsoft C++ 中，`/permissive-` `/za` 需要或編譯器選項才能啟用替代的拼寫。

## 範例

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

int main() {
    int i = 0;
    if (!i)
        cout << "i is zero" << endl;
}
```

## 另請參閱

[具有一元運算子的運算式](#)

[C++ 內建運算子、優先順序和關聯性](#)

[一元算術運算子](#)

# 邏輯 OR 運算子 : ||

2020/11/2 • [Edit Online](#)

## 語法

邏輯 or 運算式 \* || \*\*\* 邏輯 and 運算式

## 備註

|| true 如果任一個或兩個運算元都是，則邏輯 OR 運算子()會傳回布林值 true，否則會傳回 false。在評估之前，運算元會隱含轉換成類型 bool，而結果的類型為 bool。邏輯 OR 具有由左至右的順序關聯性。

邏輯 OR 運算子的運算元不一定要有相同的類型，但它們必須是布林值、整數或指標類型。運算元通常是關聯或相等運算式。

第一個運算元會經過完整求值，並且會在所有副作用完成之後，才繼續求出邏輯 OR 運算式的值。

只有在第一個運算元評估為時，才會評估第二個運算元 false，因為當邏輯 OR 運算式為時，不需要進行評估 true。這就是所謂的短路評估。

```
printf( "%d" , (x == w || x == y || x == z) );
```

在上述範例中，如果 x 等於 w 或，則函式 y z 的第二個引數會 printf 評估為 true，然後將它升級為整數，並列印值1。否則，它會評估為 false，並列印值0。一旦其中一個條件評估為，就會 true 停止評估。

## || 的 Operator 關鍵字

C ++ 會 or 將指定為的替代拼寫 ||。在 C 中，替代的拼寫是以宏的形式在 <iso646.h> 標頭中提供。在 c ++ 中，替代的拼寫是關鍵字;使用 <iso646.h> 或 c ++ 對等用法 <ciso646> 已被取代。在 Microsoft c ++ 中，  
`/permissive- /za` 需要或編譯器選項才能啟用替代的拼寫。

## 範例

```
// expr_Logic_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;
}
```

## 另請參閱

C++ 內建運算子、優先順序和關聯性

C 邏輯運算子

# 成員存取運算子：。 和->

2020/3/25 • [Edit Online](#)

## 語法

```
postfix-expression . name  
postfix-expression -> name
```

## 備註

成員存取運算子 。 和 -> 是用來參考結構、等位和類別的成員。成員存取運算式具有選定成員的值和類型。

成員存取運算式有兩種形式：

1. 在第一個表單中，後置運算式代表 struct、class 或 union 類型的值，而 *name* 則是指定的結構、等位或類別的成員。如果後置運算式是左值，則作業的值是名稱的，而且是左值。
2. 在第二種形式中，後置運算式代表結構、等位或類別的指標，而名稱則是指定的結構、等位或類別的成員。值為名稱，而且是左值。-> 運算子會將指標取值。因此，`e->member` 和 `(*e).member` (其中 *e* 代表指標) 的運算式會產生相同的結果 (除非 -> 或 \* 的運算子超載)。

## 範例

下列範例示範成員存取運算子的兩種形式。

```
// expre_Selection_Operator.cpp  
// compile with: /EHsc  
#include <iostream>  
using namespace std;  
  
struct Date {  
    Date(int i, int j, int k) : day(i), month(j), year(k){}  
    int month;  
    int day;  
    int year;  
};  
  
int main() {  
    Date mydate(1,1,1900);  
    mydate.month = 2;  
    cout << mydate.month << "/" << mydate.day  
    << "/" << mydate.year << endl;  
  
    Date *mydate2 = new Date(1,1,2000);  
    mydate2->month = 2;  
    cout << mydate2->month << "/" << mydate2->day  
    << "/" << mydate2->year << endl;  
    delete mydate2;  
}
```

```
2/1/1900  
2/1/2000
```

## 另請參閱

[後置運算式](#)

[C++ 內建運算子、優先順序和順序關聯性](#)

[類別和結構](#)

[結構和等位成員](#)

# 乘法類運算子和模數運算子

2020/11/2 • [Edit Online](#)

## 語法

```
expression * expression  
expression / expression  
expression % expression
```

## 備註

乘法類運算子包括：

- 乘法( \* )
- 除法( / )
- 模數(從除法餘數)( % )

這些二進位運算子具有由左至右的順序關聯性。

乘法類運算子接受算術類型的運算元。模數運算子( % )具有較嚴格的要求，其運算元必須是整數類資料類型。  
(若要取得浮點除法的餘數，請使用執行時間函式fmod)。標準轉換中涵蓋的轉換適用於運算元，而結果則是轉換後的類型。

乘法運算子會產生第一個運算元與第二個運算元相乘的結果。

除法運算子會產生第一個運算元除以第二個運算元的結果。

模數運算子會產生下列運算式所指定的餘數，其中e1是第一個運算元，而e2是第二個： $e1 - (e1 / e2) * e2$ ，其中兩個運算元都是整數類資料類型。

在除法或模數運算式中除以 0 並未定義，而且會產生執行階段錯誤。因此，下列運算式會產生未定義的錯誤結果：

```
i % 0  
f / 0.0
```

如果乘法、除法或模數運算式的兩個運算元有相同的正負號，則結果為正數。否則，結果為負數。模數運算結果的正負號是由實作所定義。

### NOTE

由於乘法類運算子所執行的轉換不提供溢位或反向溢位條件，因此，如果乘法類運算的結果無法以轉換後的運算元類型表示，則資訊可能會遺失。

## Microsoft 特定的

在 Microsoft C++ 中，模數運算式的結果一律與第一個運算元的正負號相同。

## 結束 Microsoft 專有

如果兩個整數計算的除法不精確，而且只有一個運算元為負數，則結果會是小於除法運算會產生之實際值的最大整數(範圍內，忽略正負號)。例如， $-11/3$  的計算值是-3.666666666。整數除法的結果為-3。

乘法類運算子之間的關聯性是由身分識別( $e1 / e2) * e2 + e1 \% e2 == e1$ 所提供。

## 範例

下列程式將示範乘法類運算子。請注意，的任一個運算元 `10 / 3` 必須明確轉換成類型 `float`，以避免截斷，讓這兩個運算元在除法之前都屬於類型 `float`。

```
// expr_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;
}
```

## 另請參閱

[具有二元運算子的運算式](#)

[C++ 內建運算子、優先順序和順序關聯性](#)

[C 乘法類運算子](#)

# new 運算子 (C++)

2020/11/2 • [Edit Online](#)

從可用存放區為 **類型名稱** 的物件或物件陣列配置記憶體，並傳回適當類型的非零指標給物件。

## NOTE

Microsoft C++ 元件擴充功能會提供關鍵字的支援 `new`，以新增 vtable 位置專案。如需詳細資訊，請參閱 [新的 \(vtable 中的新位置\)](#)

## 語法

```
[::] new [placement] new-type-name [new-initializer]  
[::] new [placement] ( type-name ) [new-initializer]
```

## 備註

如果不成功，則傳回 `new` 零或擲回例外狀況；如需詳細資訊，請參閱 [new 和 delete 運算子](#)。您可以藉由撰寫自訂例外狀況處理常式，並使用您的函式名稱做為其引數來呼叫 `_set_new_handler` 執行時間程式庫函式，來變更此預設行為。

如需有關如何在 managed 堆積上建立物件的詳細資訊，請參閱 [gcnew](#)。

當 `new` 用來配置 C++ 類別物件的記憶體時，會在配置記憶體之後呼叫物件的函式。

使用 `delete` 運算子來解除配置以運算子配置的記憶體 `new`。

下列範例會先配置然後再釋放大小為 `dim` 乘以 10 個字元的二維陣列。配置多維陣列時，第一個維度以外的所有維度都必須是判斷值為正值的常數運算式；最左邊的陣列維度可以是任何判斷值為正值的運算式。使用運算子配置陣列時 `new`，第一個維度可以為零，而運算子會傳回 `new` 唯一指標。

```
char (*pchar)[10] = new char[dim][10];  
delete [] pchar;
```

類型名稱不能包含 `const`、`volatile`、類別宣告或列舉宣告。因此，下列運算式是不合法的：

```
volatile char *vch = new volatile char[20];
```

`new` 運算子不會配置參考型別，因為它們不是物件。

`new` 運算子不能用來配置函式，但是可以用來配置函數的指標。下列範例會配置然後再釋放含七個傳回整數之函式指標的陣列。

```
int (**p) () = new (int (*[7]) ());
delete *p;
```

如果您在 `new` 沒有任何額外引數的情況下使用運算子，並使用 `/Gx`、`/eha` 或 `/ehs` 選項進行編譯，則編譯器會產生程式碼，以便在函式擲回例外狀況時呼叫運算子 `delete`。

下列清單描述的文法元素 `new` :

#### 位置

當您多載時，提供傳遞其他引數的方法 `new`。

#### 類型名稱

指定要配置的類型；它可以是內建或使用者定義的類型。如果類型規格是複雜的，請以括號括住類型規格以強制繫結的順序。

#### 初始設定式

提供初始化物件的值。初始設定式無法指定給陣列。`new` 只有當類別具有預設的函式時，運算子才會建立物件的陣列。

## 範例：配置和釋放字元陣列

下列程式碼範例會配置一個字元陣列和一個 `CName` 類別物件，然後加以釋放。

```
// 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;
}
```

## 範例：`new` 運算子

如果您使用運算子的放置新表單 `new`，此表單除了配置的大小之外，還包含引數的表單，如果函式擲回例外狀況，則編譯器不支援運算子的放置形式 `delete`。例如：

```

// expre_new_Operator2.cpp
// C2660 expected
class A {
public:
    A(int) { throw "Fail!"; }
};

void F(void) {
    try {
        // heap memory pointed to by pa1 will be deallocated
        // by calling ::operator delete(void*).
        A* pa1 = new A(10);
    } catch (...) {
    }
    try {
        // This will call ::operator new(size_t, char*, int).
        // When A::A(int) does a throw, we should call
        // ::operator delete(void*, char*, int) to deallocate
        // the memory pointed to by pa2. Since
        // ::operator delete(void*, char*, int) has not been implemented,
        // memory will be leaked when the deallocation cannot occur.

        A* pa2 = new(__FILE__, __LINE__) A(20);
    } catch (...) {
    }
}

int main() {
    A a;
}

```

## 初始化以 new 所配置的物件

選擇性的 **初始化運算式** 欄位包含在運算子的文法中 `new`。這樣就可讓您以使用者定義的建構函式初始化新物件。如需初始化如何完成的詳細資訊，請參閱**初始化運算式**。下列範例說明如何使用初始化運算式搭配 `new` 運算子：

```

// 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 );
    // ...
}

```

在此範例中，物件 `CheckingAcct` 是使用運算子進行配置 `new`，但未指定預設初始化。因此會呼叫 `Acct()` 類別的預設建構函式。然後以相同方式配置 `SavingsAcct` 物件，不過它會明確初始化為 34.98。因為 34.98 的型別為 `double`，所以會呼叫採用該型別的引數的函式來處理初始化。最後，非類別類型 `HowMuch` 會初始化為 43.0。

如果物件是類別類型，而且該類別具有與上述範例 (一樣的函式)，則 `new` 只有在符合下列其中一個條件時，運

算子才能初始化物件：

- 在初始設定式中提供的引數與建構函式中的引數一致。
- 類別有預設建構函式(可以在沒有引數的情況下呼叫的建構函式)。

使用運算子配置陣列時，不可以進行明確的每個元素的初始化，而 `new` 只會呼叫預設的函式(如果有的話)。如需詳細資訊，請參閱 [預設引數](#)。

如果記憶體配置失敗(運算子 `new` 會傳回 0)的值，不會執行任何初始化。這樣可防止嘗試初始化不存在的資料。

就像函式呼叫一般，不會定義初始化運算式的評估順序。此外，您不應依賴這些運算式會在執行記憶體配置之前完全評估。如果記憶體配置失敗，而且 `new` 運算子傳回零，則初始化運算式中的某些運算式可能不會完全評估。

## 以 `new` 所配置物件的存留期

使用運算子所配置的物件在其 `new` 定義的範圍結束時，不會終結。因為 `new` 運算子會傳回其所設定物件的指標，所以程式必須定義具有適當範圍的指標，才能存取這些物件。例如：

```
// 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.
}
```

一旦指標 `AnotherArray` 超出範例中的範圍，就無法刪除物件。

## `new` 運作方式

配置運算式(包含運算子的運算式 `new`)會執行三項作業：

- 為要配置的物件找出並保留存放區。這個階段完成時會配置正確數量的存放區，但還不是物件。
- 將物件初始化。初始化完成後，就會出現使配置存放區成為物件的足夠資訊。
- 傳回物件的指標，該物件會從 [新的型別名稱或 型別名稱衍生的指標類型](#)(`s`)。程式會使用此指標來存取新配置的物件。

運算子會叫用 `new` 函數 `運算子 new`。針對任何類型的陣列，以及不屬於 `class`、或類型的物件 `struct`、`union`，會呼叫全域函式 `:: operator new`來配置儲存區。類別型別物件可以針對每個類別，定義自己的 `operator new` 靜態成員函式。

當編譯器遇到 `new` 運算子來配置類型類型的物件時，`type` 它會發出呼叫 `type :: operator new ( sizeof (`

`type`) ) 或者, 如果未定義使用者定義的operator new , 則 :: operator new ( sizeof( `type` \*\*\* ) ) 。因此, `new` 操作員可以設定物件的正確記憶體數量。

#### NOTE

New 的引數屬於類型 `size_t` 。此類型定義于 <direct.h> 、 <malloc.h> 、 <memory.h> <search.h> <stddef.h> <stdio.h> <stdlib.h> <string.h> 和 <time.h> 。

文法中的選項可提供 位置 的規格 (請參閱 [new 運算子](#)) 的文法。位置參數只能用於運算子 new 的使用者定義實值;它允許將額外的資訊傳遞給operator new 。如果類別 T 的成員運算子是 new , 則具有 放置 欄位(例如)的運算式 `T *TObject = new ( 0x0040 ) T;` 會轉譯為, 否則會轉譯為

```
T *TObject = T::operator new( sizeof( T ), 0x0040 ); T *TObject = ::operator new( sizeof( T ), 0x0040 ); 。
```

放置欄位的原始意圖是允許在使用者指定的位址配置硬體相依的物件。

#### NOTE

雖然上述範例只會在 [ 位置 ] 欄位中顯示一個引數, 但這並不會限制可透過這種方式將多少額外的引數傳遞給 operator new 。

即使已經為類別類型定義 operator new , 也可以使用此範例的形式來使用 global 運算子:

```
T *TObject = ::new TObject;
```

範圍解析運算子 ( :: ) 強制使用 global `new` 運算子。

## 另請參閱

[具有一元運算子的運算式](#)

[關鍵字](#)

[new 和 delete 運算子](#)

# 一補數運算子：~

2020/11/2 • [Edit Online](#)

## 語法

```
~ cast-expression
```

## 備註

一補數運算子(`~`)，有時稱為位補數運算子，會產生其運算元的位一補數。也就是說，運算元中是1的每個位元，在結果中都是0。反之，運算元中是0的每個位元，在結果中都是1。一補數運算子的運算元必須是整數類資料類型。

## ~ 的 Operator 關鍵字

C++ 會 `compl` 將指定為的替代拼寫 `~`。在 C 中，替代的拼寫是以宏的形式在 `<iso646.h>` 標頭中提供。在 C++ 中，替代的拼寫是關鍵字；使用 `<iso646.h>` 或 C++ 對等用法 `<ciso646>` 已被取代。在 Microsoft C++ 中，`/permissive- /Za` 需要或編譯器選項才能啟用替代的拼寫。

## 範例

```
// expr_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;
}
```

在這個範例中，指派至 `y` 的新值是不帶正負號的值 0xFFFF 或 0x0000 的1補數。

整數提升會在整數運算元上執行。運算元升級為的類型是結果類型。如需整數提升的詳細資訊，請參閱[標準轉換](#)。

## 另請參閱

[具有一元運算子的運算式](#)

[C++ 內建運算子、優先順序和關聯性](#)

[一元算術運算子](#)

# 指標到成員運算符:`*` 和 `->*`

2020/4/15 • [Edit Online](#)

## 語法

```
expression .* expression
expression ->* expression
```

## 備註

指標到成員運算子 `*` 和 `->*` 返回運算式左側指定物件的特定類成員的值。右邊則必須指定類別的成員。以下範例會示範如何使用這些運算子：

```
// expe_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;
}
```

## 輸出

```
m_func1  
m_func1  
1  
2
```

在上述範例中，成員 `pmfn` 的指標會在叫用成員函式 `m_func1` 時使用。另一個成員 `pmd` 的指標會在存取 `m_num` 成員時使用。

二元運算子 `.*` 會將其第一個運算元 (該運算元必須是類別類型的物件) 與其第二個運算元 (該運算元必須是成員指標類型) 結合。

二進位運算符 `->+` 將其第一個操作數(必須是指向類類型物件的指標)與其第二個操作數(必須是指向成員的指標類型)合併。

在包含 `.*` 運算子的運算式中，第一個運算元必須是第二個運算元中所指定成員指標本身所屬且可存取的類別類型，或是該類別所明確衍生且可存取的類型。

在包含 `->+` 運算符的運算式中，第一個操作數必須是第二個操作數中指定的類型的"指向類類型的指標"類型，或者它必須是明確派生自該類的類型。

## 範例

以下列類別和程式碼片段為例：

```
// 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 \*-> 指向成員的指標運算符的結果是指向成員的指標聲明中指定的類型的物件或函數。因此，在上述範例中，`ADerived.*pmfnFunc1()` 運算式的結果會是傳回 `void` 的函式指標。如果第二個運算元是左值，則這個結果會是左值。

**NOTE**

如果其中一個成員指標運算子的結果是函式，則結果只能做為函式呼叫運算子的運算元使用。

## 另請參閱

[C++ 內建運算子、優先順序和順序關聯性](#)

# 後置遞增和遞減運算子：++ 和 --

2020/11/2 • [Edit Online](#)

## 語法

```
postfix-expression ++
postfix-expression --
```

## 備註

C++ 提供前置和後置遞增及遞減運算子，本節只說明後置遞增和遞減運算子。（如需詳細資訊，請參閱[前置遞增和遞減運算子](#)）。兩者的差異在於，在後置標記法中，運算子會出現在後置運算式之後，而在前置標記法中，運算子會出現在*expression* 之前。下列範例顯示後置遞增運算子：

```
i++;
```

套用後置遞增運算子()的效果 ++ 是運算元的值會增加一個單位的適當類型。同樣地，套用後置遞減運算子()的效果，-- 是運算元的值會減少一個單位的適當類型。

請務必注意，後置遞增或遞減運算式會評估為運算式的值，然後再進行個別運算子的應用。遞增或遞減運算會在運算元評估之後發生。此問題只有在較大運算式的內容中進行後置遞增或遞減運算時發生。

使用後置運算子當做函式的引數時，引數的值在傳遞至函式之前不保證會遞增或遞減。如需詳細資訊，請參閱 C++ 標準中的 1.9.17 一節。

將後置遞增運算子套用至類型物件陣列的指標時，實際上會在 `long` 指標的內部標記法中加入四個。這個行為會導致指標(先前參考陣列的第*n*個元素)參考(*n+1*)個元素。

後置遞增和後置遞減運算子的運算元，必須可修改(不是 `const`)算術或指標類型的左值。結果的型別與後置運算式的類型相同，但不再是左值。

**Visual Studio 2017 15.3 和更新版本 (適用于 `/std: c++17`)**：後置遞增或遞減運算子的運算元不可以是類型 `bool`。

下列程式碼示範後置遞增運算子：

```
// 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;
}
```

不支援在列舉類型上進行後置遞增和後置遞減運算：

```
enum Compass { North, South, East, West };
Compass myCompass;
for( myCompass = North; myCompass != West; myCompass++ ) // Error
```

## 另請參閱

[後置運算式](#)

[C++ 內建運算子、優先順序和順序關聯性](#)

[C 後置遞增和遞減運算子](#)

# 前置遞增和遞減運算子：++ 和 --

2020/11/2 • [Edit Online](#)

## 語法

```
++ unary-expression  
-- unary-expression
```

## 備註

前置遞增運算子(++)會在其運算元中加入一個，而這個遞增的值就是運算式的結果。運算元必須是不是類型的左值 `const`。結果會是與運算元相同類型的左值。

前置遞減運算子(--)類似前置遞增運算子，不同之處在於運算元會減一，而結果為這個遞減的值。

Visual Studio 2017 15.3 和更新版本(適用於[/std: c++17](#))：遞增或遞減運算子的運算元不可以是類型 `bool`。

前置和後置遞增和遞減運算子都會影響其運算元。它們之間的主要差異在於遞增或遞減在運算式評估中發生的順序。(如需詳細資訊，請參閱後置遞增和遞減運算子)。在前置詞形式中，遞增或遞減會在運算式評估中使用值之前進行，因此運算式的值會與運算元的值不同。在後置形式中，遞增或遞減會在運算式評估中使用值之後發生，因此運算式的值會與運算元的值相同。例如，下列程式會列印 "`++i = 6`"：

```
// expre_Increment_and_Decrement_Operators.cpp  
// compile with: /EHsc  
#include <iostream>  
  
using namespace std;  
  
int main() {  
    int i = 5;  
    cout << "++i = " << ++i << endl;  
}
```

整數或浮點類型的運算元會以整數值 1 遞增或遞減。結果的類型與運算元類型相同。指標類型的運算元會以其定址之物件的大小遞增或遞減。遞增指標會指向下一個物件，遞減指標則指向一個物件。

因為遞增和遞減運算子具有副作用，所以在[預處理器宏](#)中使用具有遞增或遞減運算子的運算式可能會有不想要的結果。請思考此範例：

```
// 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 );  
}
```

巨集會展開為：

```
k = ((++i) < (j)) ? (j) : (++i);
```

如果  $i$  大於或等於  $j$ , 或是比  $j$  少 1, 則會遞增兩次。

#### NOTE

在許多情況下, C++ 內嵌函式會比巨集更理想, 因為這類函式會排除副作用 (如這裡所述的副作用), 並且可讓語言執行更完整的類型檢查。

## 另請參閱

[具有一元運算子的運算式](#)

[C++ 內建運算子、優先順序和順序關聯性](#)

[前置遞增和遞減運算子](#)

# 關係運算子：<、>、<= 和>=

2020/11/2 • [Edit Online](#)

## 語法

```
expression < expression
expression > expression
expression <= expression
expression >= expression
```

## 備註

二元關係運算子決定下列關聯性：

- 小於(<)
- 大於(>)
- 小於或等於(<=)
- 大於或等於(>=)

關係運算子具有由左到右的順序關聯性。關係運算子的兩個運算元都必須是算術或指標類型。它們會產生類型的值 `bool`。如果運算式中的關聯性是 `false`, 傳回的值會是 `(0)`; 否則, 傳回的值會是 `true` (`1`)。

## 範例

```
// 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;
}
```

上述範例中的運算式必須以括弧括住，因為資料流程插入運算子(`<<`)的優先順序高於關係運算子。因此，不加括弧的第一個運算式會評估為：

```
(cout << "The true expression 3 > 2 yields: " << 3) < (2 << "\n");
```

[標準轉換](#)中所涵蓋的一般算術轉換適用於算術類型的運算元。

## 比較指標

比較相同類型的兩個物件指標時，結果是取決於程式的位址空間中所指向的物件位置。指標也可以與評估為`0`或類型指標的常數運算式進行比較 `void *`。如果對類型的指標進行指標比較 `void *`，另一個指標會隱含地轉換成類型 `void *`。然後才進行比較。

兩個不同類型的指標無法進行比較，除非：

- 其中一個類型是衍生自另一個類型的類別類型。
- 至少有一個指標明確轉換(cast)為類型 `void *`。（另一個指標會隱含地轉換成類型 `void *` 以進行轉換）。

相同類型且指向相同物件之兩個指標的比較結果一定為相等。如果將物件的兩個非靜態成員指標相互比較，則適用下列規則：

- 如果類別類型不是 `union`，而且這兩個成員未以存取規範分隔，例如 `public`、`protected` 或 `private`，則最後宣告之成員的指標將會比稍早宣告的成員指標還要大。
- 如果兩個成員以存取規範分隔，則結果會是未定義的。
- 如果類別類型為，則 `union` 為中與相同之不同資料成員的指標 `union`。

如果兩個指標指向相同陣列的元素，或指向超出陣列結尾的元素一，則具有較高註標的物件指標比較結果會較高。只有在指標參考相同陣列中的物件或超出陣列結尾的位置一時，才能保證指標比較有效。

## 另請參閱

[具有二元運算子的運算式](#)

[C++ 內建運算子、優先順序和順序關聯性](#)

[C 關聯式和等號比較運算子](#)

# 範圍解析運算子 :

2020/12/10 • [Edit Online](#)

範圍解析運算子 `::` 可用來識別及區分不同範圍內使用的識別碼。如需範圍的詳細資訊，請參閱 [範圍](#)。

## 語法

```
qualified-id :  
    nested-name-specifier *** template * opt unqualified-id
```

```
nested-name-specifier :  
    ::  
    type-name ::  
    namespace-name ::  
    decltype-specifier ::  
    nested-name-specifier identifier ::  
    nested-name-specifier *** template * opt Opt simple-template-id *** :: *
```

```
unqualified-id :  
    identifier  
    operator-function-id  
    conversion-function-id  
    literal-operator-id  
    ~ type-name  
    ~ decltype-specifier  
    template-id
```

## 備註

`identifier` 可以是變數、函式或列舉值。

## 用於 `::` 類別和命名空間

下列範例顯示範圍解析運算子如何與命名空間和類別搭配使用：

```

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;
}

```

不含範圍限定詞的範圍解析運算子是指全域命名空間。

```

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;
}

```

您可以使用範圍解析運算子來識別的成員 `namespace`，或識別在指示詞中 `nominates` 成員命名空間的命名空間 `using`。在下列範例中，您可以使用 `NamespaceC` 來限定 `ClassB`，即使 `ClassB` 是在命名空間中宣告，因為是由指示詞所 `NamespaceB` `NamespaceB` 命名 `NamespaceC` `using`。

```

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;
}

```

您可以使用範圍解析運算子的鏈結。在下列範例中，`NamespaceD::NamespaceD1` 會識別巢狀命名空間 `NamespaceD1`，

而 `NamespaceE::ClassE::ClassE1` 會識別巢狀類別 `ClassE1`。

```
namespace NamespaceD{
    namespace NamespaceD1{
        int x;
    }
}

namespace NamespaceE{
    class ClassE{
        public:
            class ClassE1{
                public:
                    int x;
            };
    };
}

int main() {
    NamespaceD:: NamespaceD1::x = 6;
    NamespaceE::ClassE::ClassE1 e1;
    e1.x = 7 ;
}
```

## 用於 `::` 靜態成員

您必須使用範圍解析運算子來呼叫類別的靜態成員。

```
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();
}
```

## 用於 `::` 範圍列舉

範圍解析運算子也會與範圍列舉 [列舉宣告](#)的值搭配使用，如下列範例所示：

```
enum class EnumA{
    First,
    Second,
    Third
};

int main() {
    EnumA enum_value = EnumA::First;
}
```

## 另請參閱

[C++ 內建運算子、優先順序和關聯性](#)

命名空間

# sizeof 運算子

2020/11/2 • [Edit Online](#)

根據類型的大小，產生其運算元的大小 `char`。

## NOTE

如需運算子的詳細資訊 `sizeof ...`，請參閱[省略號和 variadic 範本](#)。

## 語法

```
sizeof unary-expression  
sizeof ( type-name )
```

## 備註

運算子的結果屬於 `sizeof` 類型，也就是 `size_t` 包含檔案中所定義的整數類資料類型 `<stddef.h>`。這個運算子可以避免在您的程式中指定與電腦相關的資料大小。

的運算元 `sizeof` 可以是下列其中一項：

- 類型名稱。若要搭配 `sizeof` 型別名稱使用，名稱必須以括弧括住。
- 運算式。與運算式搭配使用時，`sizeof` 可以使用或不搭配括弧來指定。並不會評估運算式。

當 `sizeof` 運算子套用至類型的物件時 `char`，它會產生1。當 `sizeof` 運算子套用至陣列時，它會產生該陣列中的總位元組數，而不是陣列識別碼所表示的指標大小。若要取得陣列識別碼所表示的指標大小，請將它當做參數傳遞至使用的函式 `sizeof`。例如：

## 範例

```
#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;  
}
```

## 範例輸出

```
The size of a char is: 1
The length of Hello, world! is: 14
The size of the pointer is 4
```

當 `sizeof` 運算子套用至 `class`、`struct` 或類型時 `union`，結果會是該類型物件中的位元組數目，加上加入以對齊字邊界上成員的任何填補。在加上個別成員的儲存需求之後，其結果不一定會對應計算的大小。[/Zp](#) 編譯器選項和 [pack](#) pragma 會影響成員的對齊界限。

`sizeof` 即使是空的類別，運算子也永遠不會產生0。

`sizeof` 運算子不能與下列運算元搭配使用：

- 函數。(不過，可以套用至函式的 `sizeof` 指標)。
- 位元欄位。
- 未定義的類別。
- 型別 `void`。
- 以動態方式配置的陣列。
- 外部陣列。
- 不完整的類型。
- 以括號括住的不完整類型名稱。

當 `sizeof` 運算子套用至參考時，其結果會與 `sizeof` 已套用至物件本身相同。

如果可變大小陣列是結構的最後一個元素，則 `sizeof` 運算子會傳回不含陣列的結構大小。

`sizeof` 運算子通常用來計算陣列中的元素數目，其使用的運算式格式如下：

```
sizeof array / sizeof array[0]
```

## 另請參閱

[具有一元運算子的運算式](#)

[關鍵字](#)

# 注標運算子 []

2020/11/2 • [Edit Online](#)

## 語法

```
postfix-expression [ expression ]
```

## 備註

後置運算式(也可以是主要運算式)後面接著注標運算子 []，指定陣列索引。

如需 C++/CLI 中 managed 陣列的詳細資訊，請參閱[陣列](#)。

通常，後置運算式所代表的值是指標值，例如陣列識別碼，而 *expression* 則是整數值(包括列舉類型)。不過，在語法上需要的是其中一個指標類型的運算式，另一個則是整數類型。因此，整數值可以在後置運算式位置中，而指標值可以在運算式或注標位置的括弧中。請考慮下列程式碼片段：

```
int nArray[5] = { 0, 1, 2, 3, 4 };
cout << nArray[2] << endl;           // prints "2"
cout << 2[nArray] << endl;          // prints "2"
```

在上述範例中，運算式 `nArray[2]` 與 `2[nArray]` 相同。原因是注標運算式的結果 `e1[e2]` 是由所提供：

```
*((e2) + (e1))
```

運算式所產生的位址不是來自位址 *e1* 的 *e2* 位元組。而是會調整位址，以產生陣列 *e2* 中的下一個物件。例如：

```
double aDb1[2];
```

和的位址 `aDb[0]` `aDb[1]` 是 8 個位元組，也就是類型的物件大小 `double`。這項根據物件類型的調整是由 C++ 語言自動完成，並定義于[加法類運算子](#)中，其中會討論指標類型運算元的加法和減法。

註標運算式也可以擁有多個註標，如下所示：

`運算式2 [ 運算式2 ] [ expression3 ] ..`

註標運算式的關聯是由左至右。最左邊的注標運算式(`運算式1 [ 運算式2 ]`)會先進行評估。*expression1* 和 *expression2* 相加所產生的位址會形成指標運算式，然後 *expression3* 會加入這個指標運算式形成新的指標運算式，依此類推，直到加入最後一個註標運算式為止。\*除非最後一個指標值會定址陣列類型，否則間接運算子()會在評估最後一個下標運算式之後套用。

具有多個註標的運算式會參考多維陣列的元素。所謂的多維陣列是指其中所包含的元素也是一種陣列。例如，三維陣列的第一個元素是具有兩個維度的陣列。下列範例會宣告和初始化一個簡單的二維字元陣列：

```

// 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;
}

```

## 正數和負數註標

陣列的第一個元素是元素 0。C++ 陣列的範圍是從陣列[0] 到陣列[大小-1]。不過，C++ 支援正註標和負註標。負註標必須在陣列界限內，如果不在界限內，則無法預測結果。下列程式碼顯示正的和負的陣列註標：

```

#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
}

```

最後一行中的負數註標可能會產生執行階段錯誤，因為它指向 `int` 記憶體中比陣列來源更低的位址256位置。指標 `midArray` 會初始化為的中間，`intArray` 因此可能(但危險)在其上使用正和負的陣列索引。陣列註標錯誤不會產生編譯時間錯誤，但是會產生無法預期的結果。

註標運算子可以交替。因此，只要註標運算子未多載(請參閱多載運算子)，運算式`陣列[索引]` 和 `索引[陣列]` 就會保證相等。第一種形式是最常用的程式撰寫作法，但兩種都可以運作。

## 另請參閱

[後置運算式](#)

[C++ 內建運算子、優先順序和順序關聯性](#)

[陣列](#)

[一維陣列](#)

[多維陣列](#)

# typeid 運算子

2020/11/2 • [Edit Online](#)

## 語法

```
typeid(type-id)
typeid(expression)
```

## 備註

`typeid` 運算子可讓您在執行時間決定物件的型別。

的結果 `typeid` 是 `const type_info&`。此值是 `type_info` 物件的參考，代表類型識別碼或運算式的類型，視使用何種形式而定 `typeid`。如需詳細資訊，請參閱[Type\\_info 類別](#)。

`typeid` 運算子不適用於 managed 類型(抽象宣告子或實例)。如需取得 [Type](#) 指定類型之的詳細資訊，請參閱[typeid](#)。

運算子會在套用至多型 `typeid` 類別類型的左值時執行執行時間檢查，其中物件的 `true` 類型無法由提供的靜態資訊判斷。這類案例包括：

- 類別的參考
- 參考的指標，使用 `*`
- 一個下標指標(`[ ]`)。(以多型類型的指標來使用注標並不安全)。

如果運算式指向基類型別，但物件實際上是衍生自該基類的型別，`type_info` 則衍生類別的參考就是結果。運算式必須指向多型類型(具有虛擬函式的類別)。否則，結果會是 `type_info` 運算式中所參考之靜態類別的。此外，必須對指標進行取值，讓使用的物件是它所指向的物件。如果沒有取值指標，則結果會是 `type_info` 指標的，而不是它所指向的。例如：

```
// expre_typeid_Operator.cpp
// compile with: /GR /EHsc
#include <iostream>
#include <typeinfo>

class Base {
public:
    virtual void vfunc() {}
};

class Derived : public Base {};

using namespace std;
int main() {
    Derived* pd = new Derived;
    Base* pb = pd;
    cout << typeid( pb ).name() << endl;    //prints "class Base *"
    cout << typeid( *pb ).name() << endl;    //prints "class Derived"
    cout << typeid( pd ).name() << endl;    //prints "class Derived *"
    cout << typeid( *pd ).name() << endl;    //prints "class Derived"
    delete pd;
}
```

如果運算式正在取值指標，且該指標的值為零，則會擲回 `typeid` [bad\\_typeid 例外](#) 狀況。如果指標未指向有效的物件，則會擲回 `_non_rtti_object` 例外狀況。這表示嘗試分析觸發錯誤的 RTTI，是因為物件是不正確。(例如，它是錯誤指標，或程式碼未使用[GR](#)進行編譯)。

如果運算式不是指標，而不是物件之基類的參考，則結果會是 `type_info` 代表運算式靜態類型的參考。運算式的靜態類型會參考在編譯時期已知的運算式類型。評估運算式的靜態類型時，會忽略執行語意。此外，在判斷運算式的靜態類型時，會盡可能忽略參考：

```
// expre_typeid_Operator_2.cpp
#include <typeinfo>

int main()
{
    typeid(int) == typeid(int&); // evaluates to true
}
```

`typeid` 也可以在範本中用來判斷樣板參數的類型：

```
// 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 );
}
```

## 另請參閱

[執行時間類型資訊](#)

[關鍵字](#)

# 一元正和負運算子：+ 和 -

2020/11/2 • [Edit Online](#)

## 語法

```
+ cast-expression  
- cast-expression
```

### + 運算子

一元加號運算子()的結果 + 是其運算元的值。一元加法運算子的運算元必須屬於算術類型。

整數提升會在整數運算元上執行。結果類型會是運算元提升後的類型。因此，運算式 `+ch` (其中 `ch` 的類型為 `char`) 會產生類型，而 `int` 值則會未經修改。如需如何完成升級的詳細資訊，請參閱[標準轉換](#)。

### - 運算子

一元負運算子(-)會產生其運算元的負數。一元負運算子的運算元必須是算術類型。

整數運算元上會執行整數提升，且結果類型是運算元提升後的類型。如需如何執行升級的詳細資訊，請參閱[標準轉換](#)。

#### Microsoft 特定的

不帶正負號數量的一元否定執行方式是  $2^n$  減去運算元的值，其中  $n$  是指定不帶正負號類型之物件的位元數。

#### 結束 Microsoft 專有

## 另請參閱

[具有一元運算子的運算式](#)

[C++ 內建運算子、優先順序和順序關聯性](#)

# 運算式 (C++)

2020/3/25 • [Edit Online](#)

本節將描述 C++ 運算式。運算式是提供下列其中一種或多種用途的運算子和運算元序列：

- 計算運算元的值。
- 指定物件或函式。
- 產生「副作用」(副作用是評估運算式之外的任何動作，例如修改物件的值)。

在 C++ 中，運算子可以多載，而且其意義可以是使用者所定義。然而，其優先順序和接受的運算元數目不能修改。本節將描述語言所提供之未多載的運算子語法和語意。除了運算式的類型之外，還涵蓋下列主題：

- [主要運算式](#)
- [範圍解析運算子](#)
- [後置運算式](#)
- [具有一元運算子的運算式](#)
- [具有二元運算子的運算式](#)
- [條件運算子](#)
- [常數運算式](#)
- [轉型運算子](#)
- [執行時間類型資訊](#)

其他各節中有關運算子的主題：

- [C++ 內建運算子、優先順序和順序關聯性](#)
- [多載運算子](#)
- [typeid \(C++/cli\)](#)

## NOTE

內建類型的運算子無法多載，其行為已預先定義。

## 另請參閱

[C++ 語言參考](#)

# 運算式的類型

2020/3/25 • [Edit Online](#)

C++ 運算式分為幾個類別：

- **主要運算式**。這些運算式是構成所有其他運算式的建置組塊。
- 後置**運算式**。這些是後面接著一個運算子 (例如, 陣列註標或後置遞增運算子) 的主要運算式。
- **以一元運算子組成的運算式**。一元運算子只會在運算式中的一個運算元上作用。
- **以二元運算子組成的運算式**。二元運算子會在運算式中的兩個運算元上作用。
- **具有條件運算子的運算式**。條件運算子是三元運算子，也是 C++ 語言中唯一的三元運算子，它會使用三個運算元。
- **常數運算式**。常數運算式完全是由常數資料所構成。
- **具有明確類型轉換的運算式**。運算式中可以使用明確類型轉換，或稱為「轉換」(Cast)。
- **具有成員指標運算子的運算式**。
- **轉換**。運算式中可以使用類型安全「轉換」(Cast)。
- **執行時間類型資訊**。在程式執行期間判斷物件的類型。

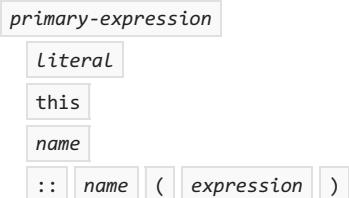
## 另請參閱

[運算式](#)

# 主要運算式

2020/11/2 • [Edit Online](#)

主要運算式為更複雜運算式的建置組塊。它們可能是由範圍解析運算子所限定的常值、名稱和名稱 ( :: )。主要運算式可以具有下列任何形式：



`Literal` 是常數的主要運算式。它的類型取決於其規格形式。如需指定常值的完整資訊，請參閱 [常值](#)。

`this` 關鍵字是類別物件的指標。它可在非靜態成員函式中使用。它會指向已叫用函式之類別的實例。`this` 關鍵字不能用在類別成員函式的主體之外。

指標的類型 `this` 是 `type * const` (, 其中 `type` 是在不會明確修改指標的函式中) 類別名稱 `this`。下列範例顯示成員函式聲明和的類型 `this`：

```
// expre_Primary_Expressions.cpp
// compile with: /LD
class Example
{
public:
    void Func();           // * const this
    void Func() const;     // const * const this
    void Func() volatile; // volatile * const this
};
```

如需有關修改指標類型的詳細資訊 `this`，請參閱 [this 指標](#)。

範圍解析運算子 ( :: ) 後面接著名稱就是主要運算式。這類名稱必須是全域範圍的名稱，而不是成員名稱。運算式的類型取決於名稱的宣告。它是左值 (也就是說，它可以出現在指派運算式的左邊) 如果宣告名稱是左值。範圍解析運算子允許參考全域名稱，即使該名稱在目前範圍中為隱藏狀態。如需如何使用範圍解析運算子的範例，請參閱 [範圍](#)。

以括弧括住的運算式是主要運算式。其類型和值與 unparenthesized 運算式的類型和值相同。如果 unparenthesized 運算式是左值，它就是左值。

主要運算式的範例包括：

```
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
```

這些範例全都被視為各種形式的 名稱，也就是主要運算式：

```
 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
```

## 另請參閱

[運算式的類型](#)

# 省略號和 Variadic 範本

2020/11/2 • [Edit Online](#)

本文說明如何搭配 `...` C++ variadic 範本使用省略號()。省略號在 C 和 C++ 中有許多用途。這些包括函數的變數引數清單。`printf()` C 執行時間程式庫中的函式是其中一個最知名的範例。

Variadic 範本是支援任意數目之引數的類別或函數範本。這項機制對於 C++ 程式庫開發人員特別有用，因為您可以將它套用至類別樣板和函式樣板，進而提供各種型別安全和非一般功能和彈性。

## 語法

Variadic 範本會以兩種方式使用省略號。在參數名稱的左邊，它表示參數套件，而在參數名稱的右邊，它會將參數套件展開成不同的名稱。

以下是 variadic 範本類別定義語法的基本範例：

```
template<typename... Arguments> class classname;
```

對於參數封裝和展開，您可以依您的喜好在省略符號周圍加入空白，如下面這些範例所示：

```
template<typename ...Arguments> class classname;
```

或是這個：

```
template<typename ... Arguments> class classname;
```

請注意，本文使用第一個範例中所示的慣例（省略號會附加至 `typename`）。

在上述範例中，`引數`是參數套件。類別 `classname` 可以接受可變數目的引數，如下列範例所示：

```
template<typename... Arguments> class vtclass;  
  
vtclass< > vtinstance1;  
vtclass<int> vtinstance2;  
vtclass<float, bool> vtinstance3;  
vtclass<long, std::vector<int>, std::string> vtinstance4;
```

藉由使用 variadic 範本類別定義，您也可以至少需要一個參數：

```
template <typename First, typename... Rest> class classname;
```

以下是 variadic 範本函數語法的基本範例：

```
template <typename... Arguments> returntype functionname(Arguments... args);
```

然後會展開 `引數`參數套件以供使用，如下一節所示：瞭解 variadic 範本。

其他形式的 variadic 範本函式語法都可行，包括（但不限於）下列範例：

```
template <typename... Arguments> returntype functionname(Arguments&... args);
template <typename... Arguments> returntype functionname(Arguments&&... args);
template <typename... Arguments> returntype functionname(Arguments*... args);
```

也可以 `const` 使用類似的規範：

```
template <typename... Arguments> returntype functionname(const Arguments&... args);
```

如同 variadic 範本類別定義，您可以建立需要至少一個參數的函式：

```
template <typename First, typename... Rest> returntype functionname(const First& first, const Rest&... args);
```

Variadic 範本會使用 `sizeof...()` 運算子（與較舊的 `sizeof()` 運算子無關）：

```
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...);
}
```

## 進一步了解省略符號位置

在過去，本文說明了定義參數封裝和展開的省略符號位置，「在參數名稱左邊的它表示參數封裝，在參數名稱右邊，它展開參數封裝至不同的名稱」。這在技術上是對的，但是在轉譯程式碼上可能會造成混淆。考量：

- 在範本-參數清單() 中，導入了 `template <parameter-list> typename...` 範本參數套件。
- 在參數宣告子句(`func(parameter-list)`) 中，「頂層」省略號會引進函式參數套件，而省略號位置則很重要：

```
// 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);
```

- 若省略符號在參數名稱之後顯示，您有參數封裝展開。

## 範例

有一個說明 variadic 範本函式機制的好方法，就是使用它來重新撰寫的部分功能 `printf`：

```
#include <iostream>

using namespace std;

void print() {
    cout << endl;
}

template <typename T> void print(const T& t) {
    cout << t << endl;
}

template <typename First, typename... Rest> void print(const First& first, const Rest&... rest) {
    cout << first << ", ";
    print(rest...); // recursive call using pack expansion syntax
}

int main()
{
    print(); // calls first overload, outputting only a newline
    print(1); // calls second overload

    // these call the third overload, the variadic template,
    // which uses recursion as needed.
    print(10, 20);
    print(100, 200, 300);
    print("first", 2, "third", 3.14159);
}
```

## 輸出

```
1
10, 20
100, 200, 300
first, 2, third, 3.14159
```

### NOTE

大部分納入 variadic 範本函式的整合都會使用某種形式的遞迴，但與傳統的遞迴有些微不同。傳統遞迴牽涉到使用相同簽章呼叫本身的函式。（它可能會多載或樣板化，但每次都會選擇相同的簽章。）Variadic 遞迴牽涉到呼叫 Variadic 函式樣板，方法是使用不同（幾乎一律遞減）的引數數目，因此每次都會將不同的簽章標記為。仍然需要「基底案例」，但遞迴的本質並不相同。

# 後置運算式

2020/11/2 • [Edit Online](#)

後置運算式包含主要運算式，或後置運算子後面接著主要運算式的運算式。下表列出後置運算子。

## 後置運算子

運算子	結果
注標運算子	[ ]
函式呼叫運算子	( )
明確類型轉換運算子	類型名稱()
成員存取運算子	. 或->
後置遞增運算子	++
後置遞減運算子	--

下列語法描述可能的後置陳述式：

```
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 )
```

以上的後置運算式可能是主要運算式或另一個後置運算式。後置運算式由左至右分組，因此可讓運算式鏈結在一起，如下所示：

```
func(1)->GetValue()++
```

在上述運算式中，`func` 是一個主要運算式，`func(1)` 它是一個函式後置運算式，`func(1)->GetValue` 是指定類別成員的後置運算式，`func(1)->GetValue()` 是另一個函式後置運算式，而整個運算式是後置運算式，會遞增 `GetValue` 的傳回值。整個運算式的意義是傳遞 1 做為呼叫 `func` 的引數，並取得類別的指標做為傳回值。然後 `GetValue()` 在該類別上呼叫，然後遞增傳回的值。

以上列出的運算式為指派運算式，表示這些運算式的結果必須是右值。

## 後置運算式格式

```
simple-type-name ( expression-list )
```

表示建構函式的引動過程。如果 `simple-type-name` 是一個基本類型，則運算式清單必須是單一運算式，而這個運算式表示將運算式的值轉型為基本類型。此種轉型運算式會模擬建構函式。由於這個格式允許使用相同的語法建構基本類型和類別，因此該格式在定義樣板類別時會特別有用。

`Cast` 關鍵字是、或的其中一個 `dynamic_cast` `static_cast` `reinterpret_cast`。如需詳細資訊 [dynamic\\_cast](#)，請參閱、[static\\_cast](#) 和 [reinterpret\\_cast](#)。

`typeid` 運算子會被視為後置運算式。請參閱 [typeid 運算子](#)。

## 型式和實質引數

呼叫程式會利用「實質引數」將資訊傳遞至所呼叫的函式。所呼叫的函式會使用對應的「型式引數」存取資訊。

呼叫函式時，會執行下列工作：

- 所有實質引數(呼叫端所提供的引數)都會進行評估。這些引數並不需要遵循任何隱含的評估順序，但是會在所有引數都經過評估且所有副作用都已完成之後，才進入函式。
- 每個型式引數都會使用它在運算式清單中的對應實質引數初始化(型式引數是在函式標頭中宣告並在函式主體中使用的引數)。轉換的完成方式就如同初始化一樣，在將實際引數轉換成正確的型別時，也會執行標準和使用者定義的轉換。所執行的初始化將以下列程式碼提供概念上的說明：

```
void Func( int i ); // Function prototype
...
Func( 7 );           // Execute function call
```

呼叫之前的概念初始化為：

```
int Temp_i = 7;
Func( Temp_i );
```

請注意，初始化的執行方式就如同使用等號語法，而不是括號語法。將值傳遞至函式之前，會先製作 `i` 的複本(如需詳細資訊，請參閱 [初始化運算式和轉換](#))。

因此，如果函式原型(宣告)呼叫類型的引數 `long`，而且呼叫程式提供類型的實質引數 `int`，則會使用類型的標準類型轉換來升級實際的引數 `long`(請參閱 [標準轉換](#))。

提供沒有轉換成型式引數類型之標準或使用者定義轉換的實質引數是不正確的做法。

對於類別類型的實質引數，型式引數會藉由呼叫類別的建構函式進行初始化。(如需這些特殊類別成員函式的詳細資訊，請參閱 [函式](#))。

- 函式呼叫將會執行。

下列程式片段將示範函式呼叫：

```
// expt_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 )
{
}
```

`func` 從 `main` 呼叫時，會使用 `param1` 的值來初始化正式參數 `i`(`i` 會轉換成類型 `long`，以對應至使用標準轉換的正確類型)，而正式參數會使用的 `param2` 值初始化 `j`(`j` 會使用標準轉換來轉換為類型 `double`)。

## 引數類型的處理方式

宣告為類型的型式引數無法在函式 `const` 主體中變更。函式可以變更任何不屬於類型的引數 `const`。不過，這項變更對函式而言是本機的，而且不會影響實際引數的值，除非實際的引數是不屬於類型之物件的參考 `const`。

下列函式將說明一些這類概念：

```
// expe_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;
}
```

## 省略號和預設引數

如需傳遞可變引數數目的詳細資訊只要使用下列兩種方法的其中一種，函式就可以宣告為接受比函式定義中所指定數目少的引數：省略符號 (`...`) 或預設引數。

省略號表示可能需要引數，但宣告中未指定數位和類型。一般來說，這並不是理想的 C++ 程式設計做法，因為它會失去其中一項 C++ 的優點：類型安全。不同的轉換會套用至以省略號宣告的函式，而不是已知型式和實際引數類型的函式：

- 如果實際引數的類型為 `float`，則會在 `double` 函式呼叫之前提升為類型。
- 任何 `signed char` or `unsigned char`、`signed short` or `unsigned short`、列舉類型或位欄位都會轉換成 `signed int` 或 `unsigned int` 使用整數提升。
- 任何類別類型的引數都會以傳值的方式做為資料結構傳遞，而複本會以二進位檔複製的方式建立，而不會以叫用類別之複製建構函式（如果有的話）的方式建立。

如果使用省略號，必須在引數清單中的最後一個宣告。如需傳遞可變引數數目的詳細資訊，請參閱《執行時間程式庫參考》中 [va\\_arg](#)、[va\\_start](#) 和 [va\\_list](#) 的討論。

如需 CLR 程式設計中預設引數的詳細資訊，請參閱 [Variable 引數清單 \(...\) \(c + +/cli\)](#)。

預設引數可讓您指定函式呼叫中未提供值時，引數應該假設的值。下列程式碼片段將示範預設引數的運作方式。如需有關指定預設引數之限制的詳細資訊，請參閱 [預設引數](#)。

```
// 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;
}
```

上述程式會宣告接受兩個引數的函式 `print`。不過，第二個引數（結束字元）具有預設值 `"\n"`。在中 `main`，的前兩個呼叫 `print` 允許預設的第二個引數提供新的行來結束列印的字串。第三個呼叫會為第二個引數指定明確的值。程式的輸出為

```
hello,
world!
good morning, sunshine.
```

## 另請參閱

[運算式的類型](#)

# 具有一元運算子的運算式

2020/11/2 • [Edit Online](#)

一元運算子只會在運算式中的一個運算元上作用。一元運算子如下：

- 間接取值運算子 (\*)
- Address 運算子 (&)
- 一元加號運算子 (+)
- 一元負運算子 (-)
- 邏輯負運算子 (!)
- 一補數運算子 (~)
- 前置遞增運算子 (++)
- 前置遞減運算子 (--)
- Cast 運算子 ()
- `sizeof` 操作
- `_uuidof` 操作
- `alignof` 操作
- `new` 操作
- `delete` 操作

這些運算子具有由右到左的順序關聯性。一元運算式的語法通常會置於後置或主要運算式的前方。

以下是一元運算式的可能形式。

- *postfix-expression*
- `++ unary-expression`
- `-- unary-expression`
- 一元運算子 *cast 運算式*
- `* sizeof ***一元運算式`
- `sizeof( 類型-名稱 )`
- `decltype( expression )`
- 配置運算式
- 解除配置-運算式

任何後置運算式都會被視為一元運算式，而且因為任何主要運算式都會被視為後置運算式，所以任何主要運算式也會視為一元運算式。如需詳細資訊，請參閱後置運算式和主要運算式。

一元運算子包含下列一個或多個符號：`*` `&` `+` `-` `!` `~`

*Cast* 運算式是具有選擇性轉換的一元運算式，可變更型別。如需詳細資訊，請參閱[轉換運算子 : \(\)](#)。

運算式可以是任何運算式。如需詳細資訊，請參閱[運算式](#)。

配置運算式會參考 `new` 運算子。解除配置運算式會參考 `delete` 運算子。如需詳細資訊，請參閱本主題稍早的連結。

## 另請參閱

[運算式的類型](#)

# 具有二元運算子的運算式

2020/4/15 • [Edit Online](#)

二元運算子會在運算式中的兩個運算元上作用。二元運算子為：

- **乘法類運算子**

- 乘 (\*)
- 除 (/)
- 模數 (%)

- **加法類運算子**

- 加 (+)
- 減 (-)

- **移位運算子**

- 右移(>>)
- 左班次 (<<)

- **關聯和相等運算子**

- 小於 (<)
- 大於 (>)
- 小於或等於 (\*)<
- 大於或等於 (>=)
- 等於 (==)
- 不等於 (!=)

- **位元運算子**

- 位元與 (&)
- 位專用 OR (\*)
- 位式包含 OR ()

- **邏輯運算子**

- 邏輯和 (&&)
- 邏輯 OR (||)

- **指派運算子**

- 指派 (=)
- 加法指派 (+=)
- 減法分配 (-\*)

- 乘法指派 ( $*=$ )
  - 除法指派 ( $/=$ )
  - 模數指派 ( $\% =$ )
  - 左班次分配 ( $<<*$ )
  - 右移位分配 ( $>>*$ )
  - 位和賦值 ( $\&*$ )
  - 位元互斥 OR 指派 ( $\wedge =$ )
  - 位式包含或賦值 ( $|*$ )
- 逗號運算子 (,)

## 另請參閱

[運算式的類型](#)

# C++ 常數運算式

2020/11/2 • [Edit Online](#)

常數值是一個不會變更的值。C++ 提供兩個關鍵字，以讓您表示不要修改物件的意圖，以及施行該意圖。

C++ 需要常數運算式 (也就是評估為常數的運算式) 才能宣告下列各項：

- 陣列界限
- case 陳述式中的選取器
- 位元欄位長度規格
- 列舉初始設定式

常數運算式中合法的運算元包括：

- 常值
- 列舉常數
- 宣告為 const 並使用常數運算式初始化的值
- `sizeof` 運算式

非整數常數必須轉換成整數類資料類型 (明確或隱含)，在常數運算式中才會是合法的。因此，下列程式碼是合法的：

```
const double Size = 11.0;
char chArray[(int)Size];
```

在常數運算式中明確轉換成整數類資料類型是合法的；所有其他類型和衍生類型都不合法，但做為運算子的運算元使用時除外 `sizeof`。

逗號運算子和指派運算子不能在常數運算式中使用。

## 另請參閱

[運算式的類型](#)

# 運算式的語意

2020/11/2 • [Edit Online](#)

運算式會根據其運算子的優先順序和群組進行評估。(在[詞法慣例](#)中，[運算子優先順序和關聯性](#)會顯示 C++ 運算子強加于運算式的關聯性)。

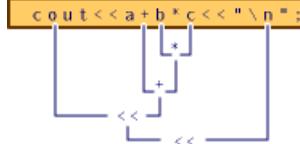
## 評估順序

請思考此範例：

```
// Order_of_Evaluation.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;
int main()
{
    int a = 2, b = 4, c = 9;

    cout << a + b * c << "\n";
    cout << a + (b * c) << "\n";
    cout << (a + b) * c << "\n";
}
```

38  
38  
54

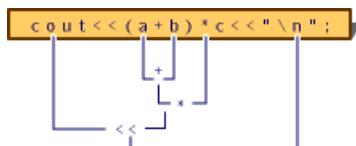


運算式評估順序

上圖中顯示運算式評估的順序可用來判斷運算子的優先順序和關聯性：

- 在此運算式中，乘法 (\*) 具有最高優先順序，因此會先評估子運算式  $b * c$ 。
- 下一個最高優先順序的項目是加法 (+)，因此會將  $a$  與  $b$  和  $c$  的乘積相加。
- 左移 (<<) 在運算式中的優先順序最低，但出現兩次。由於左移運算子是由左至右來分組，因此會先評估左方的子運算式，再評估右邊的子運算式。

在子運算式的群組中使用括號時，會改變優先順序以及運算式的評估順序，如下圖所示。



運算式-具有括弧的評估順序

運算式 (如上圖) 會單純地評估其副作用，在此例中，會將資訊傳送到標準輸出裝置。

## 運算式中的標記法

指定運算元時, C++ 語言會指定特定的相容性。下表顯示需要類型為類型之運算元的運算子可接受的運算元類型。

## 運算子可接受的運算元類型

運算子	運算元類型
type	* const ***類型 * volatile ***類型 型& * const ***類型& * volatile ***類型& volatile const 類型 volatile const 類型&
類型*	類型* * const ***類型* * volatile ***類型* volatile const 類型*
* const ***類型	type * const ***類型 * const ***類型&
* volatile ***類型	type * volatile ***類型 * volatile ***類型&

由於上述規則一律可以搭配使用，因此可以在預期出現指標的位置提供 volatile 物件的 const 指標。

## 模稜兩可的運算式

某些運算式的意義模稜兩可。若物件的值在同一個運算式中修改過一次以上，最容易出現這些運算式。這些運算式依賴特定評估順序，而語言並未定義評估順序。請考慮下列範例：

```
int i = 7;  
  
func( i, ++i );
```

C++ 語言不保證函式呼叫引數的評估順序。因此，在前述範例中，`func` 可能收到的參數值包括 7 和 8 或 8 和 8，需視是從左到右或從右到左評估參數而定。

## C++ 序列點(Microsoft 專有)

運算式只能在連續「序列點」之間修改物件的值一次。

C++ 語言定義目前未指定序列點。針對包含 C 運算子及未包含多載運算子的任何運算式，Microsoft C++ 使用與 ANSI C 相同的序列點。當運算子經過多載時，其語意會從運算子序列變更為函式呼叫序列。Microsoft C++ 使用下列序列點：

- 邏輯 AND 運算子的左運算元(&&)。繼續之前會完整評估邏輯 AND 運算子的左運算元，並且完成所有副作用。不保證會評估邏輯 AND 運算子的右運算元。
- 邏輯 OR 運算子的左運算元(||)。繼續之前會完整評估邏輯 OR 運算子的左運算元，並且完成所有副作用。不保證會評估邏輯 OR 運算子的右運算元。
- 逗號運算子的左運算元。繼續之前會完整評估逗號運算子的左運算元，並且完成所有副作用。逗號運算子的兩個運算元會一律進行評估。

- 函式呼叫運算子。在進入函式之前會先評估函式呼叫運算式和函式的所有引數(包括預設引數)，並完成所有副作用。其中並未指定引數或函式呼叫運算式之間的評估順序。
- 條件運算子的第一個運算元。繼續之前會完整評估條件運算子的第一運算元，並且完成所有副作用。
- 完整初始化運算式的結尾，例如在宣告陳述式中初始化的結尾。
- 運算陳述式中的運算式。運算式陳述式包含選擇性運算式且後面加上分號();。會針對運算式的副作用進行完整的評估。
- 選取範圍(if 或 switch)陳述式中的控制運算式。會完整評估運算式，且其所有副作用會在執行與選取範圍相關的程式碼執行之前完成。
- while 或 do 陳述式的控制運算式。會完整評估運算式，且其所有副作用會在執行 while 或 do 迴圈的下次反覆項目中的陳述式之前完成。
- for 陳述式的三個運算式中的每一個。會完整評估每個運算式，且其所有副作用會在移至下一個運算式之前完成。
- return 陳述式中的運算式。會完整評估運算式，且其所有副作用會在控制回到呼叫函式之前完成。

## 另請參閱

### 運算式

# 轉型

2020/3/25 • [Edit Online](#)

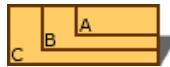
C++ 語言會假設，如果類別是從包含虛擬函式的基底類別衍生，則該基底類別類型的指標可以用來呼叫衍生類別物件內虛擬函式的實作。包含虛擬函式的類別有時稱為「多型類別」(Polymorphic Class)。

由於衍生類別完全包含本身從中衍生之所有基底類別的定義，因此可以放心在類別階層中將指標向上轉型至任何這類基底類別。假設有基底類別的指標，在階層中向下轉型指標可能是安全的。如果所指向的物件實際上是從基底類別衍生的類型，則是安全的。在這種情況下，實際物件會視為「完整物件」(Complete Object)。而基底類別的指標會指向完整物件的「子物件」(Subobject)。例如，以下圖顯示的類別階層為例。



類別階層

如下圖所示，C 類型的物件可以視覺化。



具有 B 和 A 兩個子物件的類別 C

假設有C 類別的執行個體，則會有B 子物件和A 子物件。C 的執行個體 (包括A 和B 子物件) 就是「完整物件」。

若使用執行階段類型資訊，就可以檢查指標是否確實指向完整物件，並且可以安全地轉型為指向其階層中的另一個物件。[Dynamic\\_cast](#)運算子可以用來進行這些類型的轉換。它也會執行安全作業所必要的執行階段檢查。

若要轉換非多型類型，您可以使用[static\\_cast](#)運算子(本主題將說明靜態和動態轉換轉換之間的差異，以及何時適合使用)。

本節包含下列主題：

- [轉型運算子](#)
- [執行時間類型資訊](#)

## 另請參閱

[運算式](#)

# 轉型運算子

2020/11/2 • [Edit Online](#)

C++ 語言有幾個特有的轉型運算子。這些運算子的目的在於移除舊式 C 語言轉型固有的模稜兩可和危險。這些運算子如下所列：

- `dynamic_cast`用於多型類型的轉換。
- `static_cast`用於非多型類型的轉換。
- `const_cast`用來移除 `const`、`volatile` 和 `__unaligned` 屬性。
- `reinterpret_cast`用於位的簡單 reinterpretation。
- `safe_cast`在 c + +/CLI 中用來產生可驗證的 MSIL。

`const_cast` 請使用和 `reinterpret_cast` 做為最後的手段，因為這些運算子會與舊的樣式轉換呈現相同的危險。然而，為了完全取代舊類型轉換，這些運算子仍有其必要。

## 另請參閱

[轉型](#)

# dynamic\_cast 運算子

2020/11/2 • [Edit Online](#)

將運算元轉換 `expression` 為類型的物件 `type-id` 。

## 語法

```
dynamic_cast < type-id > ( expression )
```

## 備註

`type-id` 必須是指標，或參考先前定義的類別類型或「void 的指標」。如果 `type-id` 是指標，則 `expression` 類型必須是指標；或如果 `type-id` 是參考，則為左值 (l-value)。

如需靜態和動態轉型轉換之間的差異，以及何時適合使用各項的說明，請參閱[static\\_cast](#)。

Managed 程式碼中的行為有兩個重大變更 `dynamic_cast`：

- `dynamic_cast` 對已裝箱列舉的基礎類型指標，會在執行時間失敗，傳回0而不是轉換的指標。
- `dynamic_cast` 當 `type-id` 是實數值型別的內部指標，且在執行時間發生轉換失敗時，將不會再擲回例外狀況。轉換現在會傳回0指標值，而不是擲回。

如果 `type-id` 是明確可存取的直接或間接基類的指標，則 `expression` 類型之唯一子物件的指標 `type-id` 就是結果。例如：

```
// 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
}
```

這種類型的轉換稱為「向上連結」，因為它會將指標向上移動到類別階層，從衍生的類別到衍生自的類別。向上轉換是隱含的轉換。

如果 `type-id` 是 `void *`，就會進行執行時間檢查來判斷的實際類型 `expression`。結果是指向的完整物件指標 `expression`。例如：

```

// 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
}

```

如果不 `type-id` 是 `void *`, 則會進行執行時間檢查, 以查看所指向的物件是否 `expression` 可以轉換成所指向的類型 `type-id`。

如果的類型 `expression` 是類型的基類 `type-id`, 則會進行執行時間檢查, 以查看是否 `expression` 實際指向類型的完整物件 `type-id`。如果這是 `true`, 則結果會是類型之完整物件的指標 `type-id`。例如:

```

// 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
}

```

這種類型的轉換稱為「轉換」, 因為它會將指標向下移動類別階層, 從指定的類別到衍生自它的類別。

在多重繼承的情況下, 會引進不明確的可能性。請考慮下圖所示的類別階層。

針對 CLR 類型, `dynamic_cast` 如果轉換可以隱含執行, 則會產生無 `op` 的, 或是 `isinst` 執行動態檢查並在 `nullptr` 轉換失敗時傳回的 MSIL 指令。

下列範例會使用 `dynamic_cast` 來判斷類別是否為特定類型的實例:

```

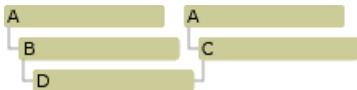
// dynamic_cast_clr.cpp
// compile with: /clr
using namespace System;

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);
}

```



顯示多重繼承的類別階層

類型物件的指標 `D` 可以安全地轉換成 `B` 或 `C`。不過，如果 `D` 轉換成指向 `A` 物件，則 `A` 會產生哪個實例？這會導致不明確的轉換錯誤。若要解決這個問題，您可以執行兩個明確的轉換。例如：

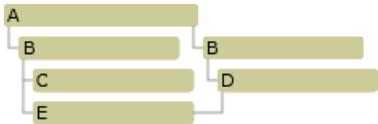
```

// 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
}

```

當您使用虛擬基類時，可以引進進一步的歧義。請考慮下圖所示的類別階層。



顯示虛擬基底類別的類別階層

在此階層中，`A` 是虛擬基類。假設類別的實例 `E` 和子物件的指標 `A`，的 `dynamic_cast` 指標 `B` 將會因為不明確而失敗。您必須先將轉換回完整的 `E` 物件，然後以明確的方式來備份階層，以達到正確的 `B` 物件。

請考慮下圖所示的類別階層。



顯示重複基底類別的類別階層

假設類型的物件 `E` 和子物件的指標 `D`，從子物件導覽 `D` 至最左邊的子物件，則 `A` 可以進行三項轉換。您可以從指標執行 `dynamic_cast` 轉換 `D` `E`，然後從轉換 (`dynamic_cast` 或隱含轉換)到 `E` `B`，最後是從到的隱含轉換 `B` `A`。例如：

```

// 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` 運算子也可以用來執行「交叉轉型」。使用相同的類別階層，只要完整的物件屬於類型，就可以將指標(例如，從子物件轉換成子物件) `B` `D` `E`。

考慮交叉轉型，實際上可以 `D` 在兩個步驟中，從的指標轉換成最左邊子物件的指標。您可以執行從到的交互 `D` 轉換 `B`，然後從隱含轉換 `B` 成 `A`。例如：

```
// 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
}
```

Null 指標值會由轉換成目的地類型的 null 指標值 `dynamic_cast`。

當您使用時，`dynamic_cast < type-id > ( expression )` 如果 `expression` 無法安全地轉換成類型 `type-id`，執行時間檢查會導致轉換失敗。例如：

```
// 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
}
```

轉換成指標類型的失敗值為 null 指標。轉換成參考型別的失敗會擲回[Bad\\_cast 例外](#)狀況。如果 `expression` 未指向或參考有效的物件，則會擲回 `_non_rtti_object` 例外狀況。

如需例外狀況的說明，請參閱[typeid \\_non\\_rtti\\_object](#)。

## 範例

下列範例會建立基類(結構 A)指標，指向物件(結構 C)。這也是虛擬函式的事實，可啟用執行時間多型。

此範例也會呼叫階層中的非虛擬函式。

```

// dynamic_cast_8.cpp
// compile with: /GR /EHsc
#include <stdio.h>
#include <iostream>

struct A {
    virtual void test() {
        printf_s("in A\n");
    }
};

struct B : A {
    virtual void test() {
        printf_s("in B\n");
    }

    void test2() {
        printf_s("test2 in B\n");
    }
};

struct C : B {
    virtual void test() {
        printf_s("in C\n");
    }

    void test2() {
        printf_s("test2 in C\n");
    }
};

void Globaltest(A& a) {
    try {
        C &c = dynamic_cast<C&>(a);
        printf_s("in GlobalTest\n");
    }
    catch(std::bad_cast) {
        printf_s("Can't cast to C\n");
    }
}

int main() {
    A *pa = new C;
    A *pa2 = new B;

    pa->test();

    B * pb = dynamic_cast<B *>(pa);
    if (pb)
        pb->test2();

    C * pc = dynamic_cast<C *>(pa2);
    if (pc)
        pc->test2();

    C ConStack;
    Globaltest(ConStack);

    // will fail because B knows nothing about C
    B BonStack;
    Globaltest(BonStack);
}

```

```
in C
test2 in B
in GlobalTest
Can't cast to C
```

## 另請參閱

[轉型運算子](#)

[關鍵字](#)

# bad\_cast 例外狀況

2020/11/2 • [Edit Online](#)

運算子會擲回bad\_cast例外狀況, `dynamic_cast` 這是轉換成參考型別失敗的結果。

## 語法

```
catch (bad_cast)
    statement
```

## 備註

Bad\_cast的介面為：

```
class bad_cast : public exception
```

下列程式碼包含失敗 `dynamic_cast` 的範例, 它會擲回bad\_cast例外狀況。

```
// expte_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();
    }
}
```

會擲回例外狀況, 因為轉換的物件(圖形)不是從指定的轉換類型(圓形)衍生而來。若要避免例外狀況, 請將這些宣告加入至 `main` :

```
Circle circle_instance;
Circle& ref_circle = circle_instance;
```

然後在區塊中反轉轉換的意義 `try` , 如下所示:

```
Shape& ref_shape = dynamic_cast<Shape&>(ref_circle);
```

## 成員

### 建構函式

██████████	██████████
bad_cast	bad_cast 類型物件的建構函式。

### 函式

██████████	██████████
我	TBD

### 操作員

██████████	██████████
operator =	指派運算子，可將一個 bad_cast 物件指派給另一個。

## bad\_cast

bad\_cast 類型物件的建構函式。

```
bad_cast(const char * _Message = "bad cast");
bad_cast(const bad_cast &);
```

## operator =

指派運算子，可將一個 bad\_cast 物件指派給另一個。

```
bad_cast& operator=(const bad_cast&) noexcept;
```

## 我

```
const char* what() const noexcept override;
```

## 另請參閱

[dynamic\\_cast 運算子](#)

[字](#)

[適用於例外狀況和錯誤處理的新式 C++ 最佳作法](#)

# static\_cast 運算子

2020/11/2 • [Edit Online](#)

根據運算式中出現的類型，將運算式轉換為類型識別碼的類型。

## 語法

```
static_cast <type-id> ( expression )
```

## 備註

標準 C++ 不會利用執行階段類型檢查來確認轉換是否安全。在 C++/CX 中會執行編譯時期和執行階段檢查。如需詳細資訊，請參閱 [轉型](#) 中定義的介面的私用 C++ 專屬實作。

`static_cast` 運算子可以用於作業，例如將基類的指標轉換為衍生類別的指標。這類轉換不一定一直都是安全的。

在一般情況 `static_cast` 下，當您想要將數值資料類型（例如列舉的列舉）轉換成浮點數或整數時，您會使用，而且您一定會有轉換所涉及的資料類型。`static_cast` 轉換與轉換並不安全 `dynamic_cast`，因為不 `static_cast` 會進行任何執行時間類型檢查，而是 `dynamic_cast` 會執行。不 `dynamic_cast` 明確指標的會失敗，而會傳回，`static_cast` 如同沒有任何錯誤；這可能是危險的。雖然 `dynamic_cast` 轉換更為安全，但 `dynamic_cast` 僅適用於指標或參考，而執行時間類型檢查則是額外負荷。如需詳細資訊，請參閱 [Dynamic\\_cast 運算子](#)。

在下列範例中，`D* pd2 = static_cast<D*>(pb);` 這一行並不安全，因為 `D` 可能會包含不在 `B` 中的欄位和方法。不過，`B* pb2 = static_cast<B*>(pd);` 這一行是安全的轉換，因為 `D` 一律都會包含所有的 `B`。

```
// static_cast_Operator.cpp
// compile with: /LD
class B {};

class D : public B {};

void f(B* pb, D* pd) {
    D* pd2 = static_cast<D*>(pb);    // Not safe, D can have fields
                                         // and methods that are not in B.

    B* pb2 = static_cast<B*>(pd);    // Safe conversion, D always
                                         // contains all of B.
}
```

相較于 `dynamic_cast`，轉換時不會進行任何執行時間檢查 `static_cast` `pb`。`pb` 所指向的物件可能不是 `D` 類型的物件，在此情況下，使用 `*pd2` 可能會得不償失。例如，呼叫屬於 `D` 類別但不屬於 `B` 類別的成員函式時，可能會造成存取違規。

`dynamic_cast` 和 `static_cast` 運算子會將指標移至整個類別階層。不過，`static_cast` 只依賴 cast 語句中提供的資訊，因此可能不安全。例如：

```
// 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` 真的指向 `D` 類型的物件，則 `pd1` 和 `pd2` 會取得相同的值。如果 `pb == 0`，它們也會得到相同的值。

如果 `pb` 指向類型的物件 `B`，而不是指向完整的 `D` 類別，則 `dynamic_cast` 會知道足以傳回零。不過，`static_cast` 會依賴程式設計人員的判斷提示，`pb` 指向類型的物件 `D`，並且只會傳回該物件的指標 `D`。

因此，`static_cast` 可以執行隱含轉換的反向動作，在此情況下，結果會是未定義的。它留給程式設計人員確認轉換的結果 `static_cast` 是安全的。

此行為也適用於類別類型以外的類型。例如，`static_cast` 可以用來從 `int` 轉換成 `char`。不過，所產生的 `char` 位可能不足以容納整個 `int` 值。同樣地，它會留給程式設計人員確認轉換的結果 `static_cast` 是否安全。

`static_cast` 運算子也可以用來執行任何隱含轉換，包括標準轉換和使用者定義的轉換。例如：

```
// 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` 運算子可以將整數值明確轉換為列舉型別。如果整數類型的值不在列舉值的範圍內，所產生的列舉值就是未定義的值。

`static_cast` 運算子會將 `null` 指標值轉換為目的地類型的 `null` 指標值。

運算子可以明確地將任何運算式轉換成 `void` 類型 `static_cast`。目的地 `void` 型別可以選擇性地包含 `const`、`volatile` 或 `_unaligned` 屬性。

`static_cast` 運算子無法轉換掉 `const`、`volatile` 或 `_unaligned` 屬性。如需移除這些屬性的詳細資訊，請參閱[Const\\_cast 運算子](#)。

C++/CLI：由於在重新配置的垃圾收集行程之上執行未檢查轉型的風險，因此，使用 `static_cast` 應該只在效能關鍵的程式碼中，才能正常運作。如果您必須 `static_cast` 在發行模式中使用，請將它取代為您的偵錯工具組建中的[safe\\_cast](#)，以確保成功。

## 另請參閱

[轉型運算子](#)

關鍵字

# const\_cast 運算子

2020/11/2 • [Edit Online](#)

`const` `volatile` `__unaligned` 從類別中移除、和屬性。

## 語法

```
const_cast <type-id> (expression)
```

## 備註

任何物件類型的指標或資料成員的指標可以明確地轉換成與 `const`、和限定詞相同的類型 `volatile` `__unaligned`。對於指標和參考，其結果會參考原始物件。對於資料成員的指標，則結果會參考與資料成員的原始（未轉型）指標相同的成員。根據所參考物件的類型，透過產生的指標、參考或資料成員的指標進行寫入作業，可能會產生未定義的行為。

您不能使用 `const_cast` 運算子直接覆寫常數變數的常數狀態。

`const_cast` 運算子會將 null 指標值轉換為目的地類型的 null 指標值。

## 範例

```
// 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();
}
```

在包含的行上 `const_cast`，指標的資料類型 `this` 是 `const CCTest *`。`const_cast` 運算子會將指標的資料類型變更 `this` 為 `CCTest *`，以允許修改該成員 `number`。轉換只會在其出現之陳述式的其餘部分中持續進行。

## 另請參閱

[轉型運算子](#)

[關鍵字](#)

# reinterpret\_cast 運算子

2020/11/2 • [Edit Online](#)

可將所有指標轉換成任何其他指標類型。也可將任何整數類資料類型轉換成任何指標類型 (反之亦然)。

## 語法

```
reinterpret_cast < type-id > ( expression )
```

## 備註

誤用運算子很 `reinterpret_cast` 容易就不安全。除非所需的轉換本質屬於低階轉換，否則您應使用其他任一轉型運算子。

`reinterpret_cast` 運算子可用於的轉換 `char*`，例如 `to int*` 或 `One_class* to Unrelated_class*`，它們原本就不安全。

的結果 `reinterpret_cast` 不能安全地用於轉換回其原始類型以外的任何專案。其他用途的最佳情況是不可攜。

`reinterpret_cast` 運算子無法轉換掉 `const`、`volatile` 或 `__unaligned` 屬性。如需移除這些屬性的詳細資訊，請參閱[Const\\_cast 運算子](#)。

`reinterpret_cast` 運算子會將 `null` 指標值轉換為目的地類型的 `null` 指標值。

的其中一個實際用法 `reinterpret_cast` 是雜湊函式，它會將值對應至索引，這種方式不常有兩個相異的值會以相同的索引結束。

```
#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` 允許將指標視為整數類資料類型。結果隨即位元移位並與其本身 XOR，以產生唯一的索引（高可攜性獨有）。之後，標準 C-Style 轉換會將該索引截斷為函式的傳回型別。

## 另請參閱

[轉型運算子](#)

[關鍵字](#)

# 執行階段類型資訊

2020/11/2 • [Edit Online](#)

執行階段類型資訊 (RTTI) 是一項機制，可在程式執行期間判斷物件的類型。C++ 語言中加入 RTTI 的原因在於，有許多類別庫的廠商本身實作這項功能。這樣會導致程式庫之間不相容。因此，在語言層級支援執行階段類型資訊的需求變得很明確。

為避免混淆，這裡討論的 RTTI 幾乎完全限於指標。不過，所討論的概念也適用於參考。

執行階段類型資訊有三個主要的 C++ 語言項目：

- [Dynamic\\_cast](#)運算子。

用於多型類型的轉換。

- [typeid](#)運算子。

用於識別物件的實際類型。

- [Type\\_info](#)類別。

用來保存由運算子傳回的類型資訊 `typeid`。

## 另請參閱

[轉型](#)

# bad\_typeid 例外狀況

2020/11/2 • [Edit Online](#)

當的運算元為 Null 指標時， typeid 運算子會擲回bad\_typeid例外狀況 typeid。

## 語法

```
catch (bad_typeid)
    statement
```

## 備註

Bad\_typeid的介面為：

```
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;
};
```

下列範例顯示擲回 typeid bad\_typeid例外狀況的運算子。

```
// 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;
}
}
```

## 輸出

Object is NULL

## 另請參閱

[執行時間類型資訊](#)

[關鍵字](#)

# type\_info 類別

2020/3/25 • [Edit Online](#)

Type\_info 類別描述編譯器在程式內產生的型別資訊。這個類別的物件會實際儲存類型的名稱指標。Type\_info 類別也會儲存編碼的值，適合用來比較兩個類型的相等或排序次序。類型的編碼規則和排序法則不會指定，而且在不同的程式之間也會有所差異。

必須包含 `<typeinfo>` 標頭檔，才能使用 type\_info 類別。Type\_info 類別的介面為：

```
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;
};
```

您無法直接具現化 type\_info 類別的物件，因為類別只有私用複製的函式。建立（暫時性）type\_info 物件的唯一方式是使用 `typeid` 運算子。因為指派運算子也是私用的，所以您無法複製或指派類別 type\_info 的物件。

`type_info::hash_code` 定義雜湊函式，適用於將 typeinfo 類型的值對應到索引值的分佈。

運算子 `==` 和 `!=` 可以分別用來比較是否相等，以及與其他 type\_info 物件的不相等。

類型的排序順序和繼承關係之間並無連結。使用 `type_info::before` 成員函式來判斷類型的排序次序。不保證

`type_info::before` 會在不同的程式中產生相同的結果，甚至是相同程式的不同執行。如此一來，

`type_info::before` 類似 `(&)` 運算子的位址。

`type_info::name` 成員函式會將 `const char*` 傳回為以 null 終止的字串，代表人類可讀取的類型名稱。所指向的記憶體會加以快取，且絕不可直接取消配置。

`type_info::raw_name` 成員函式會將 `const char*` 傳回為以 null 終止的字串，代表物件類型的裝飾名稱。這個名稱實際上是以其裝飾形式儲存，以節省空間。因此，此函數的速度會比 `type_info::name` 快，因為它不需要 undecorate 名稱。`type_info::raw_name` 函數所傳回的字串在比較作業中很有用，但無法讀取。如果您需要人類可讀的字串，請改用 `type_info::name` 函數。

只有在指定 [/GR \(啟用執行時間類型資訊\)](#) 編譯器選項時，才會針對多型類別產生類型資訊。

## 另請參閱

[執行階段類型資訊](#)

# 陳述式 (C++)

2020/4/15 • [Edit Online](#)

C++ 陳述式是控制物件如何以及以什麼順序進行操作的程式項目。本節包括：

- [概觀](#)
- [標記語句](#)
- 陳述式分類
  - [運算片語句](#). 這些陳述式評估運算式以取得其副作用或傳回值。
  - [空語句](#). 在 C++ 語法中需要陳述式但不會採取動作的地方，可以提供這些陳述式。
  - [複合語句](#). 這些陳述式是以大括號 ({ }) 括住的陳述式群組。在任何可以使用單一陳述式的位置，可以使用它們。
  - [選擇敘述](#) 這些陳述式執行測試；若測試判斷值為 true (非零)，則執行一個程式碼區段。如果測試判斷值為 false，它們可能執行另一個程式碼區段。
  - [反覆發敘述](#). 這些陳述式可供重複執行一個程式碼區塊，直到符合指定的終止準則。
  - [跳轉語句](#). 這些陳述式會立刻將控制權轉移到函式中另一個位置或將控制權從函式傳回。
  - [宣告聲明](#). 告宣告會將名稱引入程式中

有關異常處理敘述的資訊，請參閱 [錯誤](#)。

## 另請參閱

[C++ 語言參考](#)

# C++ 陳述式概觀

2020/11/2 • [Edit Online](#)

C++ 陳述式會依序執行，但運算式陳述式、選取陳述式、反覆項目陳述式或跳躍陳述式明確修改該序列時除外。

陳述式可以是下列任何一種類型：

*Labeled-statement*  
*expression-statement*  
*compound-statement*  
*selection-statement*  
*iteration-statement*  
*jump-statement*  
*declaration-statement*  
*try-throw-catch*

在大部分的情況下，C++ 語句語法與 ANSI C89 的語法相同。兩者之間的主要差異在於，在 C89 中，只允許在區塊開頭使用宣告；C++ 加入 *declaration-statement*，可有效地移除此限制。這可讓您在程式中可以計算預先計算過的初始化數值時引入變數。

在區塊內部宣告變數也可讓您精確地控制這些變數的範圍和存留期。

有關語句的文章會說明下列 C++ 關鍵字：

`break`  
`case`  
`catch`  
`continue`  
`default`  
`do`  
  
`else`  
`__except`  
`__finally`  
`for`  
`goto`  
  
`if`  
`__if_exists`  
`__if_not_exists`  
`__leave`  
`return`  
  
`switch`  
`throw`  
`__try`  
`try`  
`while`

[另請參閱](#)

陳述式

# 標記陳述式

2020/11/2 • [Edit Online](#)

標籤可用來將程式控制權直接轉移給指定的陳述式。

```
identifier : statement
case constant-expression : statement
default : statement
```

標籤的範圍宣告該標籤所在的整個函式。

## 備註

標記陳述式可分三種類型。這三種類型全都使用冒號分隔某種類型的標籤與陳述式。`case` 和 `default` 標籤為 `case` 陳述式所特有。

```
#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
}
```

## goto 陳述式

來源程式中識別碼標籤的外觀會宣告標籤。只有 `goto` 語句可以將控制項傳輸至識別碼標籤。下列程式碼片段說明 `goto` 語句和識別碼標籤的使用：

標籤不會單獨顯示，而是必須附加至陳述式。如果需要單獨使用標籤，請在標籤之後放置一個 null 陳述式。

標籤具有函式範圍，而且不能在函式中宣告。然而，在不同的函式中可以使用相同名稱做為標籤。

```

// 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 陳述式

出現在關鍵字後面的標籤 `case` 也不能出現在 `switch` 語句之外。(此限制也適用于 `default` 關鍵字)。下列程式碼片段會顯示標籤的正確用法 `case` :

```

// Sample Microsoft Windows message processing loop.
switch( msg )
{
    case WM_TIMER:      // Process timer event.
        SetClassWord( hWnd, GCW_HICON, ahIcon[nIcon++] );
        ShowWindow( hWnd, SW_SHOWNA );
        nIcon %= 14;
        Yield();
        break;

    case WM_PAINT:
        memset( &ps, 0x00, sizeof(PAINTSTRUCT) );
        hDC = BeginPaint( hWnd, &ps );
        EndPaint( hWnd, &ps );
        break;

    default:
        // This choice is taken for all messages not specifically
        // covered by a case statement.

        return DefWindowProc( hWnd, Message, wParam, lParam );
        break;
}

```

## case 陳述式中的標籤

出現在關鍵字後面的標籤 `case` 也不能出現在 `switch` 語句之外。(此限制也適用于 `default` 關鍵字)。下列程式碼片段會顯示標籤的正確用法 `case` :

```

// Sample Microsoft Windows message processing loop.
switch( msg )
{
    case WM_TIMER: // Process timer event.
        SetClassWord( hWnd, GCW_HICON, ahIcon[nIcon++] );
        ShowWindow( hWnd, SW_SHOWNA );
        nIcon %= 14;
        Yield();
        break;

    case WM_PAINT:
        // Obtain a handle to the device context.
        // BeginPaint will send WM_ERASEBKGND if appropriate.

        memset( &ps, 0x00, sizeof(PAINTSTRUCT) );
        hDC = BeginPaint( hWnd, &ps );

        // Inform Windows that painting is complete.

        EndPaint( hWnd, &ps );
        break;

    case WM_CLOSE:
        // Close this window and all child windows.

        KillTimer( hWnd, TIMER1 );
        DestroyWindow( hWnd );
        if ( hWnd == hWndMain )
            PostQuitMessage( 0 ); // Quit the application.
        break;

    default:
        // This choice is taken for all messages not specifically
        // covered by a case statement.

        return DefWindowProc( hWnd, Message, wParam, lParam );
        break;
}

```

## goto 陳述式中的標籤

來源程式中識別碼標籤的外觀會宣告標籤。只有 [goto](#) 語句可以將控制項傳輸至識別碼標籤。下列程式碼片段說明 [goto](#) 語句和識別碼標籤的使用：

標籤不會單獨顯示，而是必須附加至陳述式。如果需要單獨使用標籤，請在標籤之後放置一個 null 陳述式。

標籤具有函式範圍，而且不能在函式中宣告。然而，在不同的函式中可以使用相同名稱做為標籤。

```

// labels_with_goto.cpp
// compile with: /EHsc
#include <iostream>
int main() {
    using namespace std;
    goto Test2;

    cout << "testing" << endl;

    Test2:
    cerr << "At Test2 label." << endl;
    // At Test2 label.
}

```

## 另請參閱

[C++ 語句總覽](#)

[switch 語句 \(C++\)](#)

# 運算陳述式

2020/3/25 • [Edit Online](#)

運算陳述式會造成對於運算式的評估。運算陳述式不會發生控制權轉移或反覆項目的情形。

運算陳述式的語法很簡單

## 語法

```
[expression] ;
```

## 備註

在下一個陳述式執行之前，會完成運算陳述式中所有運算式的評估，以及所有副作用。最常見的運算陳述式是指派和函式呼叫。因為運算式是選擇性的，所以會將分號單獨視為空的運算式語句，稱為null語句。

## 另請參閱

[C++ 陳述式概觀](#)

# Null 陳述式

2020/3/25 • [Edit Online](#)

「Null 語句」是運算式語句，其中遺漏運算式。它對於呼叫陳述式，但是不進行運算式評估的語言語法相當實用。它是由一個分號所構成。

null 陳述式通常用來做為反覆項目陳述式中的預留位置，或做為在複合陳述式或函式的結尾放置一個標籤的陳述式。

下列程式碼片段顯示如何將某個字串複製到另一個字串，以及合併 null 陳述式：

```
// 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()
{}
```

## 另請參閱

[運算陳述式](#)

# 複合陳述式 (區塊)

2020/11/2 • [Edit Online](#)

複合陳述式是由零個或多個以大括弧({})括住的語句所組成。複合陳述式可以在必須有陳述式的任何位置使用。複合陳述式通常稱為「區塊」(Block)。

## 語法

```
{ [ statement-list ] }
```

## 備註

下列範例會使用複合陳述式作為語句的語句部分(如需 [if](#) 語法的詳細資訊, 請參閱[if 語句](#))：

```
if( Amount > 100 )
{
    cout << "Amount was too large to handle\n";
    Alert();
}
else
{
    Balance -= Amount;
}
```

### NOTE

因為宣告是語句, 所以宣告可以是語句清單中的其中一個語句。因此, 在複合陳述式內宣告, 但未明確宣告為靜態的名稱, 會具有區域範圍和 (為物件時) 存留期。請參閱[範圍](#), 以取得有關使用區域範圍來處理名稱的詳細資料。

## 另請參閱

[C++ 語句總覽](#)

# 選擇陳述式 (C++)

2020/3/25 • [Edit Online](#)

C++選取範圍語句([如果](#)和[參數](#))提供了有條件地執行程式碼區段的方式。

[\\_\\_If\\_exists](#)和[\\_\\_If\\_not\\_exists](#)語句可讓您根據符號是否存在, 有條件地包含程式碼。

請參閱個別主題, 了解每個陳述式的語法。

## 另請參閱

[C++ 陳述式概觀](#)

# if-else 語句 (C++)

2020/11/2 • [Edit Online](#)

If else 語句控制條件式分支。`if-branch` 只有當評估為非零值時，才會執行其中的語句 `condition` (或 `true`)。如果的值 `condition` 為非零值，則會執行下列語句，並略過選擇性的後面的語句 `else`。否則，就會略過下列語句，如果有的話，則會 `else` 執行之後的語句 `else`。

`condition` 評估為非零的運算式為：

- `true`
- 非 `null` 指標，
- 任何非零的算術值，或
- 定義明確轉換為算術、布林值或指標類型的類別類型。(如需轉換的詳細資訊，請參閱 [標準轉換](#)。)

## 語法

`init-statement` :

`expression-statement`  
`simple-declaration`

`condition` :

`expression`  
`attribute-specifier-seq` `opt` `decl-specifier-seq` `opt` `declarator` `** brace-or-equal-initializer`

`statement` :

`expression-statement`  
`compound-statement`

`expression-statement` :

`expression` `opt` ;

`compound-statement` :

{ `statement-seq` `opt` }

`statement-seq` :

`statement`  
`statement-seq` `statement`

`if-branch` :

`statement`

`else-branch` :

`statement`

`selection-statement` :

`if` `constexpr` `opt`<sup>17</sup> ( `init-statement` `opt` 17 `condition` \* ) \*\*\* `if-branch`  
`if` `constexpr` `opt`<sup>17</sup> ( `init-statement` `opt` 17 `condition` ) `if-branch` \* `else` \*\*\* `else-branch`

<sup>17</sup> 從 C++ 17 開始，可以使用這個選擇性元素。

## if else 語句

在語句的所有形式中，`if` `condition` 可以有任何值（結構除外），包括所有副作用。控制權會從 `if` 語句傳遞至程式中的下一個語句，除非執行的 `if-branch` 或 `else-branch` 包含 `break`、`continue` 或 `goto`。

`else` 語句的子句 `if...else` 與 `if` 相同範圍中不具有對應語句的最接近先前語句相關聯 `else`。

## 範例

此範例程式碼會顯示使用中的數個 `if` 語句，以及和不包含的語句 `else`：

```
// if_else_statement.cpp
#include <iostream>

using namespace std;

class C
{
public:
    void do_something(){}
};

void init(C){}
bool is_true() { return true; }
int x = 10;

int main()
{
    if (is_true())
    {
        cout << "b is true!\n"; // executed
    }
    else
    {
        cout << "b is false!\n";
    }

    // no else statement
    if (x == 10)
    {
        x = 0;
    }

    C* c;
    init(c);
    if (c)
    {
        c->do_something();
    }
    else
    {
        cout << "c is null!\n";
    }
}
```

## if 語句搭配初始化運算式

從 C++17 開始，`if` 語句也可以包含宣告 `init-statement` 和初始化命名變數的運算式。只有在 `if` 語句的範圍內需要變數時，才能使用這種形式的 `if` 語句。Microsoft專屬：從 Visual Studio 2017 15.3 版開始，可以使用此表單，而且至少需要 `/std:c++17` 編譯器選項。

## 範例

```

#include <iostream>
#include <mutex>
#include <map>
#include <string>
#include <algorithm>

using namespace std;

map<int, string> m;
mutex mx;
bool shared_flag; // guarded by mx
void unsafe_operation() {}

int main()
{
    if (auto it = m.find(10); it != m.end())
    {
        cout << it->second;
        return 0;
    }

    if (char buf[10]; fgets(buf, 10, stdin))
    {
        m[0] += buf;
    }

    if (lock_guard<mutex> lock(mx); shared_flag)
    {
        unsafe_operation();
        shared_flag = false;
    }

    string s{ "if" };
    if (auto keywords = { "if", "for", "while" }; any_of(keywords.begin(), keywords.end(), [&s](const char* kw) { return s == kw; } ))
    {
        cout << "Error! Token must not be a keyword\n";
    }
}

```

從 C++17 開始，您可以使用函式樣板 `if constexpr` 中的語句來進行編譯時間分支決策，而不需要使用多個函數多載。Microsoft 專屬：從 Visual Studio 2017 15.3 版開始，可以使用此表單，而且至少需要 [/std:c++17](#) 編譯器選項。

## 範例

這個範例會示範如何撰寫處理參數解壓縮的單一函式。不需要零參數多載：

```
template <class T, class... Rest>
void f(T&& t, Rest&&... r)
{
    // handle t
    do_something(t);

    // handle r conditionally
    if constexpr (sizeof...(r))
    {
        f(r...);
    }
    else
    {
        g(r...);
    }
}
```

## 另請參閱

[選取範圍語句](#)

[關鍵字](#)

[switch \(C++\) 的語句](#)

# `_if_exists` 陳述式

2020/11/2 • [Edit Online](#)

`_if_exists` 語句會測試指定的識別碼是否存在。如果識別項存在，就會執行指定的陳述式區塊。

## 語法

```
_if_exists ( identifier ) {  
    statements  
};
```

### 參數

#### 識別碼

要測試其是否存在的識別項。

### 語句

當 識別碼 存在時要執行的一或多個語句。

## 備註

### Caution

若要達到最可靠的結果，請使用 `_if_exists` 下列條件約束底下的語句。

- 只將 `_if_exists` 語句套用至簡單類型，而不是範本。
- 將 `_if_exists` 語句套用至類別內部或外部的識別碼。請勿將語句套用 `_if_exists` 至本機變數。
- 請 `_if_exists` 只在函式主體中使用語句。在函式主體之外，`_if_exists` 語句只能測試完整定義的類型。
- 當您對多載函式進行測試時，無法對特定形式的多載進行測試。

語句的補數 `_if_exists` 是 `_if_not_exists` 語句。

## 範例

請注意，這個範例會使用範本，但不建議這樣做。

```

// 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;
}

```

## 輸出

```

In X<T>::Dump()
In A::Dump()
In X<T>::Dump()
T::Dump does not exist
g_bFlag = 1
C::f exists

```

## 另請參閱

[選取範圍陳述式](#)

[關鍵字](#)

[\\_\\_if\\_not\\_exists 語句](#)

# `_if_not_exists` 陳述式

2020/11/2 • [Edit Online](#)

`_if_not_exists` 語句會測試指定的識別碼是否存在。如果識別項不存在，就會執行指定的陳述式區塊。

## 語法

```
_if_not_exists ( identifier ) {  
    statements  
};
```

### 參數

#### 識別碼

要測試其是否存在的識別項。

### 語句

當 識別碼 不存在時要執行的一或多個語句。

## 備註

### Caution

若要達到最可靠的結果，請使用 `_if_not_exists` 下列條件約束底下的語句。

- 只將 `_if_not_exists` 語句套用至簡單類型，而不是範本。
- 將 `_if_not_exists` 語句套用至類別內部或外部的識別碼。請勿將語句套用 `_if_not_exists` 至本機變數。
- 請 `_if_not_exists` 只在函式主體中使用語句。在函式主體之外，`_if_not_exists` 語句只能測試完整定義的類型。
- 當您對多載函式進行測試時，無法對特定形式的多載進行測試。

語句的補數 `_if_not_exists` 是 `_if_exists` 語句。

## 範例

如需如何使用的範例 `_if_not_exists`，請參閱 [\\_if\\_exists 語句](#)。

## 另請參閱

[選取範圍陳述式](#)

[關鍵字](#)

[\\_if\\_exists 語句](#)

# switch 語句 ( C ++ )

2020/12/10 • [Edit Online](#)

根據整數運算式的值，允許在多個程式碼區段中選取範圍。

## 語法

```
selection-statement :  
    switch ( [ init-statement ]opt C++17 condition ) statement
```

```
init-statement :  
    expression-statement  
    simple-declaration
```

```
condition :  
    expression  
    attribute-specifier-seqopt decl-specifier-seq declarator brace-or-equal-initializer
```

```
Labeled-statement :  
    case constant-expression : statement  
    default : statement
```

## 備註

switch 語句會 Labeled-statement 根據的值，讓控制項在其語句主體中傳送至其中一個 condition 。

condition 必須有整數類型，或必須是明確轉換成整數類資料類型的類別類型。整數升級會依照標準轉換中的說明進行。

switch 語句主體是由一系列 case 標籤和 optional default 標籤所組成。Labeled-statement 是下列其中一個標籤和後面的語句。標記的語句不是語法需求，但是 switch 語句沒有意義。constant-expression 語句中的兩個值 case 都不能評估為相同的值。default 標籤可能只會出現一次。default 語句通常會放在結尾，但它可以出現在語句主體中的任何位置 switch 。case 或 default 標籤只能出現在語句內 switch 。

constant-expression 每個 case 標籤中的會轉換成與相同類型的常數值 condition 。然後，它會與 condition 相等進行比較。控制權會傳遞給 case constant-expression 符合值之值後面的第一個語句 condition 。產生的行為如下表所示。

### switch 語句行為

if	if
轉換的值與升級之控制運算式的值相符。	控制權會轉移至標籤後面的陳述式。
沒有任何常數符合標籤中的常數 case ; default 標籤存在。	控制項會傳送至 default 標籤。

II

II

沒有任何常數符合標籤中的常數 `case`；不 `default` 存在任何標籤。

控制權會傳送至語句之後的語句 `switch`。

如果找到相符的運算式，執行可以繼續到稍後 `case` 或 `default` 標籤。`break` 語句是用來停止執行，並將控制權轉移給語句後面的語句 `switch`。如果沒有 `break` 語句，`case` 則會執行符合的標籤到結尾的每個語句 `switch`（包括 `default`）。例如：

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

在上述範例中，`uppercase_A` 如果 `c` 是上限，會遞增 `case 'A'`。`break` 之後的語句會 `uppercase_A++` 終止 `switch` 語句主體的執行，並將控制權傳遞給 `while` 迴圈。如果沒有 `break` 語句，執行會「流經」到下一個加上標籤的語句，因此 `lowercase_a` 和 `other` 也會遞增。的語句會提供類似的目的 `break` `case 'a'`。如果較 `c` 低 `case 'a'`，`lowercase_a` 會遞增，而且 `break` 語句會終止 `switch` 語句主體。如果 `c` 不是 `'a'` 或 `'A'`，則 `default` 會執行語句。

**Visual Studio 2017 和更新版本：**(適用于 `/std: c++17`) `[[fallthrough]]` 屬性是以 `c++17` 標準指定。您可以在語句中使用它 `switch`。這是編譯器的提示，或讀取程式碼的任何人，都是故意的行為。Microsoft `c++` 編譯器目前不會警告 `fallthrough` 行為，因此這個屬性不會影響編譯器行為。在此範例中，屬性會套用至未結束標記之語句內的空白語句。換句話說，這是必要的分號。

```
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 版和更新版本(適用于[/std: c + + 17](#))。`switch` 語句可能會有一個以 `init-statement` 分號結尾的子句。它會引進並初始化變數，其範圍僅限於語句的區塊 `switch`：

```
switch (Gadget gadget(args); auto s = gadget.get_status())
{
    case status::good:
        gadget.zip();
        break;
    case status::bad:
        throw BadGadget();
};
```

語句的內部區塊 `switch` 可以包含具有初始化運算式的定義，只要它們可連線 *reachable*，也就是所有可能的執行路徑都不會略過。使用這些宣告引入的名稱有區域範圍。例如：

```
// 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` 語句可以嵌套。當加上 nested 時，`case` 或 `default` 標籤會與包圍它們的最接近語句產生關聯  
`switch`。

### Microsoft 特有的行為

Microsoft C++ 不會限制 `case` 語句中的值數目 `switch`。此數目會受到可用記憶體的限制。

## 另請參閱

[選取範圍陳述式](#)

[關鍵字](#)

# 反覆運算陳述式 (C++)

2020/11/2 • [Edit Online](#)

反覆項目陳述式會使陳述式 (或複合陳述式) 依據某種迴圈終止準則執行零次或多次。當這些語句是複合陳述式時，它們會依序執行，但當遇到`break`語句或`continue`語句時除外。

C++ 提供四個反復專案語句：`while`、`do`、`for`和範圍架構的。每一個都會反復執行，直到其終止運算式評估為零 (false)，或直到迴圈終止是使用語句來強制執行為止 `break`。下表摘要說明這些陳述式及它們的動作，而且每一個陳述式會在後續章節中詳細討論。

## 反覆運算陳述式

語句	迴圈範圍	終止條件	是否強制執行
<code>while</code>	迴圈頂端	否	否
<code>do</code>	迴圈底部	否	否
<code>for</code>	迴圈頂端	是	是
■ <code>for</code>	迴圈頂端	是	是

反覆項目陳述式的陳述式部分不可以是宣告。不過，它可以是包含宣告的複合陳述式。

## 另請參閱

[C++ 語句總覽](#)

# while 陳述式 (C++)

2020/11/2 • [Edit Online](#)

重複執行語句，直到*expression*評估為零為止。

## 語法

```
while ( expression )
    statement
```

## 備註

運算式的測試會在每次執行迴圈之前進行；因此，`while` 迴圈會執行零次或多次。運算式必須是整數類資料類型、指標類型，或是明確轉換成整數或指標類型的類別類型。

在 `while` 語句主體中執行 `break`、`goto` 或 `return` 時，迴圈也可以終止。請使用 [繼續] 終止目前的反復專案，而不結束 `while` 迴圈。`continue` 將控制權傳遞給迴圈的下一個反復專案 `while`。

下列程式碼會使用 `while` 迴圈來修剪字串尾端的底線：

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

終止條件在迴圈頂端進行評估。如果沒有結尾底線，此迴圈絕不會執行。

## 另請參閱

[反覆運算陳述式](#)

[關鍵字](#)

[do while 語句 \(C++\)](#)

[for 語句 \(C++\)](#)

[以範圍為基礎的 for 陳述式 \(C++\)](#)



# do-while 陳述式 (C++)

2020/3/25 • [Edit Online](#)

重複執行語句，直到指定的終止條件(運算式)評估為零為止。

## 語法

```
do
    statement
while ( expression ) ;
```

## 備註

終止條件的測試是在每次執行迴圈之後進行，因此，**do while**迴圈會執行一次或多次，視終止運算式的值而定。在陳述式主體中執行 **break**、**goto** 或 **return** 陳述式時，**do-while** 陳述式也可能會終止。

*expression* 必須有算術或指標類型。執行程序如下所示：

1. 會執行陳述式主體。
2. 接下來會評估 *expression*。如果 *expression* 為 false，**do-while** 陳述式會終止，而並將控制權傳遞至程式中的下一個陳述式。如果 *expression* 為 true (非零)，則會從步驟 1 開始重複處理序。

## 範例

下列範例示範**do while**語句：

```
// do_while_statement.cpp
#include <stdio.h>
int main()
{
    int i = 0;
    do
    {
        printf_s("\n%d", i++);
    } while (i < 3);
}
```

## 另請參閱

[反覆運算陳述式](#)

[關鍵字](#)

[while 陳述式 \(C++\)](#)

[for 陳述式 \(C++\)](#)

[以範圍為基礎的 for 陳述式 \(C++\)](#)

# for 語句 ( C + + )

2020/11/2 • [Edit Online](#)

重複執行陳述式，直到條件變成 false。如需範圍架構語句的詳細資訊 `for`，請參閱以範圍為基礎的 `for` 語句(c++)。

## Syntax

```
for ( init-expression ; cond-expression ; Loop-expression )  
    statement
```

## 備註

使用 `for` 語句來建立必須執行指定次數的迴圈。

`for` 語句是由三個選擇性元件所組成，如下表所示。

### for 迴圈元素

元素	說明	用途
<code>init-expression</code>	在語句的任何其他元素之前 <code>for</code> ， <code>init-expression</code> 只會執行一次。控制項接著會傳遞至 <code>cond-expression</code> 。 。	通常用來初始迴圈索引。它可能包含運算式或宣告。
<code>cond-expression</code>	執行的每個反復專案之前 <code>statement</code> ，包括第一個反復專案。 <code>statement</code> 只有在 <code>cond-expression</code> 評估為 true (非零)時，才會執行。	判斷值為整數類型的運算式或具有整數類型明確轉換的類別類型。通常用來測試迴圈終止準則。
<code>Loop-expression</code>	在每個反復專案結束時 <code>statement</code> 。在 <code>Loop-expression</code> 執行之後， <code>cond-expression</code> 會評估。	通常用來遞增迴圈索引。

下列範例示範使用語句的不同方式 `for`。

```

#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* 和 *Loop-expression* 可以包含多個語句，並以逗號分隔。例如：

```

#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* 可以遞增或遞減，或以其他方式修改。

```

#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

```

`for` `break` 執行中的、`return` 或 `goto` (指向迴圈外的標籤語句) 時，迴圈 `for` 會終止 `statement`。`continue` 復圈中的語句 `for` 只會終止目前的反復專案。

如果 `cond-expression` 省略，則會將它視為 `true`，而且在 `for` 中，如果沒有 `break`、或，迴圈就不會終止 `return` `goto` `statement`。

雖然語句的三個欄位 `for` 通常用於初始化、測試終止和遞增，但它們並不限於這些用途。例如，下列程式碼會列

印數字 0 到 4。在此情況下，`statement` 是 null 語句：

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

int main()
{
    int i;
    for( i = 0; i < 5; cout << i << '\n', i++){
        ;
    }
}
```

## for 迴圈和 C + + 標準

C + + 標準指出迴圈結束之後，在迴圈中宣告的變數 `for` 應該超出範圍 `for`。例如：

```
for (int i = 0 ; i < 5 ; i++) {
    // do something
}
// i is now out of scope under /Za or /Zc:forScope
```

根據預設，在`/ze`底下，在迴圈中宣告的變數會 `for` 保留在範圍內，直到 `for` 迴圈的封閉範圍結束為止。

`/Zc: forScope` 可啟用 for 迴圈中宣告之變數的標準行為，而不需要指定 `/za`。

您也可以使用迴圈的範圍差異來重新宣告底下的 `for` 變數 `/ze`，如下所示：

```
// 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++ ) {}
}
```

這種行為更嚴格地模仿迴圈中宣告之變數的標準行為 `for`，而這需要迴圈中宣告的變數在迴圈 `for` 完成之後移出範圍。在迴圈中宣告變數時 `for`，編譯器會在內部將它升級為 `for` 迴圈封閉範圍中的區域變數。即使已經有相同名稱的本機變數，它還是會升級。

## 另請參閱

[反復專案語句](#)

[關鍵字](#)

[while 語句\(c + +\)](#)

[do while 語句\(c + +\)](#)

[以範圍為基礎的 for 語句\(c + +\)](#)

# 以範圍為基礎的 for 陳述式 (C++)

2020/11/2 • [Edit Online](#)

對於 `statement` 中的每個項目，重複且循序地執行 `expression`。

## 語法

```
* for ( ***for 範圍聲明 : 運算式 )  
    語句
```

## 備註

使用範圍架構的 `for` 語句來建立必須透過範圍執行的迴圈，其定義為您可以逐一查看的任何專案(例如)，`std::vector` 或任何其他 C++ 標準程式庫序列，其範圍是由和所定義 `begin()` `end()`。在部分中宣告的名稱在 `for-range-declaration` 語句中是本機的 `for`，而且不能在或中重新 `expression` 告知 `statement`。請注意，在 `auto` 語句的部分中，關鍵字是慣用的 `for-range-declaration`。

**Visual Studio 2017 中的新功能：**以範圍為基礎的 `for` 迴圈不再需要該 `begin()` 和傳回 `end()` 相同類型的物件。這可讓傳回 `end()` sentinel 物件，例如範圍-V3 提案中定義的範圍所使用的。如需詳細資訊，請參閱在 GitHub 上一般化以範圍為基礎的 `For` 迴圈和範圍 v3 程式庫。

這段程式碼會示範如何使用以範圍為基礎的 `for` 迴圈，逐一查看陣列和向量：

```

// 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;
}

```

輸出如下：

```

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

```

`for` 當中的其中一個執行時，以範圍為基礎的迴圈 `statement` 會終止：`break`、`return` 或 `goto` 至以範圍為基礎之迴圈以外的標記語句 `for`。`continue` 以範圍為基礎的迴圈中的語句 `for` 只會終止目前的反復專案。

請記住下列有關範圍型的事實 `for`：

- 自動辨識陣列。
- 辨識具有 `.begin()` 和 `.end()` 的容器。
- 使用與引數相依的查閱 `begin()` 和 `end()` 以取得任何其他項目。

## 另請參閱

`auto`

反覆運算陳述式

關鍵字

`while` 語句(`c++`)

`do-while` 語句(`c++`)

`for` 語句(`c++`)

# 跳躍陳述式 (C++)

2020/3/25 • [Edit Online](#)

C ++ 跳躍陳述式會立即轉移區域的控制權。

## 語法

```
break;  
continue;  
return [expression];  
goto identifier;
```

## 備註

如需 C++ 跳躍陳述式的說明，請參閱下列主題。

- [break 陳述式](#)
- [continue 陳述式](#)
- [return 陳述式](#)
- [goto 陳述式](#)

## 另請參閱

[C++ 陳述式概觀](#)

# break 陳述式 (C++)

2020/11/2 • [Edit Online](#)

`break` 語句會結束執行最近的封閉迴圈或其出現的條件陳述式。控制會傳遞至陳述式結尾之後的陳述式 (如果有有的話)。

## 語法

```
break;
```

## 備註

`break` 語句適用於條件式`switch`語句以及`do`、`for`和`while`迴圈語句。

在`switch`語句中，`break` 語句會導致程式執行語句外的下一個語句`switch`。如果沒有`break` 語句，`case` 則會執行符合的標籤到語句結尾的每個語句`switch` (包括`default` 子句)。

在迴圈中，`break` 語句會結束執行最接近的封閉式`do`、`for`或`while`語句。控制會傳遞到已結束之陳述式的下一個陳述式 (如果有有的話)。

在 nested 語句中，`break` 語句只會結束`do` 立即括住的、`for`、`switch`或`while` 語句。您可以使用`return` 或`goto` 語句，從更深層的嵌套結構中傳輸控制權。

## 範例

下列程式碼顯示如何`break` 在迴圈中使用語句`for`。

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

int main()
{
    // An example of a standard for loop
    for (int i = 1; i < 10; i++)
    {
        if (i == 4) {
            break;
        }
        cout << i << '\n';
    }

    // An example of a range-based for loop
    int nums []{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    for (int i : nums) {
        if (i == 4) {
            break;
        }
        cout << i << '\n';
    }
}
```

In each case:

1

2

3

下列程式碼顯示如何 `break` 在 `while` 迴圈和迴圈中使用 `do`。

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

int main() {
    int i = 0;

    while (i < 10) {
        if (i == 4) {
            break;
        }
        cout << i << '\n';
        i++;
    }

    i = 0;
    do {
        if (i == 4) {
            break;
        }
        cout << i << '\n';
        i++;
    } while (i < 10);
}
```

In each case:

0123

下列程式碼顯示如何 `break` 在 `switch` 語句中使用。`break` 如果您想要個別處理每個案例，您必須在每個案例中使用；如果不使用 `break`，程式碼執行會落在下一個案例中。

```
#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";
    }
}
```

## 另請參閱

[跳躍陳述式](#)

[關鍵字](#)

[continue 語句](#)

# continue 陳述式 (C++)

2020/11/2 • [Edit Online](#)

強制將控制權轉移至最小封閉式 `do`、`for` 或 `while` 迴圈的控制運算式。

## 語法

```
continue;
```

## 備註

目前反覆項目中其餘的任何陳述式都不會執行。迴圈中下一個反覆項目的判斷方式如下：

- 在 `do` 或 `while` 迴圈中，下一個反復專案一開始會對或語句的控制運算式 `do` `while`。
- 在 `for` 迴圈中(使用語法 `for( <init-expr> ; <cond-expr> ; <loop-expr> )`)，`<loop-expr>` 會執行子句。然後會重新求出 `<cond-expr>` 子句的值，而根據結果，迴圈會結束或另一個反覆項目會發生。

下列範例示範如何 `continue` 使用語句來略過程式碼區段，並開始進行迴圈的下一個反復專案。

## 範例

```
// continue_statement.cpp
#include <stdio.h>
int main()
{
    int i = 0;
    do
    {
        i++;
        printf_s("before the continue\n");
        continue;
        printf("after the continue, should never print\n");
    } while (i < 3);

    printf_s("after the do loop\n");
}
```

```
before the continue
before the continue
before the continue
after the do loop
```

## 另請參閱

[跳躍陳述式](#)

[關鍵字](#)

# return 陳述式 (C++)

2020/11/2 • [Edit Online](#)

終止函式執行並將控制項傳回至進行呼叫的函式 (或者, 如果您是從 `main` 函式傳送控制項, 則傳回至作業系統)。執行作業會在進行呼叫的函式中緊接著呼叫之後繼續進行。

## 語法

```
return [expression];
```

## 備註

`expression` 子句 (如果有的話) 會轉換成函式宣告中指定的類型, 就像執行初始化一般。從運算式的類型轉換成 `return` 函數的類型, 可以建立暫存物件。如需如何和何時建立而暫存物件的詳細資訊, 請參閱[暫存物件](#)。

`expression` 子句的值會傳回至進行呼叫的函式。如果省略運算式, 則函式的傳回值會是未定義。函式和析構函式以及類型的函式 `void`, 無法在語句中指定運算式 `return`。所有其他類型的函式都必須在語句中指定運算式 `return`。

當控制流程結束封閉函式定義的區塊時, 其結果會與未 `return` 執行運算式的語句相同。這對於宣告為傳回值的函式是無效的。

函數可以有任意數目的 `return` 語句。

下列範例會使用運算式搭配 `return` 語句來取得兩個整數的最大值。

## 範例

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

## 另請參閱

[跳躍陳述式](#)

[關鍵字](#)

# goto 陳述式 (C++)

2020/11/2 • [Edit Online](#)

`goto` 語句會無條件地將控制權轉移至指定之識別碼所標記的語句。

## 語法

```
goto identifier;
```

## 備註

`identifier` 所指定之標記的陳述式必須位於目前的函式。所有 `identifier` 名稱是內部命名空間的成員，因此不會干擾其他識別項。

語句標籤只對語句有意義 `goto`，否則會忽略語句標籤。標籤不能重新宣告。

`goto` 不允許使用語句將控制權轉移至位置，而該位置會略過該位置範圍內任何變數的初始化。下列範例會引發 C2362：

```
int goto_fn(bool b)
{
    if (!b)
    {
        goto exit; // C2362
    }
    else
    { /*...*/ }

    int error_code = 42;

exit:
    return error_code;
}
```

這是很好的程式設計風格 `break`，盡可能使用 `continue` 和 `return` 語句，而不是 `goto` 語句。不過，因為 `break` 語句只會從一個迴圈層級結束，所以您可能必須使用 `goto` 語句來結束深度的嵌套迴圈。

如需標籤和語句的詳細資訊 `goto`，請參閱標記的語句。

## 範例

在此範例中，`goto` 語句會將控制項傳輸至 `stop` `i` 等於3的點。

```
// goto_statement.cpp
#include <stdio.h>
int main()
{
    int i, j;

    for ( i = 0; i < 10; i++ )
    {
        printf_s( "Outer loop executing. i = %d\n", i );
        for ( j = 0; j < 2; j++ )
        {
            printf_s( " Inner loop executing. j = %d\n", j );
            if ( i == 3 )
                goto stop;
        }
    }

    // This message does not print:
    printf_s( "Loop exited. i = %d\n", i );

    stop:
    printf_s( "Jumped to stop. i = %d\n", i );
}
```

```
Outer loop executing. i = 0
Inner loop executing. j = 0
Inner loop executing. j = 1
Outer loop executing. i = 1
Inner loop executing. j = 0
Inner loop executing. j = 1
Outer loop executing. i = 2
Inner loop executing. j = 0
Inner loop executing. j = 1
Outer loop executing. i = 3
Inner loop executing. j = 0
Jumped to stop. i = 3
```

## 另請參閱

[跳躍陳述式](#)

[關鍵字](#)

# 控制權轉移

2020/11/2 • [Edit Online](#)

您可以 `goto`、`case` 在語句中使用語句或標籤 `switch`，來指定分支超過初始化運算式的程式。除非包含初始設定式的宣告所在區塊是由跳躍陳述式發生的區塊所包圍，否則這類程式碼是不合法的。

下列範例將示範宣告和初始化 `total`、`ch` 和 `i` 物件的迴圈。另外還有錯誤 `goto` 的語句，它會將控制權轉移到初始化運算式之後。

```
// 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;
    }
}
```

在上述範例中，`goto` 語句會嘗試將控制項傳送到超過的初始化 `i`。不過，如果 `i` 已宣告但尚未初始化，則這項傳送是合法的。

`total` `ch` `statement` `while` 當使用語句結束區塊時，會終結在區塊中宣告做為語句之語句的物件和 `break`。

# 命名空間 (C++)

2020/11/2 • [Edit Online](#)

命名空間是提供其內識別項 (類型、函式、變數等的名稱) 範圍的宣告式區域。命名空間用來將程式碼組織成邏輯群組，以及防止特別在程式碼基底包含多個程式庫時可能會發生的名稱衝突。在命名空間範圍的所有識別項都可以看得到彼此，沒有限制。命名空間外的識別碼可以使用每個識別碼的完整名稱來存取成員，例如

`std::vector<std::string> vec;`，或透過單一識別碼的`using`宣告 (`using std::string`)，或命名空間中所有識別碼的`using`指示詞 (`using namespace std;`)。標頭檔中的程式碼應該一律使用完整命名空間名稱。

下列範例顯示一個命名空間宣告，以及命名空間外部的程式碼可以存取其成員的三種方式。

```
namespace ContosoData
{
    class ObjectManager
    {
    public:
        void DoSomething() {}
    };
    void Func(ObjectManager) {}
}
```

使用完整名稱：

```
ContosoData::ObjectManager mgr;
mgr.DoSomething();
ContosoData::Func(mgr);
```

使用 `using` 宣告，將一個識別項帶入範圍中：

```
using ContosoData::ObjectManager;
ObjectManager mgr;
mgr.DoSomething();
```

使用 `using` 指示詞，將命名空間中的所有項目帶入範圍中：

```
using namespace ContosoData;

ObjectManager mgr;
mgr.DoSomething();
Func(mgr);
```

## using 指示詞

指示詞 `using` 允許使用中的所有名稱，`namespace` 而不需要命名空間名稱做為明確限定詞。如果您在命名空間中使用數個不同的識別碼，請在執行檔(亦即 \*.cpp)中使用 `using` 指示詞；如果您只是使用一或兩個識別碼，請考慮使用宣告，只將這些識別碼帶入範圍中，而不是命名空間中的所有識別碼。如果區域變數的名稱和命名空間變數相同，命名空間變數將會隱藏。命名空間變數的名稱與全域變數名稱相同，會產生錯誤。

#### NOTE

using 指示詞可以放在 .cpp 檔案頂端 (在檔案範圍中) 或是類別或函式定義內。

一般而言，請避免將 using 指示詞放在標頭檔 (\*.h) 中，因為包含該標頭的所有檔案都會將命名空間中的所有項目帶入範圍中，這樣會導致很難偵錯的名稱隱藏和命名衝突問題。請一律在標頭檔中使用完整名稱。如果這些名稱太長，您可以使用命名空間別名來縮短它們。(請參閱下方。)

## 宣告命名空間和命名空間成員

一般而言，您可以在標頭檔中宣告命名空間。如果函式實作位於個別的檔案，則請限定函式名稱 (如此範例所示)。

```
//contosoData.h
#pragma once
namespace ContosoDataServer
{
    void Foo();
    int Bar();
}
```

即使您將指示詞放在檔案的頂端，contosodata 中的函式建立也應該使用完整名稱 using：

```
#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;}
```

命名空間可以宣告在單一檔案的多個區塊中以及多個檔案中。編譯器會在前置處理期間將組件加在一起，而產生的命名空間包含所有組件中宣告的所有成員。其中一個範例是標準程式庫之每個標頭檔中所宣告的 std 命名空間。

具名命名空間的成員可以在以所要定義名稱的明確限定性宣告這些成員所在的命名空間之外定義。不過，定義必須出現在命名空間中含括宣告之命名空間的宣告位置後面。例如：

```
// 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();
}
```

如果跨多個標頭檔宣告命名空間成員，而且您尚未以正確的順序包括這些標頭，則會發生此錯誤。

## 全域命名空間

如果明確命名空間中未宣告識別項，則它是隱含全域命名空間的一部分。一般來說，請盡可能避免在全域範圍內

進行宣告，但不包括進入點main函式，這必須是在全域命名空間中。若要明確限定全域識別項，請使用沒有名稱的範圍解析運算子（如在 `::SomeFunction(x);` 中）。這會區分識別項與任何其他命名空間中具有相同名稱的任何項目，也有助於讓其他人容易了解您的程式碼。

## std 命名空間

所有 C++ 標準程式庫類型和函式都宣告于 `std` 內部的命名空間或命名空間中 `std`。

## 巢狀命名空間

命名空間可能是巢狀命名空間。一般的嵌套命名空間具有其父系成員的不合格存取權，但父成員不會對嵌套的命名空間具有不合格的存取權（除非它宣告為 inline），如下列範例所示：

```
namespace ContosoDataServer
{
    void Foo();

    namespace Details
    {
        int CountImpl;
        void Bar() { return Foo(); }
    }

    int Baz(int i) { return Details::CountImpl; }
}
```

一般巢狀命名空間可以用來封裝不屬於父命名空間之公用介面的內部實作詳細資料。

## 內嵌命名空間 (C++ 11)

相較於一般巢狀命名空間，內嵌命名空間成員是視為父命名空間的成員。這種特性會啟用多載函式上的引數相依查閱，以處理父命名空間和巢狀內嵌命名空間中具有多載的函式。它也可讓您宣告在內嵌命名空間中所宣告樣板之父命名空間中的特製化。下列範例示範外部程式碼預設如何繫結到內嵌命名空間：

```

//Header.h
#include <string>

namespace Test
{
    namespace old_ns
    {
        std::string Func() { return std::string("Hello from old"); }
    }

    inline namespace new_ns
    {
        std::string Func() { return std::string("Hello from new"); }
    }
}

#include "header.h"
#include <string>
#include <iostream>

int main()
{
    using namespace Test;
    using namespace std;

    string s = Func();
    std::cout << s << std::endl; // "Hello from new"
    return 0;
}

```

下列範例示範如何在內嵌命名空間中所宣告樣板之父系中的特製化。

```

namespace Parent
{
    inline namespace new_ns
    {
        template <typename T>
        struct C
        {
            T member;
        };
    }
    template<>
    class C<int> {};
}

```

您可以使用內嵌命名空間做為版本設定機制，來管理程式庫的公用介面變更。例如，您可以建立單一父命名空間，並封裝父命名空間內其專屬巢狀命名空間中的每個介面版本。保留最新或慣用版本的命名空間限定為內嵌，因此予以公開，就像它是父命名空間的直接成員一樣。叫用 Parent::Class 的用戶端程式碼將會自動繫結至新的程式碼。偏好使用舊版本的用戶端還是可以使用具有該程式碼之巢狀命名空間的完整路徑來存取它。

內嵌關鍵字必須套用至編譯單位中命名空間的第一個宣告。

下列範例示範介面的兩個版本，各位於巢狀命名空間中。`v_20` 命名空間具有 `v_10` 介面的某個修改，並標示為內嵌。使用新程式庫並呼叫 `Contoso::Funcs::Add` 的用戶端程式碼將會叫用 `v_20` 版本。嘗試呼叫 `Contoso::Funcs::Divide` 的程式碼現在會產生編譯時期錯誤。如果它們真正需要該函式，則仍然可以透過明確呼叫 `Contoso::v_10::Funcs::Divide` 來存取 `v_10` 版本。

```

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);
        };
    }
}

```

## 命名空間別名

命名空間名稱必須是唯一的，這表示通常應該不會太短。如果名稱的長度使程式碼難以讀取，或在標頭檔中輸入不能使用的指示詞，則您可以建立命名空間別名做為實際名稱的縮寫。例如：

```

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間

您可以建立明確命名空間，而不是指定它的名稱：

```

namespace
{
    int MyFunc(){}
}

```

這稱為「未命名」或「匿名」命名空間，當您想要讓其他檔案中的程式碼看不到變數宣告（也就是提供內部連結），而不需要建立命名空間時，這會很有用。相同檔案中的所有程式碼都可以看到未命名的命名空間中的識別項，但在該檔案外部（或更精確地來說是外部轉譯單位）看不到識別碼以及命名空間本身。

## 另請參閱

[宣告和定義](#)

# 列舉 (C++)

2020/11/2 • [Edit Online](#)

列舉是使用者定義類型，其中包含一組具名的整數常數，稱為列舉值。

## NOTE

本文涵蓋 ISO Standard C++ 語言 `enum` 類型，以及在 C++11 中引進的範圍(或強型別)■類型。如需 C++/CLI 和 C++/CX 中■或■類型的詳細資訊，請參閱[enum 類別](#)。

## 語法

```
// 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
```

## 參數

### 標識

提供給列舉的類型名稱。

### *type*

列舉值的基礎類型，所有列舉值都有相同的基礎類型。可以是任何整數類資料類型。

### 列舉清單

在列舉中，列舉值的逗號分隔清單。範圍內的每個列舉值或變數名稱都必須是唯一的。不過，值可以重複。在不限範圍的列舉中，範圍是周圍的範圍；在限定範圍的列舉中，範圍是列舉清單本身。在限定範圍的列舉中，清單可能是空的，這實際上會定義新的整數類資料類型。

### *class*

藉由在宣告中使用此關鍵字，您可以指定列舉的範圍，而且必須提供識別碼。您也可以使用 `struct` 關鍵字取代 `class`，因為在此內容中，它們在語義上是相等的。

## 列舉值範圍

列舉提供內容，描述表示為具名常數的值範圍（也稱為列舉值）。在原始 C 和 C++ 列舉類型中，不合格的列舉值會顯示在宣告列舉的範圍中。在限定範圍列舉中，列舉值名稱必須由列舉類型名稱來限定。下列範例示範這兩種列舉之間的這項基本差異：

```

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
        { /*...*/ }
    }
}

```

列舉中的每個名稱被指派整數值，這個值對應於它在列舉值順序中的位置。根據預設，第一個指派的值是 0，下一個指派的是 1，依此類推。不過，您可以明確設定列舉程式的值，如下所示：

```
enum Suit { Diamonds = 1, Hearts, Clubs, Spades };
```

指派給列舉值 `Diamonds` 的值為 `1`。後續列舉值，如果未提供其明確值，則接收前一個列舉值的值加一。在上述範例中，`Hearts` 會有值 `2`，而 `Clubs` 會有 `3`，依此類推。

每個列舉值都會被視為常數，而且在定義的範圍內 `enum`（適用於不限範圍的列舉）或本身（針對限域的列舉），必須有唯一的名稱 `enum`。指定給名稱的值不需要是唯一的。例如，如果不範圍的列舉 `Suit` 宣告如下：

```
enum Suit { Diamonds = 5, Hearts, Clubs = 4, Spades };
```

然後 `Diamonds`、`Hearts`、`Clubs` 和 `Spades` 的值分別為 `5`、`6`、`4` 和 `5`。請注意 `5` 已多次使用，這是允許的行為，即使可能不適合。對於限域範圍列舉，這些規則都相同。

## 轉型規則

不限範圍的列舉常數可以隱含地轉換成 `int`，但是永遠不會隱含地轉換 `int` 為列舉值。下列範例顯示，如果您嘗試將不是 `hand` 的值指派給 `Suit` 時，所發生的狀況：

```

int account_num = 135692;
Suit hand;
hand = account_num; // error C2440: '=' : cannot convert from 'int' to 'Suit'

```

需要進行轉換，才能將轉換成 `int` 限域範圍或不限範圍的列舉值。不過，您可以將不限範圍的列舉值提升為整數值，而不需要轉型。

```
int account_num = Hearts; //OK if Hearts is in a unscoped enum
```

以這種方式使用隱含轉換，可能造成非預期的副作用。為了協助排除與不限範圍的列舉關聯之程式設計錯誤，範圍列舉值是強類型。限域範圍列舉值必須由列舉類型名稱（識別項）來限定，且無法以隱含方式轉換為 `int`，如下列

範例所示：

```
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
    }
}
```

請注意，程式行 `hand = account_num;` 仍然會導致因為不限範圍的列舉而發生的錯誤，如上所示。使用明確轉換時，允許這個行為。不過，使用限定範圍列舉時，不再允許在沒有明確轉型的情況下，於下一個陳述式 `account_num = Suit::Hearts;` 中嘗試轉換。

## 沒有枚舉器的列舉

Visual Studio 2017 15.3 和更新版本(適用于[/std: c + + 17](#))：藉由定義具有明確基礎類型的列舉(一般或範圍)，而不使用任何列舉值，您實際上可以導入不會隱含轉換為任何其他類型的新整數類型。藉由使用這個類型，而不是其內建的基礎類型，您可以消除因不小心隱含轉換而造成的微妙錯誤的可能性。

```
enum class byte : unsigned char { };
```

新的型別是基礎型別的完整複本，因此具有相同的呼叫慣例，這表示它可以跨 Abi 使用，而不會造成任何效能上的負面影響。當使用直接清單初始化來初始化類型的變數時，不需要轉換。下列範例顯示如何在不同的內容中初始化沒有列舉值的列舉：

```
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;
}
```

## 另請參閱

[C 列舉宣告](#)

[關鍵字](#)

# union

2020/11/2 • [Edit Online](#)

## NOTE

在 C++ 17 和更新版本中，`std::variant` class 是的型別安全替代方法 union。

`union` 是使用者定義型別，其中所有成員都共用相同的記憶體位置。這項定義表示，在任何指定的時間，都 `union` 只能在其成員清單中包含一個以上的物件。這也表示無論有多少成員 `union`，一律會使用足夠的記憶體來 儲存最大的成員。

`union` 當您有許多物件和有限的記憶體時，可能會很適合用來節省記憶體。不過，`union` 需要額外小心才能正確 使用。您必須負責確保一律存取您所指派的相同成員。如果任何成員類型具有非一般的 `construct` 或，則您必須 撰寫額外的程式碼，以明確地 `construct` 並終結該成員。在使用之前 `union`，請考慮您嘗試解決的問題，是否可以 使用基底和衍生型別來更妥善地表示 `class`。

## 語法

```
union *** tag * opt { opt * member-list *** };
```

### 參數

`tag`

提供給的型別名稱 `union`。

`member-list`

`union` 可以包含的成員。

## 宣告 `union`

`union` 使用關鍵字來開始的宣告 `union`，並將成員清單括在大括弧中：

```
// 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`

在上述範例中，任何存取的程式碼都 union 需要知道哪些成員會保存資料。此問題最常見的解決方案 union \*稱為「差異」。它會將 union 放在中 struct，並包含 enum 成員來指出目前儲存在中的成員類型 union 。下列範例示範基本模式：

```
#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;
}

```

在上述範例中，union 中的沒有 `Input` struct 名稱，因此稱為 **匿名 union**。您可以直接存取其成員，就像是的成員一樣 struct。如需如何使用匿名的詳細資訊 union，請參閱[匿名 union 區段](#)。

先前的範例顯示您也可以使用 class 衍生自通用基底的型別來解決的問題 class。您可以根據容器中每個物件的執行時間類型，將程式碼分支。您的程式碼可能更容易維護及瞭解，但也可能比使用更慢 union。此外，您也 union 可以使用來儲存不相關的類型。union可讓您以動態方式變更預存值的型別，而不需要變更變數本身的型別 union。例如，您可以建立的異類陣列 `MyUnionType`，其元素會儲存不同類型的不同值。

您可以輕鬆地誤用 `Input` struct 範例中的。使用者可以正確地使用鑑別子來存取保存資料的成員。您可以藉由建立和提供特殊的存取函式來防止誤用 union `private`，如下一個範例所示。

## 不受限制的 union (c + + 11)

在 c + + 03 及更早版本中，union static 只要型別沒有 class 使用者提供 `construct or`、`de struct or` 或指派運算子，就可以包含具有類型的非資料成員。在 C++11 中，已移除這些限制。如果您在中包含這類成員 union，編譯器會自動將非使用者提供的任何特殊成員函式標示為 `deleted`。如果在 union 或中是匿名的 union class struct，則任何不是由使用者提供的特殊成員函式 class struct 都會標示為 `deleted`。下列範例顯示如何處理此案例。的其中一個成員 union 具有需要這項特殊處理的成員：

```

// 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;

```

```

        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無法儲存參考。union也不支援繼承。這表示您不能使用 union 做為基底 class，或繼承自另一個 class，或有虛擬函式。

## 初始化 union

您可以藉 union 由指派以大括弧括住的運算式，在相同的語句中宣告和初始化。運算式會進行評估並指派給的第一個欄位 union。

```

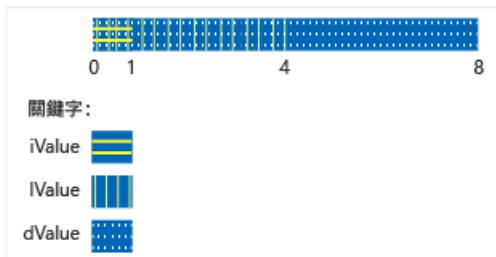
#include <iostream>
using namespace std;

union NumericType
{
    short      iValue;
    long       lValue;
    double     dValue;
};

int main()
{
    union NumericType Values = { 10 }; // iValue = 10
    cout << Values.iValue << endl;
    Values.dValue = 3.1416;
    cout << Values.dValue << endl;
}
/* Output:
10
3.141600
*/

```

NumericType union (在概念上排列)，如下圖所示。



資料儲存在 NumericType union

## 匿名 union

匿名 union 是指沒有或的宣告 `class-name declarator-list`。

<code>union {</code>	<code>member-list</code>	<code>}</code>
----------------------	--------------------------	----------------

匿名中宣告的名稱 union 會直接使用，例如非成員變數。這表示在匿名中宣告的名稱在 union 周圍的範圍中必須是唯一的。

匿名 union 可能會有下列額外的限制：

- 如果在檔案或命名空間範圍中宣告，則也必須將它宣告為 `static`。
- 它只能有 `public` 成員；`private` 或 `protected` 在匿名中具有和成員會 union 產生錯誤。
- 它不能有成員函式。

## 請參閱

[類別和結構](#)

[關鍵字](#)

<code>class</code>
<code>struct</code>

# 函式 (C++)

2020/11/2 • [Edit Online](#)

函式是執行某項作業的程式碼區塊。函式可以選擇性地定義輸入參數，以讓呼叫端將引數傳遞給函式。函式可以選擇性地傳回值做為輸出。函式適用於將一般作業封裝在單一可重複使用的區塊中，而且其名稱最好可清楚描述該函式的作用。下列函式會接受來自呼叫端的兩個整數，並傳回其總和。*a*和*b*是parameters類型的參數 `int`。

```
int sum(int a, int b)
{
    return a + b;
}
```

您可以從程式中任意數目的位置叫用或呼叫函式。傳遞給函式的值是引數，其類型必須與函式定義中的參數類型相容。

```
int main()
{
    int i = sum(10, 32);
    int j = sum(i, 66);
    cout << "The value of j is" << j << endl; // 108
}
```

函式長度沒有實際限制，但良好的設計目標在於函式可以執行單一完整定義的工作。如果可能，應該將複雜演算法分解成容易了解的較簡單函式。

在類別範圍定義的函式稱為成員函式。與其他語言不同，在 C++ 中，函式也可以定義於命名空間範圍（包括隱含全域命名空間）。這類函式稱為免費函數或非成員函式；它們在標準程式庫中廣泛使用。

函式可能會多載，這表示如果函式的不同版本因型式參數的數目和/或類型不同，可能會共用相同的名稱。如需詳細資訊，請參閱[函數多載](#)。

## 函式宣告的組件

最小函式宣告包含傳回類型、函式名稱和參數清單（可能是空的），以及提供其他指示給編譯器的選擇性關鍵字。下列範例是函式宣告：

```
int sum(int a, int b);
```

函式定義包含宣告，加上主體，這是大括弧之間的所有程式碼：

```
int sum(int a, int b)
{
    return a + b;
}
```

後面接著一個分號的函式宣告可能會出現在程式的多個位置。它必須出現在每個轉譯單位中該函式的任何呼叫之前。根據「一個定義規則 (ODR)」，函式定義只能出現在程式中一次。

函式宣告的必要組件如下：

1. 傳回型別，指定函式所傳回之值的型別，`void` 如果沒有傳回值，則為。在 C++ 11 中，`auto` 是有效的傳

回型別，可指示編譯器從 return 語句推斷型別。在 C++14 中，`decltype(auto)` 也允許。如需詳細資訊，請參閱下面的〈傳回型別中的類型推斷〉。

2. 函式名稱，開頭必須為字母或底線，而且不能包含空格。一般而言，標準程式庫函式名稱中的前置底線表示私用成員函式或不是要供您的程式碼使用的非成員函式。
3. 參數清單，是以大括號分隔並以逗號隔開的零或多個參數集合，指定類型以及選擇性指定可用來存取函式主體內值的本機名稱。

函式宣告的選擇性組件如下：

1. `constexpr`，這表示函數的傳回值是常數值，可以在編譯時期計算。

```
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. 其連結規格，`extern` 或 `static`。

```
//Declare printf with C linkage.
extern "C" int printf( const char *fmt, ... );
```

如需詳細資訊，請參閱[轉譯單位和連結](#)。

3. `inline`，指示編譯器將函式的每個呼叫都取代為函式程式碼本身。如果函式執行速度快速並在效能關鍵的程式碼區段中重複予以叫用，則內嵌有助於提高效能。

```
inline double Account::GetBalance()
{
    return balance;
}
```

如需詳細資訊，請參閱[內嵌函數](#)。

4. `noexcept` 運算式，指定函數是否可以擲回例外狀況。在下列範例中，如果運算式評估為，則函數不會擲回例外狀況 `is_pod` `true`。

```
#include <type_traits>

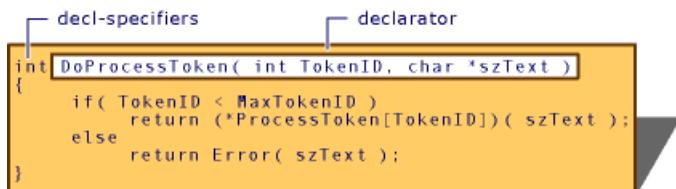
template <typename T>
T copy_object(T& obj) noexcept(std::is_pod<T>) {...}
```

如需詳細資訊，請參閱 [noexcept](#)。

5. (僅限成員函式) Cv 限定詞，指定函數是 `const` 或 `volatile`。
6. (僅限成員函式) `virtual`、`override` 或 `final`。`virtual` 指定可在衍生類別中覆寫函數。`override` 表示衍生類別中的函式會覆寫虛擬函式。`final` 表示無法在任何進一步的衍生類別中覆寫函數。如需詳細資訊，請參閱[虛擬函式](#)。
7. (僅限成員函式) `static` 套用至成員函式表示函式未與類別的任何物件實例相關聯。
8. (僅限非靜態成員函式) Ref-限定詞，指定當隱含物件參數(`*this`)為右值參考，或左值參考時，編譯器所要

選擇的函數多載。如需詳細資訊，請參閱[函數](#)多載。

下圖顯示函式定義的組件。陰影區域是函式主體。



函式定義的部分

## 函式定義

函式定義包含宣告和函式主體，以大括弧括住，其中包含變數宣告、語句和運算式。下列範例顯示完整的函式定義：

```
int foo(int i, std::string s)
{
    int value {i};
    MyClass mc;
    if(strcmp(s, "default") != 0)
    {
        value = mc.do_something(i);
    }
    return value;
}
```

主體內宣告的變數稱為區域變數。它們會在函式結束時消失；因此，函式永遠不應該傳回區域變數的參考！

```
MyClass& boom(int i, std::string s)
{
    int value {i};
    MyClass mc;
    mc.Initialize(i,s);
    return mc;
}
```

## const 和 constexpr 函式

您可以將成員函式宣告為 `const`，以指定不允許函數變更類別中任何資料成員的值。藉由將成員函式宣告為 `const`，您可以協助編譯器強制執行 *const* 正確性。如果有人錯誤地嘗試使用宣告為的函式來修改物件 `const`，則會引發編譯器錯誤。如需詳細資訊，請參閱[const](#)。

宣告函式，就像 `constexpr` 它所產生的值可能會在編譯時期決定一樣。Constexpr 函式的執行速度通常比一般函數快。如需詳細資訊，請參閱 [constexpr](#)。

## 函式樣板

函式樣板與類別樣板類似；它會根據樣板引數產生具象函式。在許多情況下，樣板可以推斷類型引數，因此不需要明確指定它們。

```
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
```

如需詳細資訊，請參閱[函數範本](#)

## 函式參數和引數

函式具有零或多個類型的參數清單 (以逗號分隔)，且各有用來在函式主體內進行存取的名稱。函式樣板可以指定其他類型或值參數。呼叫端會傳遞引數，而引數是類型與參數清單相容的具象值。

根據預設，會以傳值方式將引數傳遞給函式，這表示函式會收到所傳遞物件的複本。針對大型物件，建立複本可能十分耗費資源，而且不一定是必要的。若要讓引數以傳址方式傳遞 (特別是左值參考)，請將參考數量詞新增至參數：

```
void DoSomething(std::string& input){...}
```

函式修改透過傳址方式所傳遞的引數時，會修改原始物件，而不是本機複本。若要防止函數修改這類引數，請將參數限定為 `const&`：

```
void DoSomething(const std::string& input){...}
```

C++ 11：若要明確處理由右值參考或左值參考所傳遞的引數，請在參數上使用雙連字號來表示通用參考：

```
void DoSomething(const std::string&& input){...}
```

`void` 只要關鍵字 `void` 是引數宣告清單的第一個和唯一成員，使用參數宣告清單中的單一關鍵字所宣告的函式就不會採用引數。清單中其他位置的類型引數會 `void` 產生錯誤。例如：

```
// OK same as GetTickCount()
long GetTickCount( void );
```

請注意，雖然 `void` 除了這裡所述之外，指定引數是不合法的，但衍生自類型的類型 `void` (例如 `void` 的指標和陣列 `void`) 可以出現在引數宣告清單的任何位置。

### 預設引數

函式簽章中的最後一個參數或參數可能獲指派預設引數，這表示除非呼叫端想要指定其他值，否則在呼叫函式時可能會省略引數。

```

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)
{...}

```

如需詳細資訊，請參閱[預設引數](#)。

## 函式傳回型別

函式不能傳回另一個函數或內建陣列；不過，它可以傳回這些類型的指標，或是會產生函式物件的`lambda`。除了這些情況之外，函式可能會傳回範圍內任何類型的值，或不會傳回任何值，在此情況下，傳回類型為 `void`。

### 尾端傳回類型

"ordinary" 傳回型別位於函式簽章左邊。`尾端傳回類型`位在簽章的最右邊，且前面加上 `->` 運算子。傳回值的類型取決於樣板參數時，尾端傳回類型特別適用於函式樣板。

```

template<typename Lhs, typename Rhs>
auto Add(const Lhs& lhs, const Rhs& rhs) -> decltype(lhs + rhs)
{
    return lhs + rhs;
}

```

當 `auto` 與尾端傳回型別搭配使用時，它只會當做 `decltype` 運算式所產生的預留位置，而且本身不會執行型別推斷。

## 函式區域變數

在函式主體內宣告的變數稱為「區域變數」(*local variable*)，或簡稱為「區域變數」。非靜態區域變數只顯示於函式主體內，如果它們宣告於堆疊上，則會在函式結束時消失。當您建立本機變數並以傳值方式傳回它時，編譯器通常會執行已命名的傳回值優化，以避免不必要的複製作業。如果您以傳址方式傳回區域變數，則編譯器會發出警告，因為呼叫端使用該參考的任何嘗試都是在終結區域變數之後。

在 C++ 中，區域變數可能宣告為靜態。變數只會顯示在函式主體內，但函式的所有執行個體都有變數的單一複本。區域靜態物件會在 `atexit` 指定的終止時被終結。如果因為程式的控制流程略過其宣告而未建構靜態物件，就不會嘗試終結該物件。

## 傳回類型中的類型推斷(C++14)

在 C++14 中，您可以使用 `auto` 指示編譯器從函式主體推斷傳回型別，而不需要提供尾端傳回型別。請注意，`auto` 一律會會推算為逐值。使用 `auto&&` 指示編譯器推斷參考。

在此範例中，將會推算 `auto` 為 `lhs` 和 `rhs` 總和的非 `const` 值複本。

```

template<typename Lhs, typename Rhs>
auto Add2(const Lhs& lhs, const Rhs& rhs)
{
    return lhs + rhs; //returns a non-const object by value
}

```

請注意，不 `auto` 會保留其所會推算之類型的常數性質。如果轉送函式的傳回值需要保留其引數的常數性質或 `ref` 特性，您可以使用 `decltype(auto)` 關鍵字，這會使用 `decltype` 型別推斷規則並保留所有型別資訊。

`decltype(auto)` 可用來做為左側的一般傳回值，或做為尾端的傳回值。

下列範例(以[N3493](#)的程式碼為基礎)顯示 `decltype(auto)` 如何使用，在範本具現化之前，能夠完美地轉送不知道的傳回型別中的函式引數。

```

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());
}

```

## 從函式傳回多個值

有各種方式可從函式傳回一個以上的值：

1. 封裝已命名的類別或結構物件中的值。要求呼叫者可以看到類別或結構定義：

```

#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. 傳回 `std::元組` 或 `std::p 空中物件`：

```

#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 15.3 和更新版本(適用於 [/std:c++17](#))：使用結構化系結。結構化系結的優點是，儲存傳回值的變數會在宣告時初始化，在某些情況下可能會大幅提高效率。在語句中 `auto[x, y, z] = f();`，方括弧會引進並初始化整個函式區塊範圍內的名稱。

```

#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. 除了使用傳回值本身之外，您還可以藉由定義任意數目的參數來「傳回」值，以使用「傳遞參考」，讓函式可以修改或初始化呼叫者所提供的物件值。如需詳細資訊，請參閱[參考型別函式引數](#)。

## 函式指標

C++ 支援函式指標的方式與 C 語言相同。不過，較具類型安全的替代方案通常是使用函式物件。

如果宣告的函式會傳回函式 `typedef` 指標類型，建議使用來宣告函數指標類型的別名。例如：

```

typedef int (*fp)(int);
fp myFunction(char* s); // function returning function pointer

```

如果沒有這樣做，函式宣告的適當語法可能會從函式指標的宣告子語法推算，方法是將識別項（在上述範例中為 `fp`）取代為函式名稱和引數清單，如下所示：

```

int (*myFunction(char* s))(int);

```

上述宣告相當於使用上述的宣告 `typedef`。

## 另請參閱

[函數多載](#)

[具有變數引數清單的函式](#)

[明確預設和已刪除的函式](#)

[函式上的引數相依名稱\(Koenig\)查閱](#)

[預設引數](#)

[內嵌函數](#)

# 具有變數引數清單的函式 (C++)

2020/11/2 • [Edit Online](#)

最後一個成員是省略符號 (...) 的函式宣告可以接受可變數目的引數。在這些情況下，C++ 只會針對明確宣告的引數提供類型檢查。需要撰寫連引數數目和類型都可以不同的一般函式時，您可以使用變數引數清單。函數系列是使用可變引數清單的函式範例。

| [printf 引數-宣告-清單](#)

## 具有變數引數的函式

若要在宣告之後存取引數，請使用包含在標準 include 檔中的宏，`<stdarg.h>` 如下所述。

### Microsoft 特定的

如果省略符號是最後一個引數，且省略符號在逗號之後，則 Microsoft C++ 允許將省略符號指定為引數。因此，

`int Func( int i, ... );` 告訴是合法的，`int Func( int i ... );` 則不合法。

### 結束 Microsoft 專有

宣告接受可變引數數目的函式至少需要一個預留位置引數 (即使不使用該引數)。如果未提供這個預留位置引數，就無法存取其餘引數。

當類型的引數當做 `char` 變數引數傳遞時，它們會轉換成類型 `int`。同樣地，當類型的引數當做 `float` 變數引數傳遞時，它們會轉換成類型 `double`。其他類型的引數受限於一般整數和浮點數提升。如需詳細資訊，請參閱**標準轉換**。

需要變數清單的函式是使用引數清單中的省略符號 (...) 宣告。使用包含檔案中所述的類型和宏 `<stdarg.h>` 來存取由變數清單所傳遞的引數。如需這些宏的詳細資訊，請參閱[va\\_arg](#)、[va\\_copy](#)、[va\\_end](#)、[va\\_start](#)。(位於 C 執行階段程式庫文件中)。

下列範例顯示宏如何與類型一起使用 (在中宣告 `<stdarg.h>`)：

```
// variable_argument_lists.cpp
#include <stdio.h>
#include <stdarg.h>

// Declaration, but not definition, of ShowVar.
void ShowVar( char *szTypes, ... );
int main() {
    ShowVar( "fcsi", 32.4f, 'a', "Test string", 4 );
}

// ShowVar takes a format string of the form
// "ifcs", where each character specifies the
// type of the argument in that position.
//
// i = int
// f = float
// c = char
// s = string (char *)
//
// Following the format specification is a variable
// list of arguments. Each argument corresponds to
// a format character in the format string to which
// the szTypes parameter points
void ShowVar( char *szTypes, ... ) {
    va_list vl;
    int i;
```

```

// szTypes is the last argument specified; you must access
// all others using the variable-argument macros.
va_start( vl, szTypes );

// Step through the list.
for( i = 0; szTypes[i] != '\0'; ++i ) {
    union Printable_t {
        int      i;
        float    f;
        char     c;
        char    *s;
    } Printable;

    switch( szTypes[i] ) { // Type to expect.
        case 'i':
            Printable.i = va_arg( vl, int );
            printf_s( "%i\n", Printable.i );
            break;

        case 'f':
            Printable.f = va_arg( vl, double );
            printf_s( "%f\n", Printable.f );
            break;

        case 'c':
            Printable.c = va_arg( vl, char );
            printf_s( "%c\n", Printable.c );
            break;

        case 's':
            Printable.s = va_arg( vl, char * );
            printf_s( "%s\n", Printable.s );
            break;

        default:
            break;
    }
}
va_end( vl );
}

//Output:
// 32.400002
// a
// Test string

```

前述範例說明了下列重要概念：

1. 您必須先建立清單標記做為 `va_list` 類型的變數，才能存取任何變數引數。在上述範例中，標記稱為 `vl`。
2. 個別引數是使用 `va_arg` 巨集存取。您必須告知 `va_arg` 巨集要擷取的引數類型，以便從堆疊傳送正確的位元組數目。如果您對 `va_arg` 指定的大小類型不正確，且與呼叫程式所提供的大小不同，則結果將無法預期。
3. 您應該將使用 `va_arg` 巨集取得的結果明確轉型為您需要的類型。

您必須呼叫 `va_end` 巨集終止處理變數引數。

# 函式多載

2020/11/2 • [Edit Online](#)

C++ 允許在相同範圍內指定多個同名的函式。這些函數稱為多載函式。多載函式可讓您針對函式提供不同的語義，視引數的類型和數目而定。

例如，`print` 接受引數的函式 `std::string` 可能會執行非常不同的工作，而不是採用類型之引數的作業 `double`。多載可讓您不需要使用或之類的名稱 `print_string` `print_double`。在編譯時期，編譯器會根據呼叫者傳入的引數類型，選擇要使用的多載。如果您呼叫 `print(42.0)`，則會叫用函式 `void print(double d)`。如果您呼叫 `print("hello world")`，則會叫用多載 `void print(std::string)`。

您可以多載成員函式和非成員函式。下表將說明 C++ 使用函式宣告的哪些部分區分在相同範圍內具有相同名稱的函式群組。

## 多載考量

	?
函式傳回類型	否
引數數目	是
引數類型	是
省略符號是否存在	是
名稱的使用 <code>typedef</code>	否
未指定的陣列範圍	否
<code>const</code> 或** <code>volatile</code>	是，適用於整個函式
<code>Ref</code> 限定詞	是

## 範例

下列範例將示範如何使用多載。

```
// 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();
}

```

上述程式碼示範檔案範圍中 `print` 函式的多載。

預設引數不會視為函式類型的一部分。因此，它不會用於選取多載函式。兩個函式的不同之處只有在其預設引數是視為多重定義而不是多載函式。

無法為多載運算子提供預設引數。

## 引數對應

多載函式會在為了找出目前範圍中最符合函式呼叫所提供之引數的函式宣告項目時選取。如果找到適合的函式，

就會呼叫該函式。在此內容中，「適合」的意義是：

- 找到完全相符項目。
- 已執行一般轉換。
- 已執行整數提升。
- 有轉換成所需引數類型的標準轉換存在。
- 有轉換成所需引數類型的使用者定義轉換(轉換運算子或建構函式)存在。
- 找到省略符號所表示的引數。

編譯器會為每一個引數建立一組候選函式。候選函式是指函式中該位置的實際引數可以轉換成正式引數的類型。

每個引數都會有一組建置的「最相符函式」，而且選取的函式是所有集合的交集。如果交集包含多個函式，則多載會變得模稜兩可並產生錯誤。最後選取的函式對於至少一個引數來說，永遠是比群組中其他每一個函式更理想的相符項目。如果沒有明顯的優勝者，函式呼叫會產生錯誤。

以下列宣告為例(在下面的討論中，函式會標記為 Variant 1、Variant 2 和 Variant 3 以便識別)：

```
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;
```

以下列陳述式為例：

```
F1 = Add( F2, 23 );
```

上面的陳述式會建置兩個集合：

SET 1: FRACTION

Variant 1

Variant 3

SET 2: INT\*\*

Variant 1 ( int 可以 long 使用標準轉換轉換成)

Set 2 中的函式是指有從實際的參數類型隱含轉換成正式參數類型的函數，而在這類函數中，有一個函數可將實際參數類型轉換成其型式參數類型的「成本」為最小值。

這兩個集合的交集為 Variant 1。模稜兩可函式呼叫的範例如下：

```
F1 = Add( 3, 6 );
```

上述函式呼叫會建立下列集合：

SET 1: INT\*\*

Variant 2 ( int 可以 long 使用標準轉換轉換成)

SET 2: INT\*\*

Variant 1 ( int 可以 long 使用標準轉換轉換成)

因為這兩個集合的交集是空的，所以編譯器會產生錯誤訊息。

針對引數比對，具有  $n$  個預設引數的函式會被視為  $n+1$  個不同的函式，每個函數都有不同數目的引數。

省略符號 (...) 做為萬用字元使用，它會比對任何實質引數。如果您不想要特別小心設計多載函式集合，它可能會導致許多不明確的集合。

#### NOTE

在遇到函式呼叫之前，無法判斷多載函式的不明確。遇到函式呼叫時，會為函式呼叫中的每個引數建置集合，如此就可以判斷是否有明確的多載存在。這表示，模稜兩可的情況可能會持續存在程式碼中，直到特定函式呼叫撤銷這些情況為止。

## 引數類型差異

多載函式會區分採用不同初始設定式的各種引數類型。因此，做為多載的用途時，特定類型的引數以及該類型的參考都會視為相同。因為它們會採用相同的初始設定式，所以會將它們視為相同。例如，`max( double, double )` 與 `max( double &, double & )` 會視為相同。同時宣告兩個這類函式會產生錯誤。

基於相同的原因，或所修改之類型的函式引數，其 `const` `volatile` 處理方式不會與多載用途的基底類型不同。

不過，函式多載機制可以區別由和所限定的 `const` 參考 `volatile`，以及基底類型的參考。它可以讓程式碼如下：

```
// 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& ).
```

## 輸出

```
Over default constructor
Over&
Over default constructor
const Over&
Over default constructor
volatile Over&
```

`const` 和物件的指標 `volatile` 也會視為與多載用途的基底類型指標不同。

## 引數比對和轉換

當編譯器嘗試比對實質引數與函式宣告中的引數時，如果找不到完全相符的項目，它可以提供取得正確類型的標準或使用者定義轉換。轉換的應用會受限於下列規則：

- 若轉換包含多個使用者定義的轉換，則不考慮其序列。

- 若轉換可以藉由移除中繼轉換而縮短，則不考慮其序列。

轉換產生的序列(如果有的話)稱為「最相符序列」(Best Matching Sequence)。有數種方式可使用標準轉換將類型的物件轉換 `int` 為類型 `unsigned long` (如[標準轉換](#)中所述)：

- 從轉換 `int` 成 `long`，然後從轉換成 `long` `unsigned long`。
- 從轉換 `int` 成 `unsigned long`。

第一個序列雖然達到所需的目標，但並不是最符合的順序，但仍有較短的順序。

下表將顯示稱為一般轉換 (Trivial Conversion) 的轉換群組，這類轉換對於判斷最符合序列的效果有限。有關一般轉換影響序列選擇的執行個體，將在下表後面的清單中討論。

### 一般轉換

.....	.....
類型-名稱	類型-名稱***&*
類型-名稱***&*	類型-名稱
類型名稱 []	類型-名稱*
類型名稱 ( 引數清單 )	( * 類型名稱 )( 引數清單 )
類型-名稱	* <code>const</code> ***類型-名稱
類型-名稱	* <code>volatile</code> ***類型-名稱
類型-名稱*	* <code>const</code> ***類型-名稱*
類型-名稱*	* <code>volatile</code> ***類型-名稱*

轉換嘗試執行的序列如下：

- 完全相符。呼叫函式所使用的類型與函式原型中所宣告的類型之間完全相符的項目，必定是最相符項目。  
一般轉換的序列會分類為完全相符項目。不過，未進行任何轉換的序列，會被視為優於轉換的序列：

- From 指標，指向 `const` (`type` \* 至 `const type` \*) 的指標。
  - From 指標，指向 `volatile` (`type` \* 至 `volatile type` \*) 的指標。
  - 從參考，到的參考 `const` (`type` & 至 `const type` &)。
  - 從參考，到的參考 `volatile` (`type` & 至 `volatile type` &)。
- 使用提升的相符項目。任何未分類為完全相符的序列，只包含整數提升、從轉換 `float` 為 `double`，以及一般轉換會使用升級分類為相符項。雖然不如任何完全相符項目一般合適，但是使用提升的相符項目與使用標準轉換的相符項目相比之下仍較為理想。
  - 使用標準轉換的相符項目。未分類為完全相符或使用提升的相符項目，而且只包含標準轉換和一般轉換的任何序列，都會分類為使用標準轉換的相符項目。這個分類適用下列規則：
    - 從衍生類別的指標轉換為直接或間接基類的指標，最好是轉換成 `void *` 或 `const void *`。
    - 從指標轉換成衍生類別、轉換成基底類別的指標會產生較佳的相符項目，基底類別也會越接近直接基底類別。假設類別階層架構如下圖所示。



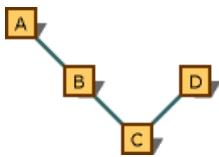
顯示慣用轉換的圖表

從 `D*` 類型轉換成 `C*` 類型，比從 `D*` 類型轉換成 `B*` 類型更理想。同樣地，從 `D*` 類型轉換成 `B*` 類型，比從 `D*` 類型轉換成 `A*` 類型更理想。

相同的規則適用於參考轉換。從 `D&` 類型轉換成 `C&` 類型，比從 `D&` 類型轉換成 `B&` 類型更理想，以此類推。

相同的規則適用於指標至成員轉換。從 `T D::*` 類型轉換成 `T C::*` 類型，比從 `T D::*` 類型轉換成 `T B::*` 類型更理想，以此類推（其中 `T` 是成員的類型）。

前述規則僅適用於所指的衍生路徑。以下圖顯示的圖形為例。



顯示慣用轉換的多重繼承圖形

從 `C*` 類型轉換成 `B*` 類型，比從 `C*` 類型轉換成 `A*` 類型更理想。這是因為它們位於相同路徑上，且 `B*` 較接近。不過，從類型轉換成類型 `C*` `D*` 不是比較理想的轉換成類型 `A*`；因為轉換會遵循不同的路徑，所以沒有任何喜好設定。

1. 使用使用者定義轉換的相符項目 此順序無法分類為完全相符、使用促銷的相符項，或使用標準轉換的相符項。序列必須只包含使用者定義的轉換、標準轉換或一般轉換，才能分類為透過使用者定義轉換的相符項目。透過使用者定義轉換的相符項目通常會比使用省略符號的相符項目更理想，但仍不如使用標準轉換的相符項目理想。
2. 使用省略符號的相符項目。符合宣告中省略符號的任何序列都會分類為使用省略符號的相符項目。這會被視為最弱的相符項。

如果沒有內建的提升或轉換存在，則會套用使用者定義的轉換。這些轉換會依據所比對的引數類型為基礎選取。請考慮下列程式碼：

```
// argument_matching1.cpp
class UDC
{
public:
    operator int()
    {
        return 0;
    }
    operator long();
};

void Print( int i )
{
};

UDC udc;

int main()
{
    Print( udc );
}
```

類別的可用使用者定義轉換 `UDC` 是來自類型 `int` 和類型 `long`。因此，編譯器會考慮所要比對之物件的類型轉換：`UDC`。轉換成 `int exists`，而且已選取。

在比對引數的過程中，標準轉換可以同時套用至引數和使用者定義轉換的結果。因此，下列程式碼可執行：

```
void LogToFile( long l );
...
UDC udc;
LogFile( udc );
```

在上述範例中，會叫用使用者定義的轉換(`operator long`)以轉換 `udc` 成類型 `long`。如果未定義任何使用者定義的類型轉換 `long`，則轉換會如下所示：`UDC int` 使用使用者定義的轉換，將類型轉換成類型。然後，從類型到類型的標準轉換 `int long` 會套用以符合宣告中的引數。

如果需要任何使用者定義的轉換以符合引數，則在評估最符合的項時，不會使用標準轉換。即使有多個候選函數需要使用者定義的轉換，函式也會視為相等。例如：

```
// 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 );
}
```

這兩個版本都 `Func` 需要使用者定義的轉換，才能將類型轉換成 `int` 類別類型引數。可能的轉換包括：

- 從類型轉換 `int` 成類型 `UDC1` (使用者定義的轉換)。
- 從類型轉換成 `int` 類型，`long` 然後轉換成類型 `UDC2` (兩個步驟的轉換)。

雖然第二個需要標準轉換和使用者定義的轉換，但這兩個轉換仍會視為相等。

#### NOTE

使用者定義轉換會視為以建構方式轉換或以初始化方式轉換(轉換函式)。在考慮最符合項目時，這兩種方法會視為相等。

## 引數比對和 this 指標

類別成員函式的處理方式不同，取決於它們是否宣告為 `static`。由於非靜態函式具有提供指標的隱含引數 `this`，因此非靜態函式會被視為具有一個以上的引數，而不是靜態函式，否則會宣告為相同。

這些非靜態成員函式要求隱含的 `this` 指標必須符合呼叫函式所使用的物件類型，或者，如果是多載運算子，則會要求第一個引數必須符合套用運算子的物件。(如需多載運算子的詳細資訊，請參閱多載運算子)。

不同于多載函式中的其他引數，不會引進任何暫存物件，而且在嘗試比對指標引數時，不會嘗試轉換 `this`。

當 `->` 成員選取運算子用來存取類別的成員函式時 `class_name`，`this` 指標引數的類型為 `class_name * const`。如果成員宣告為 `const` 或 `volatile`，則類型 `const class_name * const` 分別為和 `volatile class_name * const`。

成員選取運算子的工作方式完全相同，但有一點除外，就是其物件名稱前方會放置隱含的 `&` (傳址) 運算子。下列範例顯示如何執行這項工作：

```
// Expression encountered in code
obj.name

// How the compiler treats it
(&obj)->name
```

`->*` 和 `.*` (成員的指標) 運算子的左運算元處理方式，與具有相符引數的 `.` 和 `->` (成員選取) 運算子相同。

## 成員函式的 Ref 限定詞

Ref 限定詞可以讓成員函式根據所指向的物件 `this` 是 rvalue 或左值來多載。這項功能可在您選擇不提供資料指標存取的情況下，用來避免不必要的複製作業。例如，假設 class `C` 會初始化其函式中的某些資料，並在成員函式中傳回該資料的複本 `get_data()`。如果類型的物件是即將終結的 `C` 右值，則編譯器會選擇 `get_data() &&` 多載，這會移動資料而不復制它。

```
#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;
}
```

## 多載的限制

有幾項限制負責管理一組可接受的多載函式：

- 一組多載函式中的任兩個函式必須具有不同的引數清單。
- 單獨依據傳回類型來判斷，具有相同類型引數清單的多載函式為錯誤。

### Microsoft 特定的

您可以根據傳回型別來多載operator new，具體而言，是以指定的記憶體模型修飾詞為基礎。

### 結束 Microsoft 專有

- 成員函式不能只多載一個為靜態，而另一個非靜態。
- `typedef` 告訴不會定義新的類型；它們會為現有的類型引進同義字。它們不會影響多載機制。請考慮下列程式碼：

```
typedef char * PSTR;

void Print( char *szToPrint );
void Print( PSTR szToPrint );
```

上述兩個函式擁有相同的引數清單。`PSTR` 是類型的同義字 `char *`。在成員範圍內，這個程式碼會產生錯誤。

- 列舉類型是不同的類型，可以用來區別多載函式。
- 型別「陣列」和「指標」會視為相同，用於區別多載函式，但僅適用於單獨維度陣列。這就是為什麼這些多載函式發生衝突，並產生錯誤訊息：

```
void Print( char *szToPrint );
void Print( char szToPrint[] );
```

對於多維度陣列而言，第二個和後續所有維度都會視為類型的一部分。因此，它們會是用來區別多載函式：

```
void Print( char szToPrint[] );
void Print( char szToPrint[][7] );
void Print( char szToPrint[][9][42] );
```

## 多載、覆寫和隱藏

同一個範圍中任何兩個相同名稱的函式宣告可以參考同一個函式，也可以參考兩個多載的個別函式。如果宣告的引數清單包含相同類型的引數（如先前章節所述），函式宣告會參考相同的函式。否則，它們會參考使用多載選取的兩個不同函式。

嚴格觀察到類別範圍；因此，在基類中宣告的函式與衍生類別中所宣告的函式不在同一個範圍內。如果衍生類別中的函式是使用與基類中的虛擬函式相同的名稱來宣告，衍生類別函式會覆寫基類函數。如需詳細資訊，請參閱[虛擬函式](#)。

如果基類函式未宣告為 'virtual'，則衍生類別函式會被視為隱藏它。覆寫和隱藏都與多載不同。

嚴格觀察到區塊範圍；因此，在檔案範圍中宣告的函式與在本機宣告的函式不在相同的範圍中。如果本機宣告的函式與在檔案範圍中宣告的函式相同名稱，本機宣告的函式會隱藏檔案範圍涵式，而不會引發多載。例如：

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

上述程式碼會顯示函式 `func` 的兩個定義。採用類型引數的定義在 `char *` 本機, `main` 因為 `extern` 語句是。因此, 會隱藏採用類型之引數的定義 `int`, 而對的第一個呼叫 `func` 會發生錯誤。

對於多載成員函式, 可以將不同的存取權限授與不同版本的函式。這些函式仍在封入類別的範圍內, 因此是多載函式。請考慮下列程式碼, 其中的成員函式 `Deposit` 是多載函式; 其中一個是公用版本, 另一個則是私用版本。

這個範例的目的在於提供 `Account` 類別, 在此類別中需要有正確的密碼才能執行儲放。其做法是使用多載。

對的呼叫 `Deposit` 會 `Account::Deposit` 呼叫私用成員函式。這個呼叫是正確的 `Account::Deposit`, 因為是成員函式, 而且可以存取類別的私用成員。

```
// 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;
}
```

## 另請參閱

[函數\(c++\)](#)

# 明確的預設和被刪除的函式

2020/11/2 • [Edit Online](#)

在 C++11 中，預設和已刪除的函式可讓您明確控制是否要自動產生特殊成員函式。被刪除的函式也提供您簡單語言，防止在所有類型函式（特殊成員函式、一般成員函式和非成員函式）的引數中發生有問題的類型提升（原本可能會導致不必要的函式呼叫）。

## 明確預設和已刪除的函式的優點

在 C++ 中，如果類型未自動宣告，編譯器會自動產生預設建構函式、複製建構函式、複製指派運算子和解構函式。這些函數稱為 **特殊成員函式**，而且它們是在 C++ 中讓簡單的使用者定義型別運作的行為，例如結構。也就是說，您可以建立、複製和終結它們，而不需要撰寫任何額外的程式碼。C++11 語言引進移動語意，在編譯器可以自動產生的特殊成員函式清單中，加入移動建構函式和移動指派運算子。

這對簡單類型而言十分方便，但是複雜類型通常會自行定義一個或多個特殊成員函式，而且這可以防止自動產生其他特殊成員函式。實際上：

- 如果明確宣告任何建構函式，則不會自動產生任何預設建構函式。
- 如果明確宣告虛擬解構函式，則不會自動產生任何預設解構函式。
- 如果已明確宣告移動建構函式或移動指派運算子：
  - 不會自動產生複製建構函式。
  - 不會自動產生複製指派運算子。
- 如果已明確宣告複製建構函式、複製指派運算子、移動建構函式、移動指派運算子或解構函式：
  - 不會自動產生移動建構函式。
  - 不會自動產生移動指派運算子。

### NOTE

此外，C++11 標準指定下列額外規則：

- 如果已明確宣告複製建構函式或解構函式，則複製指派運算子自動產生為已被取代。
- 如果已明確宣告複製指派運算子或解構函式，則複製建構函式自動產生為已被取代。

在這兩種情況下，Visual Studio 會繼續自動隱含產生必要函式，且不會發出警告。

這些規則的結果也可能滲入物件階層架構中。例如，如果基類無法從衍生類別呼叫預設的函式（也就是不接受任何參數的或函式），`public` `protected` 則從它衍生的類別就無法自動產生自己的預設函式。

這些規則會讓應該簡單的實作、使用者定義類型和一般 C++ 慣用語更加複雜；例如，私下宣告複製建構函式和複製指派運算子，但未進行定義，以將使用者定義類型設定為不可複製。

```
struct noncopyable
{
    noncopyable() {};

private:
    noncopyable(const noncopyable&);
    noncopyable& operator=(const noncopyable&);
};
```

在 C++11 之前，此程式碼片段是不可複製類型的慣用語表單。不過，它有數個問題：

- 複製的函式必須私下宣告以隱藏它，但因為它完全宣告，所以無法自動產生預設的函式。如果您需要預設建構函式，則必須明確地定義預設建構函式，即使它不執行任何動作也是一樣。
- 即使明確定義的預設建構函式不執行任何動作，編譯器還是會將它視為非一般。其效率比自動產生的預設建構函式還要低，而且會防止 `noncopyable` 變成真正的 POD 類型。
- 即使外部程式碼看不見複製建構函式和複製指派運算子，`noncopyable` 的成員函式和 friend 仍可以看見及呼叫它們。如果宣告但未定義它們，則呼叫它們會導致連結器錯誤。
- 雖然這是普遍接受的慣用語，但是用意不清楚，除非您了解自動產生特殊成員函式的所有規則。

在 C++11 中，`non-copyable` 慣用語可以用更直接的方式來實作。

```
struct noncopyable
{
    noncopyable() =default;
    noncopyable(const noncopyable&) =delete;
    noncopyable& operator=(const noncopyable&) =delete;
};
```

請注意如何解決 C++11 之前的慣用語問題：

- 預設建構函式的產生仍是透過宣告複製建構函式加以避免，但是您可以明確預設以再次產生預設建構函式。
- 明確預設的特殊成員函式仍視為一般，因此不會對效能造成負面影響，而且不會防止 `noncopyable` 成為真正的 POD 類型。
- 複製建構函式和複製指派運算子是公用的，但已遭刪除。它是定義或呼叫已刪除函式的編譯時期錯誤。
- 意圖對所有了解 `=default` 和 `=delete` 的人來說很清楚。您不需要了解自動產生特殊成員函式的規則。

具有類似的慣用語，可產生不可移動、只能動態配置或無法動態配置的使用者定義類型。所有這些慣用語都有 C++11 之前的實作，而這些實作發生類似的問題，而且同樣可在 C++11 中獲得解決，方法是根據預設和已刪除的特殊成員函式來實作它們。

## 明確預設函式

您可以預設任何特殊成員函式：明確陳述特殊成員函式使用預設實作、定義具有非公用存取限定詞的特殊成員函式，或復原在其他情況下無法自動產生的特殊成員函式。

透過宣告，即可預設特殊成員函式宣告（如此範例所示）：

```
struct widget
{
    widget()=default;

    inline widget& operator=(const widget&);

};

inline widget& widget::operator=(const widget&) =default;
```

請注意，您可以在類別的主體外部預設特殊成員函式，只要它是內嵌。

因為一般特殊成員函式的效能優勢，建議您在想要預設行為時偏好使用透過空白函式主體自動產生的特殊成員函式。做法是明確預設特殊成員函式，或不予宣告（也不宣告會防止自動產生它的其他特殊成員函式）。

## 已刪除的函式

您可以刪除特殊成員函式以及一般成員函式和非成員函式，以防止定義或呼叫它們。刪除特殊成員函式可讓編譯器以更清楚的方式產生您不想要的特殊成員函式。函式必須在宣告時被刪除；不能透過可以先宣告再預設函式的方式之後刪除。

```
struct widget
{
    // deleted operator new prevents widget from being dynamically allocated.
    void* operator new(std::size_t) = delete;
};
```

刪除一般成員函式或非成員函式，可防止有問題的類型提升呼叫非預期的函式。這種方式適用，因為已刪除的函式仍參與多載解析，而且提供的相符程度高於提升類型之後可呼叫的函式。函式呼叫解析為更特定、但已刪除的函式，且造成編譯器錯誤。

```
// 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; }
```

請注意，在上述範例中，`call_with_true_double_only` 使用 `float` 引數呼叫會造成編譯器錯誤，但 `call_with_true_double_only` 不會使用 `int` 引數呼叫；在 `int` 這種情況下，引數會從提升 `int` 為 `double`，並成功呼叫函式的 `double` 版本，即使這可能不是預期的情況也一樣。若要使用非雙精確度引數來確保對此函式的任何呼叫都會導致編譯器錯誤，您可以宣告已刪除之函式的範本版本。

```
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.
```

# 函式上的引數相依名稱 (Koenig) 查閱

2020/3/25 • [Edit Online](#)

編譯器可以使用與引數相關的名稱查閱來尋找不合格的函式呼叫的定義。與引數相關的名稱查閱也稱為 Koenig 查閱。函式呼叫中每個引數的類型都是在命名空間、類別、結構、等位或範本階層內定義。當您指定**不合格的**後置函式呼叫時，編譯器會在與每個引數類型相關聯的階層中搜尋函式定義。

## 範例

在這個範例中，編譯器會注意到 `f()` 函式接受 `x` 引數。引數 `x` 的類型是 `A::x`，而該類型是在命名空間 `A` 中定義的。編譯器會搜尋命名空間 `A` 並尋找 `f()` 函式的定義，且此定義接受類型為 `A::x` 的引數。

```
// 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);
}
```

# 預設引數

2020/11/2 • [Edit Online](#)

在許多情況下，函式的引數不常使用，因此使用預設值即已足夠。為解決此問題，預設引數機能只能用於指定在特定呼叫中具有意義之函式的這些引數。為了說明這個概念，請考慮函式多載中所呈現的範例。

```
// 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.
```

許多應用程式可以提供 `prec` 適用的合理預設值，因此不需要兩個函式：

```
// 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.
```

函式的執行 `print` 會稍微變更，以反映只有一個這類函式存在於類型的事實 `double`：

```
// default_arguments.cpp  
// compile with: /EHsc /c  
  
// Print a double in specified precision.  
// Positive numbers for precision indicate how many digits  
// precision after the decimal point to show. Negative  
// numbers for precision indicate where to round the number  
// to the left of the decimal point.  
  
#include <iostream>  
#include <math.h>  
using namespace std;  
  
int print( double dvalue, int prec ) {  
    // Use table-lookup for rounding/truncation.  
    static const double rgPow10[] = {  
        10E-7, 10E-6, 10E-5, 10E-4, 10E-3, 10E-2, 10E-1, 10E0,  
        10E1, 10E2, 10E3, 10E4, 10E5, 10E6  
    };  
    const int iPowZero = 6;  
    // If precision out of range, just print the number.  
    if( prec >= -6 && prec <= 7 )  
        // Scale, truncate, then rescale.  
        dvalue = floor( dvalue / rgPow10[iPowZero - prec] ) *  
            rgPow10[iPowZero - prec];  
    cout << dvalue << endl;  
    return cout.good();  
}
```

若要叫用新的 `print` 函式，請使用如下所示的程式碼：

```
print( d ); // Precision of 2 supplied by default argument.  
print( d, 0 ); // Override default argument to achieve other  
// results.
```

使用預設引數時請注意下列重點：

- 預設引數僅用於忽略結尾引數的函式呼叫，這些引數必須是最後的引數。因此，下列程式碼是不合法的：

```
int print( double dvalue = 0.0, int prec );
```

- 即使重新定義與原始定義完全相同，之後宣告時也不能重新定義預設引數。因此，下列程式碼會產生錯誤：

```
// Prototype for print function.  
int print( double dvalue, int prec = 2 );  
  
...  
  
// Definition for print function.  
int print( double dvalue, int prec = 2 )  
{  
    ...  
}
```

這個程式碼的問題在於定義中的函式宣告會重新定義 `prec` 的預設引數。

- 您可以利用之後的宣告加入其他預設引數。
- 可以對函式指標提供預設引數。例如：

```
int (*pShowIntVal)( int i = 0 );
```

# 內嵌函式 (C++)

2020/11/2 • [Edit Online](#)

在類別宣告的主體中定義的函式是內嵌函式。

## 範例

在下列類別宣告中，`Account` 建構函式是內嵌函式。成員函式 `GetBalance`、`Deposit` 和 `Withdraw` 並未指定為 `inline`，但是可以實作為內嵌函式。

```
// Inline_Member_Functions.cpp
class Account
{
public:
    Account(double initial_balance) { balance = initial_balance; }
    double GetBalance();
    double Deposit( double Amount );
    double Withdraw( double Amount );
private:
    double balance;
};

inline double Account::GetBalance()
{
    return balance;
}

inline double Account::Deposit( double Amount )
{
    return ( balance += Amount );
}

inline double Account::Withdraw( double Amount )
{
    return ( balance -= Amount );
}

int main()
{
```

### NOTE

在類別宣告中，函式是以不含關鍵字的方式宣告 `inline`。`inline` 關鍵字可以在類別宣告中指定，結果會相同。

指定的內嵌成員函式在每個編譯單位中必須以相同方式宣告。這個條件約束會造成內嵌函式的行為如同具現化函式。此外，必須只有一個內嵌函式的定義。

除非該函式的定義包含規範，否則類別成員函式預設為外部連結 `inline`。上述範例顯示您不需要使用規範來明確宣告這些函式 `inline`。`inline` 在函式定義中使用，會使它成為內嵌函數。不過，不允許在呼叫該函式之後重新宣告函式 `inline`。

`inline`、`_inline` 和 `_forceinline`

和規範會指示編譯器將函式 `inline`、`_inline` 主體的複本插入至呼叫函式的每個位置。

只有在編譯器的成本效益分析顯示有意義時，才會發生插入（稱為\*\*內嵌展開或內嵌）。內嵌擴充會以較大的程式碼大小的可能成本，將函式呼叫的額外負荷降到最低。

`_forceinline` 關鍵字會覆寫成本效益分析，並改為依賴程式設計人員的判斷。使用時請小心 `_forceinline`。的任意使用 `_forceinline` 可能會產生較大的程式碼，而且只會提升效能，而在某些情況下，甚至會造成效能損失（例如，較大的可執行檔的分頁增加）。

使用內嵌函式可以讓程式更快速，因為它們不需要與函式呼叫相關聯的負荷。內嵌展開的函式必須遵從不適用於一般函式的程式碼最佳化。

編譯器會將內嵌展開選項和關鍵字視為建議，不保證函式將會內嵌。您不能強制編譯器內嵌特定函式，即使使用關鍵字也一樣 `_forceinline`。使用進行編譯時 `/clr`，如果函式已套用安全性屬性，則編譯器不會內嵌函式。

`inline` 關鍵字僅適用於 C++。`inline` 和 `_forceinline` 關鍵字都可以在 C 和 C++ 中使用。為了與舊版相容，`inline` 和 `_forceinline` 是的同義字。`inline`，`_forceinline` 除非指定了編譯器選項 [[/za \(停用語言擴充功能\)](#)]。

`inline` 關鍵字會告訴編譯器偏好內嵌展開。不過，編譯器可能會建立函式的個別執行個體（具現化）和建立標準呼叫連結，而不是將程式碼內嵌插入。有兩種情況可能會發生這種行為：

- 遞迴函式
- 透過在轉譯單位其他地方的指標所參考的函式。

這些原因可能會干擾內嵌，而不像其他人一樣，也就是編譯器的判斷。您不應該依賴 `inline` 規範來使函式成為內嵌。

如同一般函式，內嵌函數中的引數評估沒有已定義的順序。事實上，使用一般函式呼叫通訊協定傳遞時，它可能會與引數評估順序不同。

[/Ob](#) 編譯器優化選項可協助判斷內嵌函式展開是否確實發生。

[/LTCG](#) 跨模組內嵌是否在原始程式碼中要求。

## 範例 1

```
// inline_keyword1.cpp
// compile with: /c
inline int max( int a , int b ) {
    if( a > b )
        return a;
    return b;
}
```

類別的成員函式可以使用 `inline` 關鍵字或藉由將函式定義放在類別定義內，以內嵌方式宣告。

## 範例 2

```
// inline_keyword2.cpp
// compile with: /EHsc /c
#include <iostream>
using namespace std;

class MyClass {
public:
    void print() { cout << i << ' '; } // Implicitly inline
private:
    int i;
};
```

## Microsoft 特定

`_inline` 關鍵字相當於 `inline`。

即使使用 `_forceinline`，編譯器還是無法在所有情況下內嵌程式碼。編譯器無法在下列情況內嵌函式：

- 函式或其呼叫端是以編譯 `/Ob0` (debug 組建的預設選項)。
- 函式和呼叫端使用不同的例外狀況處理 (一個使用 C++ 例外狀況處理，另一個使用結構化例外狀況處理)。
- 函式具有變數引數清單。
- 除非使用、或進行編譯，否則函式會使用內嵌組解碼 `/Ox` `/O1` `/O2`。
- 函式是遞迴的，而且沒有 `#pragma inline_recursion(on)` 設定。伴隨 pragma，遞迴函式內嵌至預設為 16 個呼叫的深度。若要減少內嵌深度，請使用 `inline_depth` pragma。
- 函式是虛擬的，也是以虛擬方式呼叫。對虛擬函式的直接呼叫可以內嵌。
- 程式使用函式的位址，而且此呼叫是透過函式指標進行的。其位址已取用之函式的直接呼叫可以內嵌。
- 函式也會以修飾詞標記 `naked` `_declspec`。

如果編譯器無法內嵌以宣告的函式 `_forceinline`，它會產生層級1警告，但下列情況除外：

- 函數是使用/Od 或/Ob0. 來編譯 在這些情況下，不需要任何內嵌。
- 函式會在外部、內含的程式庫或其他轉譯單位中定義，或是虛擬呼叫目標或間接呼叫目標。編譯器無法識別在目前轉譯單位中找不到的非內嵌程式碼。

遞迴函式可以用內嵌程式碼取代為 pragma 所指定的深度 `inline_depth`，最多可達16個呼叫。該深度之後，遞迴函式呼叫視為函式執行個體的呼叫。內嵌啟發式檢查遞迴函數的深度不能超過16。`inline_recursion` Pragma 會控制目前在展開下之函式的內嵌展開。如需相關資訊，請參閱[內嵌函式展開](#) (/Ob) 編譯器選項。

## 結束 Microsoft 專有

如需使用規範的詳細資訊 `inline`，請參閱：

- [內嵌類別成員函式](#)
- [使用 `dllexport` 和 `dllimport` 定義內嵌 C++ 函式](#)

## 何時使用內嵌函式

內嵌函式用於小型功能時最為理想，例如存取私用資料成員。這一或兩行的「存取子」函式的主要目的是傳回物件的相關狀態資訊。Short 函數對函式呼叫的額外負荷很敏感。較長的函式在呼叫和傳回序列中會以較少的時間來花費，而不會因內嵌而降低

`Point` 類別可以定義如下：

```

// when_to_use_inline_functions.cpp
class Point
{
public:
    // Define "accessor" functions as
    // reference types.
    unsigned& x();
    unsigned& y();

private:
    unsigned _x;
    unsigned _y;
};

inline unsigned& Point::x()
{
    return _x;
}

inline unsigned& Point::y()
{
    return _y;
}

int main()
{
}

```

假設座標管理在這種類別的用戶端中是相當常見的作業，請指定兩個存取子函式(`x`和`y`上述範例中的)，`inline`通常會將額外負荷儲存在：

- 函式呼叫(包括參數傳遞以及將物件的位址放置在堆疊上)
- 保留呼叫端的堆疊框架
- 新的堆疊框架設定
- 傳回值通訊
- 還原舊的堆疊框架
- 傳回

## 內嵌函式與宏的比較

內嵌函式類似于宏，因為函式程式碼會在編譯時期的呼叫點展開。不過，內嵌函式是由編譯器進行剖析，而宏則是由預處理器展開。因此，兩者有數項重要的差異：

- 內嵌函式會遵循一般功能強制執行的所有類型安全通訊協定。
- 內嵌函式是使用與任何其他函式相同的語法來指定，不同之處在於它們在 `inline` 函式宣告中包含關鍵字。
- 做為引數傳遞至內嵌函式的運算式會評估一次。在某些情況下，做為引數傳遞至巨集的運算式可以多次評估。

下列範例示範將小寫字母轉換為大寫的巨集：

```
// inline_functions_macro.c
#include <stdio.h>
#include <conio.h>

#define toupper(a) ((a) >= 'a' && ((a) <= 'z') ? ((a)-('a'-'A')):(a))

int main() {
    char ch;
    printf_s("Enter a character: ");
    ch = toupper( getc(stdin) );
    printf_s( "%c", ch );
}

// Sample Input: xyz
// Sample Output: Z
```

運算式的目的 `toupper(getc(stdin))` 是應該從主控台裝置()讀取字元，`stdin` 並在必要時轉換成大寫。

由於宏的執行，`getc` 會執行一次，判斷字元是否大於或等於 "a"，以及一次以判斷它是否小於或等於 "z"。如果字元在該範圍內，`getc` 會再次執行，將字元轉換成大寫。這表示程式會等候兩個或三個字元，在理想的情況下，它只應等候一個。

內嵌函式可補救前述問題：

```
// inline_functions_inline.cpp
#include <stdio.h>
#include <conio.h>

inline char toupper( char a ) {
    return ((a >= 'a' && a <= 'z') ? a-('a'-'A') : a);
}

int main() {
    printf_s("Enter a character: ");
    char ch = toupper( getc(stdin) );
    printf_s( "%c", ch );
}
```

Sample Input: a  
Sample Output: A

## 另請參閱

[noinline](#)

[auto\\_inline](#)

# 運算子多載

2020/11/2 • [Edit Online](#)

關鍵字會宣告 `operator` 函式，以指定套用至類別實例時的運算子符號。這讓運算子有多個意義，或稱為「多載」。編譯器會檢查運算元的類型，以區別運算子的不同意義。

## 語法

類型 \* `operator` \*\*\*`operator`-符號 ( 參數清單 )

## 備註

您可以用全域方式或以逐一類別為基礎，重新定義大多數內建運算子的函式。多載運算子實做為函式。

多載運算子的名稱是 `operator` *x*，其中*x*是下表中顯示的運算子。例如，若要多載加號，請定義稱為`operator +` 的函式。同樣地，若要多載加法/指派運算子，`+=` 請定義稱為`operator +=` 的函式。

### 可重新定義的運算子

III	II	I
,	Comma (逗號)	Binary
!	邏輯 NOT	一元 (Unary)
!=	不等	Binary
%	模組	Binary
%=	模數指派	Binary
&	位元 AND	Binary
&	傳址	一元 (Unary)
&&	邏輯 AND	Binary
&=	位元 AND 指派	Binary
()	函式呼叫	—
()	轉換運算子	一元 (Unary)
*	乘	Binary
*	指標取值 (Dereference)	一元 (Unary)
*=	乘法指派	Binary

+	加法	Binary
+	一元加號	一元 (Unary)
++	增量 <sup>1</sup>	一元 (Unary)
+ =	加法指派	Binary
-	減	Binary
-	一元負運算	一元 (Unary)
--	遞減 <sup>1</sup>	一元 (Unary)
- =	減法指派	Binary
- >	成員選取	Binary
-> *	成員指標選取	Binary
/	部門	Binary
/=	除法指派	Binary
<	小於	Binary
<<	左移	Binary
<< =	左移指派	Binary
<=	小於或等於	Binary
=	指派	Binary
==	等式	Binary
>	大於	Binary
> =	大於或等於	Binary
>>	右移	Binary
>> =	右移指派	Binary
[ ]	陣列註標	—
^	互斥 OR	Binary
^ =	互斥 OR 指派	Binary

III	II	I
	位元包含 OR	Binary
=	位元包含 OR 指派	Binary
	邏輯 OR	Binary
~	一補數	一元 (Unary)
<code>delete</code>	刪除	—
<code>new</code>	新增	—
轉換運算子	轉換運算子	一元 (Unary)

<sup>1</sup>兩個版本的一元遞增和遞減運算子存在：前置遞增和後置遞增。

如需詳細資訊，請參閱運算子多載的[一般規則](#)。多載運算子的各種分類限制描述於下列主題：

- [一元運算子](#)
- [二元運算子](#)
- [指派](#)
- [函式呼叫](#)
- [下標](#)
- [類別成員存取](#)
- [遞增和遞減。](#)
- [使用者定義型別轉換](#)

下表中顯示的運算子無法多載。資料表包含預處理器符號 # 和 ##。

### 不可重新定義的運算子

III	II
.	成員選取
. *	成員指標選取
::	範圍解析
? :	條件式
#	前置處理器轉換成字串
##	前置處理器串連

雖然多載運算子通常由編譯器在程式碼中遇到時隱含地呼叫，不過也能像任何成員或非成員函式呼叫一樣地明確叫用：

```
Point pt;
pt.operator+( 3 ); // Call addition operator to add 3 to pt.
```

## 範例

下列範例會多載 + 運算子，以加入兩個複數並傳回結果。

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

struct Complex {
    Complex( double r, double i ) : re(r), im(i) {}
    Complex operator+( Complex &other );
    void Display( ) { cout << re << ", " << im << endl; }
private:
    double re, im;
};

// Operator overloaded using a member function
Complex Complex::operator+( Complex &other ) {
    return Complex( re + other.re, im + other.im );
}

int main() {
    Complex a = Complex( 1.2, 3.4 );
    Complex b = Complex( 5.6, 7.8 );
    Complex c = Complex( 0.0, 0.0 );

    c = a + b;
    c.Display();
}
```

```
6.8, 11.2
```

## 本節內容

- 運算子多載的一般規則
- 多載一元運算子
- 二元運算子
- 指派
- 函式呼叫
- 下標
- 成員存取

## 另請參閱

[C++ 內建運算子、優先順序和順序關聯性](#)  
[關鍵字](#)

# 運算子多載的一般規則

2020/11/2 • [Edit Online](#)

下列規則限制多載運算子的實作方式。不過，它們不會套用至個別涵蓋的new和delete運算子。

- 您不能定義新的運算子，例如`.`。
- 將運算子套用於內建資料類型時，您就無法重新定義運算子的意義。
- 多載運算子必須為非靜態類別成員函式或全域函式。全域函式需要存取私用的或受保護的類別成員，必須宣告為該類別的friend。全域函式必須至少接受一個為類別或列舉類型或為類別或列舉類型參考的引數。  
例如：

```
// 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()
{
}
```

上述程式碼範例將小於運算子宣告為成員函式；不過，加法運算子會宣告為具有friend存取權限的全域函式。請注意，可以對指定運算子提供一個以上的實作。上述加法運算子的範例中，提供了兩個實作以協助交替。就像是將加入至、等等的運算子可能會實作為 `Point Point int Point`。

- 運算子會遵守其優先順序、群組，以及其搭配內建類型使用時指定的運算元數目。因此，沒有任何方法可以表達「將2和3新增至類型的物件」的概念 `Point`，其必須將2加入x座標，而3則會加入至y座標中。
- 宣告為成員函式的一元運算子不接受任何引數；如果宣告為全域函式，則會接受一個引數。
- 宣告為成員函式的二元運算子接受一個引數；如果宣告為全域函式，則會接受兩個引數。
- 如果運算子可以當做一元或二元運算子（`&`、`*`、`+` 和 `-`）使用，您可以分別多載每個使用。
- 多載運算子不可以有預設引數。
- 衍生類別會繼承除了指派(`operator =`)以外的所有多載運算子。
- 成員函式多載運算子的第一個引數，一定是叫用運算子之物件的類別類型(宣告該運算子的類別，或者從該類別衍生的類別)。不會對第一個引數提供任何轉換。

請注意，您可以完全改變任何運算子的意義。其中包括 `address` (`&`)、指派(`=`)和函式呼叫運算子的意義。此外，還可使用運算子多載變更內建類型所依賴的識別。例如，下列四個陳述式若進行完整評估，通常是相等的：

```
var = var + 1;
var += 1;
var++;
++var;
```

多載運算子的類別類型無法依賴這個識別。此外，就多載運算子而言，基本類型使用這些運算子的某些隱含條件比較不嚴謹。例如，當套用至基本類型時，加法/指派運算子 `+=` 會要求左運算元是左值；多載運算子時則沒有這項需求。

**NOTE**

為求一致，最好的作法通常是在定義多載運算子時遵循內建類型的模型。如果多載運算子的語意與其在其他內容中的意義大不相同，可能會比較容易混淆。

## 另請參閱

[運算子多載](#)

# 多載一元運算子

2020/11/2 • [Edit Online](#)

可以多載的一元運算子如下：

1. `!` (邏輯 NOT)

2. `&` (位址)

3. `~` (一補數)

4. `*` (指標取值)

5. `+` (一元加號)

6. `-` (一元否定)

7. `++` (遞增)

8. `--` (遞減)

9. 轉換運算子

後置遞增和遞減運算子 (`++` 和 `--`) 會分別以遞增和遞減的方式處理。

轉換運算子也會在另一個主題中討論；請參閱 [使用者定義型別轉換](#)。

下列規則對於其他所有一元運算子皆成立。若要將一元運算子函式宣告為非靜態成員，您必須以此格式進行宣告：

```
ret-類型 * operator ***op ()
```

其中，`ret` 類型是傳回類型，`op` 則是上表所列的其中一個運算子。

若要將一元運算子函式宣告為全域函式，您必須以此格式進行宣告：

```
ret-類型 * operator ***op ( arg )
```

其中，`ret` 類型和 `op` 是成員運算子函式的描述，而 `arg` 是要在其上操作之類別類型的引數。

## NOTE

一元運算子的傳回型別沒有任何限制。例如，邏輯 NOT (`!`) 傳回整數值是合理的，但不會強制執行。

## 另請參閱

[運算子多載](#)

# 遞增和遞減運算子多載 (C++)

2020/11/2 • [Edit Online](#)

遞增和遞減運算子屬於特殊的分類，因為每個運算子都有兩種變數：

- 前置遞增和後置遞增
- 前置遞減和後置遞減

當您撰寫多載運算子函式時，分別針對這些運算子實作前置和後置的不同版本可能會很有用。為了區分兩者，會觀察到下列規則：運算子的前置格式宣告方式與任何其他一元運算子完全相同；後置形式接受類型為的其他引數 `int`。

## NOTE

為遞增或遞減運算子的後置形式指定多載運算子時，其他引數必須是類型，而 `int` 指定任何其他類型會產生錯誤。

下列範例顯示如何為 `Point` 類別定義前置和後置遞增及遞減運算子：

```

// 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()
{
}

```

相同的運算子可以在檔案範圍 (全域) 中使用下列函式標頭加以定義：

```

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

```

類型的引數 `int`，代表遞增或遞減運算子的後置形式，通常不會用來傳遞引數。它通常包含值 0。不過，可以依如下的方式使用：

```
// increment_and_decrement2.cpp
class Int
{
public:
    Int &operator++( int n );
private:
    int _i;
};

Int& Int::operator++( int n )
{
    if( n != 0 )      // Handle case where an argument is passed.
        _i += n;
    else
        _i++;         // Handle case where no argument is passed.
    return *this;
}
int main()
{
    Int i;
    i.operator++( 25 ); // Increment by 25.
}
```

除了明確的引動過程之外，沒有其他語法會使用遞增或遞減運算子來傳遞這些值，如上述程式碼所示。執行此功能的更簡單方法是多載加法/指派運算子(`+=`)。

## 另請參閱

[運算子多載](#)

# 二元運算子

2020/11/2 • [Edit Online](#)

下表顯示可以多載的運算子清單。

## 可重新定義的二元運算子

```	```
,	Comma (逗號)
!=	不等
%	模組
%=	模數/指派
&	位元 AND
&&	邏輯 AND
&=	位元 AND/指派
*	乘
*=	乘法/指派
+	加法
+ =	加法/指派
-	減
- =	減法/指派
- >	成員選取
- > *	成員指標選取
/	部門
/ =	除法/指派
<	小於
<<	左移
<< =	左移/指派

< =	小於或等於
=	指派
==	等式
>	大於
> =	大於或等於
>>	右移
>> =	右移/指派
^	互斥 OR
^ =	互斥 OR/指派
	位元包含 OR
=	位元包含 OR/指派
	邏輯 OR

若要將二元運算子函式宣告為非靜態成員，您必須以此格式進行宣告：

```
ret-類型 * operator ***op( arg )
```

其中，*ret 類型*是傳回類型，*op*是上表中所列的其中一個運算子，而*arg*是任何類型的引數。

若要將二元運算子函式宣告為全域函式，您必須以此格式進行宣告：

```
ret-類型 operator ***op( arg1**, * arg2 )
```

其中，*ret 類型*和*op*是成員運算子函式的描述，而*arg1*和*arg2*則是引數。至少要有一個引數是類別類型。

#### NOTE

二元運算子的傳回型別不受限制；不過，大部分使用者定義的二元運算子會傳回類別類型或類別類型的參考。

## 另請參閱

[運算子多載](#)

# 指派

2020/3/25 • [Edit Online](#)

指派運算子( = )是嚴格的說，這是二元運算子。它的宣告與任何其他二元運算子相同，但有下列例外狀況：

- 它必須為非靜態成員函式。No **operator =** 可以宣告為非成員函式。
- 衍生類別不會進行繼承。
- 預設的**operator =** 函式可以由編譯器針對類別類型產生(如果沒有的話)。

下列範例說明如何宣告指派運算子：

```
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;
}
```

提供的引數是運算式的右側。運算子會傳回物件以保留指派運算子的行為，而該運算子會在指派完成後傳回左方的值。這可允許指派的連結，例如：

```
pt1 = pt2 = pt3;
```

複製指派運算子不會與複製的處理函式混淆。從現有物件的結構中，會呼叫後者：

```
// Copy constructor is called--not overloaded copy assignment operator!
Point pt3 = pt1;

// The previous initialization is similar to the following:
Point pt4(pt1); // Copy constructor call.
```

## NOTE

建議遵循[三個規則](#)，定義複製指派運算子的類別也應該明確地定義複製的「程式」、「析構函式」和「從 c + + 11 開始」、「移動程式」和「移動指派運算子」。

## 另請參閱

- [運算子多載](#)
- [複製建構函式和複製指派運算子 \(C++\)](#)

# 函式呼叫 (C++)

2020/3/25 • [Edit Online](#)

使用括號叫用的函式呼叫運算子是二元運算子。

## 語法

```
primary-expression ( expression-list )
```

## 備註

在此內容中，`primary-expression` 是第一個運算元，而 `expression-list` (可能空白的引數清單) 是第二個運算元。函式呼叫運算子用於需要一些參數的運算。這不會有問題，因為 `expression-list` 是清單，而不是單一運算元。函式呼叫運算子必須是非靜態成員函式。

函數呼叫運算子在多載時，不會修改呼叫函式的方式；而是會修改運算子在套用到特定類別類型的物件時如何解釋。例如，下列程式碼通常會沒有意義：

```
Point pt;
pt( 3, 2 );
```

但若是有適當多載的函式呼叫運算子，此語法可以用來將 `x` 座標偏移 3 個單位，`y` 座標偏移 2 個單位。下列程式碼表示此定義：

```
// 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 );
}
```

請注意，函式呼叫運算子會套用到物件的名稱，而不是函式的名稱。

您也可以使用函式指標 (而不是函式本身) 來多載函式呼叫運算子。

```
typedef void(*ptf)();
void func()
{
}
struct S
{
    operator ptf()
    {
        return func;
    }
};

int main()
{
    S s;
    s(); //operates as s.operator ptf()()
}
```

## 另請參閱

[運算子多載](#)

# 下標

2020/11/2 • [Edit Online](#)

注標運算子([])就像函式呼叫運算子一樣，會被視為二元運算子。註標運算子必須是使用單一引數的非靜態成員函式。這個引數可以是任何類型，並且指定所需的陣列註標。

## 範例

下列範例示範如何建立類型的向量來 `int` 執行界限檢查：

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

using namespace std;
class IntVector {
public:
    IntVector( int cElements );
    ~IntVector() { delete [] _iElements; }
    int& operator[](int nSubscript);
private:
    int *_iElements;
    int _iUpperBound;
};

// Construct an IntVector.
IntVector::IntVector( int cElements ) {
    _iElements = new int[cElements];
    _iUpperBound = cElements;
}

// Subscript operator for IntVector.
int& IntVector::operator[](int nSubscript) {
    static int iErr = -1;

    if( nSubscript >= 0 && nSubscript < _iUpperBound )
        return _iElements[nSubscript];
    else {
        clog << "Array bounds violation." << endl;
        return iErr;
    }
}

// Test the IntVector class.
int main() {
    IntVector v( 10 );
    int i;

    for( i = 0; i <= 10; ++i )
        v[i] = i;

    v[3] = v[9];

    for ( i = 0; i <= 10; ++i )
        cout << "Element: [" << i << "] = " << v[i] << endl;
}
```

```
Array bounds violation.  
Element: [0] = 0  
Element: [1] = 1  
Element: [2] = 2  
Element: [3] = 9  
Element: [4] = 4  
Element: [5] = 5  
Element: [6] = 6  
Element: [7] = 7  
Element: [8] = 8  
Element: [9] = 9  
Array bounds violation.  
Element: [10] = 10
```

## 註解

在 `i` 上一個程式中達到10時，`operator []` 會偵測到使用超出界限的注標，併發出錯誤訊息。

請注意，函數運算子 `[]` 會傳回參考型別。這會使它變成左值，讓您能夠在指派運算子的任一端使用註標運算式。

## 另請參閱

[運算子多載](#)

# 成員存取

2020/3/25 • [Edit Online](#)

類別成員存取可以藉由多載成員存取運算子 ( -> ) 來加以控制。在這種用法中，這個運算子會視為一元運算子，而多載運算子函式必須是類別成員函式。因此，這類函式的宣告如下：

## 語法

```
class-type *operator->()
```

## 備註

其中，類別類型是此運算子所屬的類別名稱。成員存取運算子函式必須是非靜態成員函式。

這個運算子 (通常會搭配指標取值運算子) 會用來實作「智慧型指標」，這類指標會在取值或計數用法之前驗證指標。

. 成員存取運算子無法多載。

## 另請參閱

[運算子多載](#)

# 類別和結構 (C++)

2020/11/2 • [Edit Online](#)

本節介紹 C++ 類別和結構。這兩個建構在 C++ 中相同，差異在於結構中的預設存取範圍是公用，而類別中的預設值是私用。

類別和結構是可讓您定義專屬類型的建構。類別和結構可以同時包含資料成員和成員函式，以讓您描述類型的狀態和行為。

本文包含下列主題：

- [class](#)
- [結構](#)
- [類別成員總覽](#)
- [成員存取控制](#)
- [繼承](#)
- [靜態成員](#)
- [使用者定義型別轉換](#)
- [可變動的資料成員 \(mutable 規範\)](#)
- [嵌套類別宣告](#)
- [匿名類別類型](#)
- [成員的指標](#)
- [this 指標](#)
- [C++ 位欄位](#)

三個類別類型是結構、類別和等位。它們是使用[struct](#)、[class](#)和[union](#)關鍵字來宣告。下表顯示這三個類別類型的差異。

如需聯集的詳細資訊，[請參閱等位](#)。如需 c++/CLI 和 c++/CX 中類別和結構的詳細資訊，[請參閱類別和結構](#)。

## 結構、類別和等位的存取控制和條件約束

II	II	II
類別機碼為** <code>struct</code> **	類別機碼為** <code>class</code> **	類別機碼為** <code>union</code> **
預設存取權是 public	預設存取權是 private	預設存取權是 public
沒有使用條件約束	沒有使用條件約束	一次只使用一個成員

## 另請參閱

[C++ 語言參考](#)

# class (C++)

2020/11/2 • [Edit Online](#)

關鍵字會宣告 `class` 類別類型或定義類別類型的物件。

## 語法

```
[template-spec]
class [ms-decl-spec] [tag [: base-list ]]
{
    member-list
} [declarators];
[ class ] tag declarators;
```

### 參數

#### 範本-規格

選擇性樣板規格。如需詳細資訊，請參閱 [範本](#)。

#### *class*

`class` 關鍵字。

#### *decl* 規格

選擇性儲存類別規格。如需詳細資訊，請參閱 [\\_declspec](#) 關鍵字。

### 標籤

提供給類別的類型名稱。標記會變成類別範圍內的保留字。標記是選擇項。如果省略，則會定義匿名類別。如需詳細資訊，請參閱 [匿名類別類型](#)。

#### *base-list*

這個類別從中衍生其成員的選擇性類別或結構清單。如需詳細資訊，請參閱 [基底類別](#)。每個基類或結構名稱的前面都可以是存取規範 ([public](#)、[private](#)、[protected](#)) 和 [virtual](#) 關鍵字。如需詳細資訊，請參閱 [控制類別成員存取](#) 的成員訪問表格。

### 成員清單

類別成員的清單。如需詳細資訊，請參閱 [類別成員總覽](#)。

#### *declarators*

宣告子清單，指定類別類型的一個或多個執行個體名稱。如果類別的所有資料成員都是，宣告子可以包含初始化運算式清單 `public`。這在結構中比較常見，其資料成員 `public` 預設是在類別中。如需詳細資訊，請參閱宣告子的 [總覽](#)。

## 備註

一般來說，如需類別的詳細資訊，請參閱下列其中一個主題：

- [結構](#)
- [union](#)
- [\\_multiple\\_inheritance](#)
- [\\_single\\_inheritance](#)
- [\\_virtual\\_inheritance](#)

如需 c++/CLI 和 c++/CX 中 managed 類別和結構的詳細資訊，請參閱[類別和結構](#)。

## 範例

```
// 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;
}
```

## 另請參閱

[關鍵字](#)

[類別和結構](#)

# struct (C++)

2020/11/2 • [Edit Online](#)

關鍵字會定義結構類型 `struct` 和/或結構類型的變數。

## 語法

```
[template-spec] struct [ms-decl-spec] [tag [: base-list ]]
{
    member-list
} [declarators];
[struct] tag declarators;
```

### 參數

#### 範本規格

選擇性樣板規格。如需詳細資訊，請參閱 [範本規格](#)。

#### 結構

`struct` 關鍵字。

#### *decl* 規格

選擇性儲存類別規格。如需詳細資訊，請參閱 [\\_declspec](#) 關鍵字。

#### 標籤

提供給結構的類型名稱。標記會變成結構範圍內的保留字。標記是選擇項。如果省略，則會定義匿名結構。如需詳細資訊，請參閱 [匿名類別類型](#)。

#### *base-list*

這個結構從中衍生其成員的選擇性類別或結構清單。如需詳細資訊，請參閱 [基底類別](#)。每個基類或結構名稱的前面都可以是存取規範 (`public`、`private`、`protected`) 和 `virtual` 關鍵字。如需詳細資訊，請參閱 [控制類別成員存取](#) 的成員訪問表格。

#### 成員清單

結構成員清單。如需詳細資訊，請參閱 [類別成員總覽](#)。唯一的差異在於，它是 `struct` 用來取代 `class`。

#### *declarators*

指定結構名稱的宣告子清單。宣告子清單會宣告結構類型的一個或多個執行個體。如果結構的所有資料成員都是，宣告子可以包含初始化運算式清單 `public`。初始化運算式清單在結構中是常見的，因為資料成員 `public` 預設為。如需詳細資訊，請參閱宣告子的 [總覽](#)。

## 備註

結構類型是使用者定義的複合類型。它是由具有不同類型的欄位或成員所組成。

在 C++ 中，結構與類別相同，不同之處在於它的成員 `public` 預設為。

如需 C++/CLI 中 managed 類別和結構的詳細資訊，請參閱 [類別和結構](#)。

## 使用結構

在 C 中，您必須明確使用 `struct` 關鍵字來宣告結構。在 C++ 中，您不需要在 `struct` 定義型別之後使用關鍵字。

您可以選擇在定義結構類型時宣告變數，方法是將一個或多個逗號分隔的變數名稱放在右大括號和分號之間。

結構變數可以初始化。每個變數的初始化都必須以大括號括住。

如需相關資訊，請參閱 [class](#)、[union](#)和 [enum](#)。

## 範例

```
#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; // 000?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
```

# 類別成員概觀

2020/11/2 • [Edit Online](#)

類別或結構包含其成員。類別所執行的工作是由其成員函式執行。它所維護的狀態會儲存在其資料成員中。成員的初始化是由「函式」完成，而清除工作（例如釋放記憶體和釋出資源）則由「析構函式」完成。在 C++11 和更新版本中，可以（且通常應該）在宣告時初始化資料成員。

## 類別成員的類型

成員分類的完整清單如下：

- [特殊成員函式](#)。
- [成員函式的總覽](#)。
- 包含內建類型和其他使用者定義類型的[資料成員](#)。
- 操作員
- [嵌套類別宣告](#)和。)
- [等位](#)
- [Enumerations](#)列舉。
- [位欄位](#)。
- [朋友](#)。
- [別名和 typedef](#)。

### NOTE

前述清單包含 Friend，是因為它們包含在類別宣告中。不過，它們不是真正的類別成員，因為它們不在類別的範圍內。

## 範例類別宣告

下列範例示範簡單類別宣告：

```

// 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 };

```

## 成員存取範圍

成員清單中所宣告類別的成員。類別的成員清單可以使用稱為存取規範的關鍵字，分割成任意數目的 `private`、`protected` 和 `public` 區段。冒號：必須遵循存取規範。這些區段不需要是連續的，也就是說，這些關鍵字中任何一個都可以在成員清單中出現多次。關鍵字會指定到下一個存取指定名稱，或右大括號之前所有成員的存取權限。如需詳細資訊，請參閱[成員存取控制\(c++\)](#)。

## 靜態成員

資料成員可以宣告為靜態，這表示類別的所有物件都可以存取其相同複本。成員函式可以宣告為靜態，在此情況下，它只能存取類別的靜態資料成員（而且沒有此指標）。如需詳細資訊，請參閱[靜態資料成員](#)。

## 特殊成員函式

特殊成員函式是未在原始程式碼中指定函式時由編譯器自動提供的函式。

1. 預設建構函式
2. 複製建構函式
3. (C++11) 移動構造函式
4. 複製指派運算子

## 5. (C++11) 移動指派運算子

## 6. 解構函式

如需詳細資訊，請參閱[特殊成員函式](#)。

# 成員方式初始化

在 C++11 和更新版本中，非靜態成員宣告子可以包含初始設定式。

```
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()
{
}
```

如果成員在建構函式中獲指派值，該值會覆寫用來在宣告時初始化成員的值。

特定類別類型的所有物件只會有一個共用的靜態資料成員複本。靜態資料成員必須以檔案範圍定義，而且可以依檔案範圍初始化（如需靜態資料成員的詳細資訊，請參閱[靜態資料成員](#)）。下列範例顯示如何執行這些初始化：

```
// class_members2.cpp
class CanInit2
{
public:
    CanInit2() {} // Initializes num to 7 when new objects of type
                  // CanInit are created.
    long num {7};
    static int i;
    static int j;
};

// At file scope:

// i is defined at file scope and initialized to 15.
// The initializer is evaluated in the scope of CanInit.
int CanInit2::i = 15;

// The right side of the initializer is in the scope
// of the object being initialized
int CanInit2::j = i;
```

### NOTE

類別名稱 `CanInit2` 前面必須加上 `i`，才能將所要定義的 `i` 指定為 `CanInit2` 類別的成員。

## 另請參閱

## 類別和結構

# 成員存取控制 (C++)

2020/11/2 • [Edit Online](#)

存取控制可讓您將類別的公用介面與私用執行詳細資料和僅供衍生類別使用的受保護成員分開。除非發現下一個存取規範，否則存取規範會套用至在其後宣告的所有成員。

```
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();
};
```

預設存取是 `private` 在類別中，而 `public` 在結構或等位中。您可以依任意順序使用類別中的存取規範任何次數。類別類型物件的儲存配置依實作而定，不過，成員一定會具有存取指定名稱之間指派的後續較高之記憶體位址。

## 成員存取控制

■■■■	■■
私人	宣告為的類別成員只能 <code>private</code> 由類別的成員函式和朋友(類別或函式)使用。
受保護	宣告為的類別成員可供類別的成員函式 <code>protected</code> 和朋友(類別或函式)使用。此外，類別所衍生的類別也可以使用這些類別成員。
public	宣告為的類別成員 <code>public</code> 可以由任何函數使用。

存取控制有助於避免您誤用物件。執行明確類型轉換(轉型)時，會失去這項保護。

### NOTE

存取控制同樣適用於所有名稱：成員函式、成員資料、巢狀類別及列舉程式。

## 衍生類別中的存取控制

在衍生類別中可存取哪些基底類別的成員是由兩個因素所控制，這些相同的因素可控制在衍生類別中對於繼承成員的存取：

- 衍生類別是否使用存取規範來宣告基類 `public`。

- 要存取哪些基底類別的成員。

下表顯示這些因素之間的互動，以及如何判斷基底類別成員存取。

### 基底類別中的成員存取

PRIVATE	PROTECTED	PUBLIC
永遠無法存取，不論衍生存取為何	如果使用 private 衍生，則可存取衍生類別中的 private 成員	如果使用 private 衍生，則可存取衍生類別中的 private 成員
	如果使用 protected 衍生，則可存取衍生類別中的 protected 成員	如果使用 protected 衍生，則可存取衍生類別中的 protected 成員
	如果使用 public 衍生，則可存取衍生類別中的 Protected 成員	如果使用 public 衍生，則可存取衍生類別中的 Public 成員

下列範例會加以說明：

```
// 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，成員函式 `PublicFunc` 是 public 成員，而 `ProtectedFunc` 是 protected 成員，因為 `BaseClass` 是 public 基底類別。`PrivateFunc` 對 `BaseClass` 而言是 private，且無法存取任何衍生類別。

在 `DerivedClass2` 中，因為 `PublicFunc` 是 private 基底類別，因此會將函式 `ProtectedFunc` 和 `BaseClass` 視

為是 `private` 成員。同樣地，`PrivateFunc` 對 `BaseClass` 而言是 `private`，且無法存取任何衍生類別。

您可以在未使用基底類別存取指定名稱的情況下宣告衍生類別。在這種情況下，如果衍生類別宣告使用關鍵字，則會將衍生視為私用 `class`。如果衍生類別宣告使用關鍵字，則會將衍生視為公用 `struct`。例如，下列程式碼：

```
class Derived : Base  
...
```

相當於：

```
class Derived : private Base  
...
```

同樣地，下列程式碼：

```
struct Derived : Base  
...
```

相當於：

```
struct Derived : public Base  
...
```

請注意，宣告為具有私用存取權的成員無法存取函式或衍生類別，除非使用基類中的宣告來宣告這些函式或類別 `friend`。

`union` 類型不能有基類。

#### NOTE

指定私用基類時，建議明確使用 `private` 關鍵字，讓衍生類別的使用者瞭解成員存取權。

## 存取控制和靜態成員

當您將基類指定為時 `private`，它只會影響非靜態成員。在衍生類別中仍然可以存取公用的靜態成員。不過，使用指標、參考或物件存取基底類別的成員可能需要進行轉換，此時會重新套用存取控制。請考慮下列範例：

```

// access_control.cpp
class Base
{
public:
    int Print();           // Nonstatic member.
    static int CountOf();  // Static member.
};

// Derived1 declares Base as a private base class.
class Derived1 : private Base
{
};

// Derived2 declares Derived1 as a public base class.
class Derived2 : public Derived1
{
    int ShowCount();      // Nonstatic member.
};

// Define ShowCount function for Derived2.
int Derived2::ShowCount()
{
    // Call static member function CountOf explicitly.
    int cCount = Base::CountOf();    // OK.

    // Call static member function CountOf using pointer.
    cCount = this->CountOf(); // C2247. Conversion of
                             // Derived2 * to Base * not
                             // permitted.

    return cCount;
}

```

在上述程式碼中，存取控制項禁止從 `Derived2` 的指標轉換為 `Base` 的指標。`this` 指標的類型是隱含的 `Derived2 *`。若要選取 `CountOf` 函數，`this` 必須將轉換成類型 `Base *`。由於 `Base` 是 `Derived2` 的私用間接基底類別，因此不允許進行這類轉換。只有直接衍生類別的指標可以轉換為私用的基底類別類型。因此，`Derived1 *` 類型的指標可以轉換成 `Base *` 類型。

請注意，明確呼叫 `CountOf` 函式，而不使用指標、參考或物件加以選取，表示沒有進行轉換。因此，可以進行呼叫。

衍生類別的成員和 friend (即 `T`) 可以將 `T` 的指標轉換為 `T` 的私用直接基底類別的指標。

## 存取虛擬函式

套用至虛擬函式的存取控制取決於用來進行函式呼叫的型別。覆寫函式的宣告不會影響特定類型的存取控制。例如：

```

// 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;
}

```

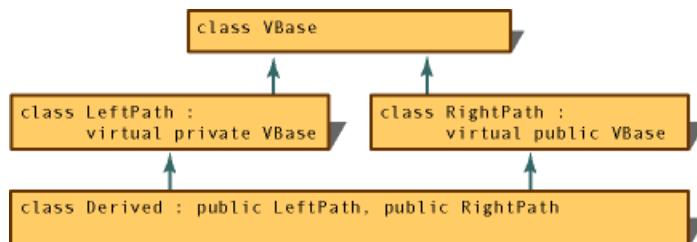
在上述範例中，使用 `GetState` 類型的指標呼叫虛擬函式 `VFuncBase` 會呼叫 `VFuncDerived::GetState`，而 `GetState` 會視為 Public。不過，使用 `GetState` 類型的指標呼叫 `VFuncDerived` 會發生存取控制違規，因為 `GetState` 在 `VFuncDerived` 類別中宣告為 Private。

#### Caution

虛擬函式 `GetState` 可以使用基底類別 `VFuncBase` 的指標呼叫。這並不表示呼叫的函式是該函式的基底類別版本。

## 具有多重繼承的存取控制

在包含虛擬基底類別的多重繼承斜格紋中，可以透過多個路徑存取指定的名稱。由於可以依循這些不同路徑套用不同的存取控制，編譯器會選擇授予較多存取權的路徑。請參閱下圖。



依序繼承圖形的路徑進行存取

在圖中，類別 `VBase` 中宣告的名稱一定會透過類別 `RightPath` 進行存取。正確的路徑會更容易存取，因為 `RightPath` 會將 `VBase` 宣告為公用基底類別，`LeftPath` 則是將 `VBase` 宣告為私用。

## 另請參閱

[C++ 語言參考](#)

# friend (C++)

2020/11/2 • [Edit Online](#)

在某些情況下，將成員層級存取權授與不是類別成員或個別類別中所有成員的函式，會更加方便。只有類別實作器才能宣告它的 friend 是誰。函式或類別不能將它自己宣告為任何類別的 friend。在類別定義中，使用 `friend` 關鍵字和非成員函式或其他類別的名稱，授與類別的私用和受保護成員的存取權。在範本定義中，類型參數可以宣告為 friend。

## 語法

```
class friend F  
friend F;
```

## friend 告白

如果您宣告先前未宣告的 friend 函式，會將該函式匯出至封入 nonclass 範圍。

在 friend 告白中宣告的函式會被視為使用關鍵字宣告的函式 `extern`。如需詳細資訊，請參閱[extern](#)。

雖然可以在全域範圍函式的原型之前將此類函式宣告為 friend，但不可在其完整類別宣告出現之前將成員函式宣告為 friend。下列程式碼示範失敗的原因：

```
class ForwardDeclared; // Class name is known.  
class HasFriends  
{  
    friend int ForwardDeclared::IsAFriend(); // C2039 error expected  
};
```

上述範例在範圍中輸入類別名稱 `ForwardDeclared`，不過，完整宣告（明確地說是宣告函式 `IsAFriend` 的部分）是未知的。因此，類別中的宣告會 `friend HasFriends` 產生錯誤。

從 C++11 開始，類別有兩種形式的 friend 告白：

```
friend class F;  
friend F;
```

第一個表單會引進新的類別 F，如果在最內層的命名空間中找不到該名稱的現有類別。C++11：第二個表單不會引進新的類別；它可以在已宣告類別時使用，而且必須在宣告樣板類型參數或做為 friend 的 `typedef` 時使用。

`class friend F` 當參考的類型尚未宣告時，請使用：

```
namespace NS  
{  
    class M  
    {  
        class friend F; // Introduces F but doesn't define it  
    };  
}
```

```
namespace NS
{
    class M
    {
        friend F; // error C2433: 'NS::F': 'friend' not permitted on data declarations
    };
}
```

在下列範例中，是 `friend F` 指在 `F` NS 範圍外宣告的類別。

```
class F {};
namespace NS
{
    class M
    {
        friend F; // OK
    };
}
```

使用 `friend F` 將範本參數宣告為 friend:

```
template <typename T>
class my_class
{
    friend T;
    //...
};
```

使用 `friend F` 將 `typedef` 告訴為 friend:

```
class Foo {};
typedef Foo F;

class G
{
    friend F; // OK
    friend class F // Error C2371 -- redefinition
};
```

若要宣告兩個類別為彼此的 Friend，必須將第二個類別完整指定為第一個類別的 friend。這項限制的理由是編譯器的資訊只足以在宣告第二個類別的點宣告個別 friend 函式。

#### NOTE

雖然整個第二個類別必須是第一個類別的 friend，但您可以第一個類別中的哪些函式是第二個類別的 friend。

## friend 函式

函式 `friend` 是不是類別成員的函式，但可以存取類別的私用和受保護成員。friend 函式並非類別成員，而是擁有特殊存取權限的一般外部函式。Friend 不在類別的範圍內，而且不是使用成員選取運算子`()`來呼叫。`和->`，除非它們是另一個類別的成員。函式 `friend` 是由授與存取權的類別所宣告。宣告 `friend` 可以放在類別宣告中的任何位置。不會受存取控制關鍵字的影響。

下列範例顯示 `Point` 類別和 friend 函式 `ChangePrivate`。函式可以 `friend` 存取它所接收之物件的私用資料成員，`Point` 做為參數。

```

// 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
}

```

## 做為 friend 的類別成員

類別成員函式可以宣告為其他類別的 friend。請考慮下列範例：

```

// 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

```

上述範例只授與 `A::Func1( B& )` 函式類別 `B` 的 friend 存取權限。因此，對私 `_b` 用成員的存取在類別中是正確的，`Func1` 在中則否 `Func2`。

`friend` 類別是一種類別，其成員函式是類別的 friend 函式，也就是，其成員函式可存取其他類別的私用和受保護成員。假設 `friend` 類別中的宣告 `B` 已經：

```
friend class A;
```

在這種情況下，**A** 類別中的所有成員函式將獲得類別 **B** 的 friend 存取權限。下列程式碼是 friend 類別的範例：

```
// 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;}
};

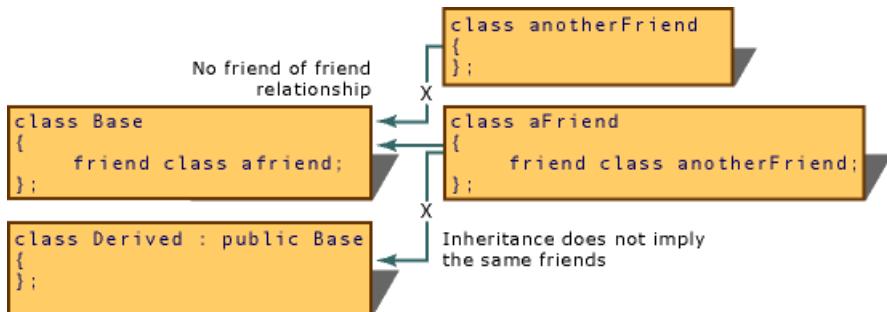
int main() {
    YourClass yc1;
    YourOtherClass yoc1;
    yc1.printMember();
    yoc1.change( yc1, 5 );
    yc1.printMember();
}
```

除非明確指定，否則夥伴關係不是雙向的。在上述範例中，**YourClass** 的成員函式無法存取 **YourOtherClass** 的 private 成員。

Managed 類型(在 c++/CLI 中)不能有任何 friend 函數、friend 類別或 friend 介面。

夥伴關係不能繼承，也就是說，衍生自 **YourOtherClass** 的類別不能存取 **YourClass** 的 private 成員。夥伴關係不可轉移，因此，若類別屬於 **YourOtherClass** 的 friend，就無法存取 **YourClass** 的 private 成員。

下圖說明四種類別宣告：**Base**、**Derived**、**aFriend** 和 **anotherFriend**。只有類別 **aFriend** 可以直接存取 **Base** 的 private 成員(以及 **Base** 可能繼承的任何成員)。



friend 關聯性的含意

## 內嵌 friend 定義

Friend 函式可以在類別宣告內定義(提供函式主體)。這些函式為內嵌函式，而且如同成員內嵌函式一般，它們就像是緊接著在已查看所有類別成員之後，類別範圍關閉之前(類別宣告的結尾)所定義。在類別宣告內定義的 Friend 函式會在封入類別的範圍中。

另請參閱

關鍵字

# private (C++)

2020/11/2 • [Edit Online](#)

## 語法

```
private:  
    [member-list]  
private base-class
```

## 備註

在類別成員清單前面時，`private` 關鍵字會指定只能從成員函式和類別的 friend 存取這些成員。這種做法適用於在下一個存取指定名稱或類別結尾之前宣告的所有成員。

在基類的名稱前面時，`private` 關鍵字會指定基類的 public 和 protected 成員為衍生類別的私用成員。

類別中成員的預設存取方式是 private。結構或等位中成員的預設存取方式是 public。

對於類別而言，基底類別的預設存取方式為 private，對於結構而言則為 public。等位不可以具有基底類別。

如需相關資訊，請參閱[friend](#)、[public](#)、[Protected](#)和[控制對類別成員的存取](#)中的成員存取表格。

## /clr 專屬資訊

在 CLR 類型中，c++ 存取規範關鍵字（`public`、`private` 和 `protected`）可能會影響元件的類型和方法可見度。如需詳細資訊，請參閱[成員存取控制](#)。

### NOTE

以/LN 編譯的檔案不會受到此行為的影響。在這種情況下，所有 Managed 類別 (public 或 private) 都會是可見。

## /clr 專屬資訊結束

## 範例

```
// 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
}
```

## 另請參閱

[控制對類別成員的存取](#)

[關鍵字](#)

# protected (C++)

2020/11/2 • [Edit Online](#)

## 語法

```
protected:  
    [member-list]  
protected base-class
```

## 備註

`protected` 關鍵字會指定成員清單中類別成員的存取權，直到下一個存取規範 (`public` 或 `private`) 或類別定義的結尾。宣告為的類別成員只能 `protected` 由下列各項使用：

- 原本宣告這些成員之類別的成員函式。
- 原本宣告這些成員之類別的 Friend。
- 從原本宣告這些成員的類別中所衍生，具有 `public` 或 `protected` 存取權限的類別。
- 直接以 `private` 方式衍生的類別，這些類別也具有對 `protected` 成員的 `private` 存取權限。

在基類的名稱前面時，`protected` 關鍵字會指定基類的 `public` 和 `protected` 成員是其衍生類別的受保護成員。

受保護的成員不是私用成員，只有在其宣告的 `private` 類別成員才可存取，但它們不是 `public` `public` 成員，而是可在任何函式中存取。

也宣告為的受保護成員可供 `static` 衍生類別的任何 friend 或成員函式存取。未宣告為的受保護成員 `static`，只能透過衍生類別的指標、參考或物件，在衍生類別中存取 friend 和成員函式。

如需相關資訊，請參閱[friend](#)、[public](#)、[Private](#)和[控制對類別成員的存取](#)中的成員存取表格。

## /clr 專屬資訊

在 CLR 類型中，`c++` 存取規範關鍵字 (`public`、`private` 和 `protected`) 可能會影響元件的類型和方法可見度。如需詳細資訊，請參閱[成員存取控制](#)。

### NOTE

以`/LN`編譯的檔案不會受到此行為的影響。在這種情況下，所有 Managed 類別 (`public` 或 `private`) 都會是可見。

## /clr 專屬資訊結束

## 範例

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

using namespace std;
class X {
public:
    void setProtMemb( int i ) { m_protMemb = i; }
    void Display() { cout << m_protMemb << endl; }
protected:
    int m_protMemb;
    void Protfunc() { cout << "\nAccess allowed\n"; }
} x;

class Y : public X {
public:
    void useProtfunc() { Protfunc(); }
} y;

int main() {
    // x.m_protMemb;      error, m_protMemb is protected
    x.setProtMemb( 0 ); // OK, uses public access function
    x.Display();
    y.setProtMemb( 5 ); // OK, uses public access function
    y.Display();
    // x.Protfunc();      error, Protfunc() is protected
    y.useProtfunc();    // OK, uses public access function
                        // in derived class
}
```

## 另請參閱

[控制對類別成員的存取](#)

[關鍵字](#)

# public (C++)

2020/11/2 • [Edit Online](#)

## 語法

```
public:  
    [member-list]  
public base-class
```

## 備註

在類別成員清單前面時，`public` 關鍵字會指定可以從任何函式存取這些成員。這種做法適用於在下一個存取指定名稱或類別結尾之前宣告的所有成員。

在基類的名稱前面，`public` 關鍵字會指定基類的 `public` 和 `protected` 成員分別為衍生類別的 `public` 和 `protected` 成員。

類別中成員的預設存取方式是 `private`。結構或等位中成員的預設存取方式是 `public`。

對於類別而言，基底類別的預設存取方式為 `private`，對於結構而言則為 `public`。等位不可以具有基底類別。

如需詳細資訊，請參閱[私用、受保護、Friend和控制對類別成員的存取](#)中的成員存取表格。

## /clr 專屬資訊

在 CLR 類型中，`c++` 存取規範關鍵字（`public`、`private` 和 `protected`）可能會影響元件的類型和方法可見度。如需詳細資訊，請參閱[成員存取控制](#)。

### NOTE

以`/LN`編譯的檔案不會受到此行為的影響。在這種情況下，所有 Managed 類別 (`public` 或 `private`) 都會是可見。

## /clr 專屬資訊結束

## 範例

```
// 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
}
```

## 另請參閱

[控制對類別成員的存取](#)

[關鍵字](#)

# 大括弧初始化

2020/11/2 • [Edit Online](#)

您不一定需要定義類別的建構函式，尤其是相當簡單的類別。使用者可以使用統一初始化來初始化類別或結構的物件（如下列範例所示）：

```
// no_constructor.cpp
// Compile with: cl /EHsc no_constructor.cpp
#include <time.h>

// No constructor
struct TempData
{
    int StationId;
    time_t timeSet;
    double current;
    double maxTemp;
    double minTemp;
};

// Has a constructor
struct TempData2
{
    TempData2(double minimum, double maximum, double cur, int id, time_t t) :
        stationId{id}, timeSet{t}, current{cur}, maxTemp{maximum}, minTemp{minimum} {}
    int stationId;
    time_t timeSet;
    double current;
    double maxTemp;
    double minTemp;
};

int main()
{
    time_t time_to_set;

    // Member initialization (in order of declaration):
    TempData td{ 45978, time(&time_to_set), 28.9, 37.0, 16.7 };

    // Default initialization = {0,0,0,0,0}
    TempData td_default{};

    // Uninitialized = if used, emits warning C4700 uninitialized local variable
    TempData td_noInit;

    // Member declaration (in order of ctor parameters)
    TempData2 td2{ 16.7, 37.0, 28.9, 45978, time(&time_to_set) };

    return 0;
}
```

請注意，當類別或結構沒有任何函式時，您會以類別中宣告成員的順序提供清單元素。如果類別有一個函式，請以參數的順序提供元素。如果類型具有隱含或明確宣告的預設的處理常式，您可以使用預設的大括弧初始化（含空的大括弧）。例如，您可以使用預設和非預設的大括弧初始化來初始化下列類別：

```

#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);
}

```

如果類別具有非預設的處理常式，則類別成員在大括弧初始化運算式中出現的順序，就是對應參數出現在函式中的順序，而不是宣告成員的順序（如同 `class_a` 上一個範例中的）。否則，如果類型沒有宣告的函式，成員在大括弧初始化運算式中出現的順序會與宣告的順序相同。在此情況下，您可以視需要初始化多個公用成員，但不能略過任何成員。下列範例顯示沒有宣告的函式時，在大括弧初始化中使用的順序：

```

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
}

```

如果預設的函式已明確宣告，但標示為已刪除，則無法使用預設的大括弧初始化：

```

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
}

```

您可以在一般執行初始化的任何地方使用括弧初始化，例如，做為函式參數或傳回值，或使用 `new` 關鍵字：

```
class_d* cf = new class_d{4.5};  
kr->add_d({ 4.5 });  
return { 4.5 };
```

在 /std: c + + 17 模式中，空白大括弧初始化的規則會略有限制。請參閱[衍生的函數和擴充的匯總初始化](#)。

## initializer\_list 的構造函式

`Initializer_list` 類別代表指定類型的物件清單，可用於函數和其他內容中。您可以使用大括弧初始化來建立 `initializer_list`：

```
initializer_list<int> int_list{5, 6, 7};
```

### IMPORTANT

若要使用這個類別，您必須包含 `<initializer_list>` 標頭。

`initializer_list` 可以複製。在此情況下，新清單的成員會參考原始清單的成員：

```
initializer_list<int> ilist1{ 5, 6, 7 };  
initializer_list<int> ilist2( ilist1 );  
if (ilist1.begin() == ilist2.begin())  
    cout << "yes" << endl; // expect "yes"
```

標準程式庫容器類別，以及、`string` `wstring` 和 `regex` 都有一個 `initializer_list` 函數。下列範例示範如何使用這些函式進行括弧初始化：

```
vector<int> v1{ 9, 10, 11 };  
map<int, string> m1{ {1, "a"}, {2, "b"} } ;  
string s{ 'a', 'b', 'c' } ;  
regex rgx{ 'x', 'y', 'z' } ;
```

## 另請參閱

[類別和結構](#)

[建構函式](#)

# 物件存留期及資源管理 (RAII)

2020/3/21 • [Edit Online](#)

不同于 managed 語言 C++，並不會自動進行垃圾收集。這是內部進程，會在程式執行時釋放堆積記憶體和其他資源。C++ 程式會負責將所有取得的資源傳回作業系統。無法釋放未使用的資源稱為「流失」。在進程結束之前，其他程式無法使用流失的資源。記憶體流失特別是 C 樣式程式設計中錯誤的常見原因。

新式 C++ 可避免在堆疊上宣告物件，盡可能使用堆積記憶體。當資源對堆疊而言太大時，它應該會由物件所擁有。當物件初始化時，它會取得它所擁有的資源。物件接著會負責釋放其析構函式中的資源。擁有物件本身會在堆疊上宣告。物件本身資源的原則也稱為「資源取得初始化」或 RAII。

當資源擁有堆疊物件超出範圍時，就會自動叫用其析構函式。如此一來，中 C++ 的垃圾收集與物件存留期緊密相關，而且具決定性。資源一律會在程式中的已知點釋放，您可以加以控制。只有像 C++ 中的那些具決定性的解構函式能均衡處理記憶體和非記憶體資源。

下列範例顯示簡單的物件 `w`。它會在函式範圍的堆疊上宣告，並在函式區塊的結尾終結。物件 `w` 不會擁有任何資源(例如堆積配置的記憶體)。其唯一的成員 `g` 本身會在堆疊上宣告，而且只會隨著 `w` 而超出範圍。`widget` 的析構函式中不需要任何特殊的程式碼。

```
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(); }"
```

在下列範例中，`w` 擁有記憶體資源，因此在其析構函式中必須有程式碼來刪除記憶體。

```
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 開始，有更好的方法可以撰寫先前的範例：使用標準程式庫中的智慧型指標。智慧型指標會處理所擁有之記憶體的配置和刪除。使用智慧型指標，就不需要 `widget` 類別中的明確析構函式。

```
#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
```

藉由使用智慧型指標進行記憶體配置，您可以消除記憶體流失的可能性。此模型適用於其他資源，例如檔案控制代碼或通訊端。您可以在類別中以類似的方式管理您自己的資源。如需詳細資訊，請參閱[智慧型指標](#)。

的設計 C++ 可確保物件在超出範圍時終結。也就是說，它們會被終結，因為區塊會以相反的結構順序結束。在物件終結時，它的基底和成員會以特殊的順序終結。在全域範圍的任何區塊外宣告的物件，可能會導致問題。如果全域物件的函式擲回例外狀況，可能會很難以進行 debug。

## 另請參閱

[歡迎回到 C++](#)

[C++ 語言參考](#)

[C++ 標準程式庫](#)

# 編譯時期封裝的 Pimpl (現代 C++)

2019/12/2 • [Edit Online](#)

*Pimp*的方法是用來C++隱藏執行的現代化技術，可將結合性降至最低，以及分隔介面。Pimpl是「執行指標」的簡短。您可能已經熟悉此概念，但可透過其他名稱(例如 Cheshire Cat 或編譯器防火牆)來加以瞭解。

## 為何要使用 pimpl？

以下是 pimpl 的方法可以改善軟體發展生命週期的方式：

- 編譯相依性的小化。
- 區隔介面和實作為區隔。
- 相容。

## Pimpl 標頭

```
// my_class.h
class my_class {
    // ... all public and protected stuff goes here ...
private:
    class impl; unique_ptr<impl> pimpl; // opaque type here
};
```

Pimpl 的用法可避免重建級聯和脆弱物件版面配置。它非常適合(可轉移)的熱門類型。

## Pimpl 執行

定義 .cpp 檔案中的 `impl` 類別。

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

## 最佳作法

請考慮是否要新增對非擲回交換特製化的支援。

## 另請參閱

[歡迎回到C++](#)

[C++ 語言參考](#)

[C++ 標準程式庫](#)

# ABI 界限的可攜性

2019/12/2 • [Edit Online](#)

在二進位介面界限使用充分的可攜性類型和慣例。「可移植類型」是一種 C 內建類型，或僅包含 C 內建類型的結構。只有當呼叫端和被呼叫端同意版面配置、呼叫慣例等時，才可以使用類別類型。只有當使用相同的編譯器和編譯器設定來編譯時，才有可能發生這種情況。

## 如何簡化 C 可攜性的類別

當呼叫端可以使用另一個編譯器/語言進行編譯時，請使用特定的呼叫慣例將「簡維」為extern "C" API：

```
// 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"  
}
```

## 另請參閱

[歡迎回到C++](#)

[C++ 語言參考](#)

[C++ 標準程式庫](#)

# 建構函式 (C++)

2020/11/2 • [Edit Online](#)

若要自訂類別成員的初始化方式，或在建立類別的物件時叫用函數，請定義一個函式。建構函式的名稱與類別的名稱相同，但沒有傳回值。您可以視需要定義任意數量的多載的函式，以各種方式自訂初始化。一般而言，這些函式具有公用存取範圍，因此類別定義或繼承階層之外的程式碼可以建立類別的物件。但是，您也可以將函式宣告為 `protected` 或 `private`。

您可以選擇性地採用成員 init 清單。這是比起在函式主體中指派值更有效率的方式，來初始化類別成員。下列範例顯示具有三個多載的函式的類別 `Box`。最後兩個使用成員 init 清單：

```
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 };  
};
```

當您宣告類別的實例時，編譯器會根據多載解析的規則，選擇要叫用的函式：

```
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)  
}
```

- 函式可以宣告為 `inline`、`explicit`、`friend` 或 `constexpr`。
- 函式可以初始化已宣告為、或的物件 `const`、`volatile`、`const volatile`。在完成函式之後，物件就會變成 `const`。
- 若要在實檔案中定義一個函式，請提供它一個限定名稱，如同其他任何成員函式：`Box::Box(){...}`。

## 成員初始化運算式清單

函式可以選擇性地擁有成員初始化運算式清單，其會在執行函式主體之前初始化類別成員。(請注意，成員初始化運算式清單與 `std:: initializer_list <T>` 類型的 [初始化運算式清單](#) 不同。)

使用成員初始化運算式清單時，最好不要在函式主體中指派值，因為它會直接初始化成員。在下列範例中，會顯示成員初始化運算式清單是由所有 [識別碼 \(引數\)](#) 在冒號之後的運算式所組成：

```
Box(int width, int length, int height)
    : m_width(width), m_length(length), m_height(height)
{}
```

識別碼必須參考類別成員；它會使用引數的值進行初始化。引數可以是其中一個函式參數、函式呼叫或 `std:: initializer_list <T>`。

`const` 參考型別的成員和成員必須在成員初始化運算式清單中初始化。

在初始化運算式清單中，應該在初始化運算式清單中建立參數化基類的函式的呼叫，以確保在執行衍生的函式之前，會完整初始化基類。

## 預設的函式

預設的函式通常沒有任何參數，但它們可以有具有預設值的參數。

```
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){}
    ...
}
```

預設的函式是其中一個 [特殊成員](#) 函式。如果在類別中未宣告任何的函式，編譯器會提供隱含的預設函式 `inline`。

```
#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
}
```

如果您依賴隱含預設的函式，請務必在類別定義中初始化成員，如先前的範例所示。如果沒有這些初始化運算式，成員就會被初始化，而磁片區 (# A1 呼叫) 會產生垃圾值。一般來說，即使不依賴隱含預設的函式，也能以這種方式初始化成員是不錯的作法。

您可以藉由將它定義為 [已刪除](#) 來防止編譯器產生隱含預設的函式：

```
// Default constructor  
Box() = delete;
```

如果任何類別成員不是預設的可建構，則編譯器產生的預設函式將會定義為 deleted。例如，類別型別的所有成員及其類別型別成員都必須有可存取的預設函式和析構函數。所有參考型別的資料成員以及 `const` 成員都必須有預設成員初始化運算式。

當您呼叫編譯器產生的預設的函式，並嘗試使用括弧時，會發出警告：

```
class myclass{};  
int main(){  
    myclass mc(); // warning C4930: prototyped function not called (was a variable definition intended?)  
}
```

這是「最令人惱怒的語法解析」(Most Vexing Parse) 問題範例。由於範例運算式可解譯為函式的宣告或做為預設建構函式的引動過程，而且由於 C++ 剖析器偏好宣告更勝於其他項目，因此運算式被視為函式宣告。如需詳細資訊，請參閱 [大部分的繁瑣語法剖析](#)。

如果已宣告任何非預設建構函式，編譯器不會提供預設建構函式：

```
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  
}
```

如果類別沒有預設建構函式，該類別的物件陣列無法透過單獨使用方括號語法來建構。例如，根據上述程式碼區塊，Boxes 陣列無法宣告如下：

```
Box boxes[3]; // C2512: no appropriate default constructor available
```

不過，您可以使用一組初始化運算式清單來初始化 Box 物件的陣列：

```
Box boxes[3]{ { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };
```

如需詳細資訊，請參閱 [初始化運算式](#)。

## 複製函式

複製函式會藉由複製相同類型之物件的成員值來初始化物件。如果您的類別成員都是簡單類型(例如純量值)，則編譯器產生的複製函式就已足夠，您也不需要自行定義。如果您的類別需要更複雜的初始化，則您需要執行自訂複製的函式。例如，如果類別成員是指標，則您需要定義複製的函式來配置新的記憶體，並複製另一個指向物件的值。編譯器產生的複製函式只會複製指標，讓新的指標仍然指向其他的記憶體位置。

複製的函式可能會有下列其中一種簽章：

```
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");
```

當您定義複製的函式時，您也應該定義複製指派運算子 (=)。如需詳細資訊，請參閱 [指派](#) 和複製函式 [和複製指派運算子](#)。

您可以藉由將複製的方法定義為已刪除，來防止複製您的物件：

```
Box (const Box& other) = delete;
```

嘗試複製物件時，會產生錯誤 C2280: 嘗試參考已刪除的函式。

## 移動函數

移動函式是特殊成員函式，可將現有物件資料的擁有權移至新的變數，而不會複製原始資料。它會採用右值參考做為第一個參數，而任何其他參數都必須有預設值。移動函式在傳遞大型物件時，可能會大幅增加程式的效率。

```
Box(Box&& other);
```

編譯器會在某些情況下，選擇移動的函式，其中物件是由相同類型的另一個物件初始化，而該物件是即將終結且不再需要它的資源。下列範例顯示使用多載解析選取移動函式時的一個案例。在呼叫的函 `get_Box()` 式中，傳回的值為 `Xvalue`(到期) 的值。它不會指派給任何變數，因此即將移出範圍。為了提供此範例的動機，讓我們為 `Box` 提供代表其內容的大型字串向量。移動函式不會複製向量和其字串，而是從過期值 "box" 中「竊取」，讓向量現在屬於新的物件。的呼叫 `std::move` 是必要的，因為 `vector` 和 `string` 類別會執行自己的移動函式。

```

#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;
}

```

如果類別未定義移動函式，則編譯器會在沒有使用者宣告的複製函式、複製指派運算子、移動指派運算子或自訂函數時產生隱含的函式。如果未定義明確或隱含移動的函式，則會改為使用複製函式的作業。如果類別宣告移動函式或移動指派運算子，則會將隱含宣告的複製函式定義為已刪除。

如果任何屬於類別類型的成員缺少任何一個函式，或編譯器無法判斷要用於移動作業的哪一個函式，則會將隱含宣告的移動函式定義為已刪除。

如需如何撰寫非一般移動函式的詳細資訊，請參閱 [移動函式和移動指派運算子 \(C++\)](#)。

## 明確預設和已刪除的函式

您可以明確 **預設** 的複製函式、預設的函式、移動的函式、複製指派運算子、移動指派運算子和析構函數。您可以明確地 **刪除** 所有特殊成員函式。

```
class Box
{
public:
    Box2() = delete;
    Box2(const Box2& other) = default;
    Box2& operator=(const Box2& other) = default;
    Box2(Box2&& other) = default;
    Box2& operator=(Box2&& other) = default;
    //...
};
```

如需詳細資訊，請參閱 [明確預設和已刪除的函式](#)。

## constexpr 函式

若為，則可將函式宣告為 **constexpr**

- 它會宣告為預設值，否則會滿足所有 **constexpr** 函式的條件（一般）；
- 類別沒有任何虛擬基類；
- 每個參數都是 **常數值型別**；
- 主體不是函數 try 區塊；
- 所有非靜態資料成員和基類子物件都會初始化；
- 如果類別 () 具有 variant 成員的等位，或 (b) 具有名聯集，則只會初始化其中一個等位成員；
- 類別類型的每個非靜態資料成員，以及所有基類子物件都有 **constexpr** 的函式

## 初始化運算式清單的函式

如果函式接受 `std::initializer_list <T>` 做為其參數，而且任何其他參數都有預設引數，則會在透過直接初始化來具現化類別時，于多載解析中選取該函式。您可以使用 `initializer_list` 來初始化任何可以接受它的成員。例如，假設先前的 `Box` 類別（有 `std::vector<string>` 成員 `m_contents`）。您可以提供如下的函數：

```
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)
{}
```

然後建立如下的 `Box` 物件：

```
Box b{ "apples", "oranges", "pears" }; // or ...
Box b2(initializer_list<string> { "bread", "cheese", "wine" }, 2, 4, 6);
```

## 明確的函式

如果類別的建構函式具有單一參數，或者，所有參數（但其中一個除外）都有預設值，則參數類型可以隱含地轉換為類別類型。例如，如果 `Box` 類別具有建構函式，如下：

```
Box(int size): m_width(size), m_length(size), m_height(size){}
```

Box 可能初始化如下：

```
Box b = 42;
```

或將 int 傳遞給採用 Box 的函式：

```
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);
```

在某些情況下，這類轉換十分有用，但它們可能更常導致您程式碼中的細微但嚴重的錯誤。一般來說，您應該 **explicit** 在 (的函式和使用者定義的運算子) 上使用關鍵字，以防止這種類型的隱含類型轉換：

```
explicit Box(int size): m_width(size), m_length(size), m_height(size){}
```

建構函式是明確建構函式時，此行會造成編譯器錯誤：`ShippingOrder so(42, 10.8);`。如需詳細資訊，請參閱 [使用者定義型別轉換](#)。

## 結構的順序

建構函式會依此順序執行其工作：

1. 它會依宣告順序呼叫基底類別和成員建構函式。
2. 如果類別是從虛擬基底類別衍生，它會初始化物件的虛擬基底指標。
3. 如果類別具有或繼承虛擬函式，它會初始化物件的虛擬函式指標。虛擬函式指標指向類別的虛擬函式表，以便讓虛擬函式呼叫正確繫結至程式碼。
4. 它會執行其函式主體內的任何程式碼。

下列範例顯示在衍生類別的建構函式中呼叫基底類別和成員建構函式的順序。首先會呼叫基底建構函式，然後依其出現在類別宣告中的順序初始化基底類別成員，最後會呼叫衍生的建構函式。

```

#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;
}

```

輸出如下：

```

Contained1 ctor
Contained2 ctor
BaseContainer ctor
Contained3 ctor
DerivedContainer ctor

```

衍生類別建構函式一定會呼叫基底類別建構函式，因此，它可以依賴完全建構的基底類別，才進行任何額外的工作。基類的函式會以衍生的順序來呼叫，例如，如果衍生自衍生自 ClassA ClassB 的，則會 ClassC 先呼叫此函式 ClassC，然後再呼叫函式 ClassB，然後再呼叫 ClassA 函式。

如果基底類別沒有預設建構函式，您必須在衍生類別建構函式中提供基底類別建構函式參數：

```

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);
}

```

如果建構函式擲回例外狀況，解構順序是建構順序的相反：

1. 在建構函式主體中的程式碼會回溯。
2. 基底類別和成員物件會依宣告的反向順序終結。
3. 如果建構函式為非委派，所有完全建構的基底類別物件和成員都會終結。不過，因為物件本身未完全建構，所以不會執行解構函式。

## 衍生的函式和延伸匯總初始化

如果基類的函式不是公用的，但可供衍生類別存取，則在 /std: c + + 17 模式的 Visual Studio 2017 和更新版本中，您無法使用空的大括弧來初始化衍生類型的物件。

下列範例顯示 C + + 14 一致性行為：

```

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 中，`Derived` 已視作彙總類型。因此，透過私用預設建構函式將 `Base` 初始化會直接包含在擴充彙總初始化規則的過程。`Base` 私用建構函式在先前會透過 `Derived` 建構函式呼叫，而成功的因素是 `friend` 品質所致。

下列範例顯示在 /std: c + + 17 模式中 Visual Studio 2017 和更新版本中的 c + + 17 行為：

```
struct Derived;

struct Base {
    friend struct Derived;
private:
    Base() {}
};

struct Derived : Base {
    Derived() {} // add user-defined constructor
                 // to call with {} initialization
};

Derived d1; // OK. No aggregate init involved.

Derived d2 {}; // error C2248: 'Base::Base': cannot access
               // private member declared in class 'Base'
```

## 具有多重繼承之類別的函式

如果類別從多個基底類別衍生，基底類別建構函式是依照其列在衍生類別宣告中的順序進行叫用：

```
#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;
}
```

可預期下列輸出：

```
BaseClass1 ctor
BaseClass2 ctor
BaseClass3 ctor
DerivedClass ctor
```

## 委派函式

委派的函式會在相同的類別中呼叫不同的函式，以執行一些初始化工作。當您有多個必須執行類似工作的多個函式時，這非常有用。您可以在一個函式中撰寫主要邏輯，並從其他函式叫用它。在下列的簡單範例中，Box (int) 會

將其工作委派給 Box (int, int, int) :

```
class Box {  
public:  
    // Default constructor  
    Box() {}  
  
    // Initialize a Box with equal dimensions (i.e. a cube)  
    Box(int i) : Box(i, i, i); // delegating constructor  
    {}  
  
    // Initialize a Box with custom dimensions  
    Box(int width, int length, int height)  
        : m_width(width), m_length(length), m_height(height)  
    {}  
    //... rest of class as before  
};
```

在任何建構函式完成時，建構函式建立的物件會立即完全初始化。如需詳細資訊，請參閱 [委派函數](#)。

## (C + + 11) 繼承函數

衍生的類別可以使用宣告來繼承直接基類的函式，`using` 如下列範例所示：

```

#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 和更新版本：`using /std: c++17` 模式中的語句會從基類將所有的函式納入範圍中，除了對衍生類別中的函式具有相同簽章的函式。一般而言，衍生類別未宣告新的資料成員或建構函式時，最好使用繼承建構函式。

如果類型引數指定基底類別，則類別樣板可以繼承該類型的所有建構函式：

```

template< typename T >
class Derived : T {
    using T::T;    // declare the constructors from T
    // ...
};

```

如果多個基底類別的建構函式具有相同簽章，則衍生類別無法繼承自這些基底類別。

## 函數和複合類別

包含類別成員的類別稱為 **複合類別**。在建立複合類別的類別成員時，會先呼叫建構函式，然後呼叫類別自己的建構函式。當包含的類別缺少預設建構函式時，您必須在複合類別的建構函式中使用初始設定清單。在先

前的 `StorageBox` 範例中，如果將 `m_label` 成員變數的類型變更為新的 `Label` 類別，您必須呼叫基底類別建構函式和初始化 `m_label` 建構函式中的 `StorageBox` 變數：

```
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"});
}
```

## 本節內容

- [複製建構函式和複製指派運算子](#)
- [移動建構函式和移動指派運算子](#)
- [委派建構函式](#)

## 請參閱

[類別和結構](#)

# 複製建構函式和複製指派運算子 (C++)

2020/11/2 • [Edit Online](#)

## NOTE

從 C++ 11 開始，語言中支援兩種指派：複製指派和移動指派。在本文中，除非明確指定，否則「指派」表示複製指派。如需移動指派的詳細資訊，請參閱移動函數和移動指派運算子 (C++)。

指派作業和初始化作業都會導致複製物件。

- **指派**：將一個物件的值指派給另一個物件時，第一個物件會複製到第二個物件。因此，

```
Point a, b;  
...  
a = b;
```

會導致 `b` 的值複製到 `a`。

- **初始化**：當宣告新物件、引數以傳值方式傳遞至函式，或從函式依值傳回值時，就會進行初始化。

您可以定義類別類型物件的「複製」語意。例如，請思考下列程式碼：

```
TextFile a, b;  
a.Open( "FILE1.DAT" );  
b.Open( "FILE2.DAT" );  
b = a;
```

上述程式碼可能表示「將 FILE1.DAT 的內容複製到 FILE2.DAT」，也可能表示「忽略 FILE2.DAT 並且讓 `b` 成為 FILE1.DAT 的第二個控制代碼」。您必須將適當的複製語意附加至每一個類別，如下所示。

- 藉由使用指派運算子 `operator =` 搭配類別類型的參考，做為傳回型別和以傳址方式傳遞的參數 `const`（例如 `ClassName& operator=(const ClassName& x);`）。
- 使用複製建構函式。

如果您未宣告複製建構函式，則編譯器會產生一個成員複製建構函式。如果您未宣告複製指派建構函式，則編譯器會產生一個成員複製指派運算子。宣告複製建構函式不會隱藏編譯器產生的複製指派運算子，反之亦然。如果您實作任一種，建議您一併實作另一種，如此程式碼的意義才會明確。

複製的函式會採用類型類別名稱的引數 `&`，其中類別名稱是已定義其函式的類別名稱。例如：

```
// spec1_copying_class_objects.cpp  
class Window  
{  
public:  
    Window( const Window& ); // Declare copy constructor.  
    // ...  
};  
  
int main()  
{  
}
```

#### NOTE

請盡可能將複製參數的引數 `const` 類別名稱類型設為 `&`。這樣可避免複製建構函式意外變更做為複製來源的物件。它也可讓您從物件進行複製 `const`。

## 編譯器產生的複製建構函式

編譯器產生的複製函數(例如使用者定義的複製函式)具有「類別名稱的參考」類型的單一引數。當所有基類和成員類別都有宣告為接受類型 `const` 類別名稱之單一引數的複製函數時，就會發生例外狀況 `&`。在這種情況下，編譯器產生的複製函式的引數也是 `const`。

當複製程式化的引數類型不是時 `const`，藉由複製物件來初始化會 `const` 產生錯誤。相反的：如果引數是 `const`，您可以藉由複製不是的物件來初始化 `const`。

編譯器產生的指派運算子會遵循與 `const` 有關的相同模式。*class-name &* 除非所有基底和成員類別中的指派運算子接受類型 `const` 類別名稱的引數，否則它們會採用類型類別名稱的單一引數 `&`。在此情況下，類別產生的指派運算子會採用 `const` 引數。

#### NOTE

虛擬基底類別是由複製建構函式進行初始化、由編譯器所產生或使用者所定義時，只會在建構時初始化一次。

這些影響類似複製建構函式的影響。當引數類型不是時 `const`，從物件指派會 `const` 產生錯誤。相反的情況：如果將 `const` 值指派給不是的值 `const`，指派會成功。

如需多載指派運算子的詳細資訊，請參閱[指派](#)。

# 移動建構函式和移動指派運算子 (C++)

2020/11/2 • [Edit Online](#)

本主題說明如何撰寫 C++ 類別的 移動函式和移動指派運算子。移動的函式可讓右值物件所擁有的資源移至左值，而不需要複製。如需有關移動語義的詳細資訊，請參閱右值參考宣告子：`&&`。

這個主題是以下列管理記憶體緩衝區的 C++ 類別 `MemoryBlock` 為基礎。

```
// 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.
};

```

下列程序說明如何撰寫 C++ 類別的移動建構函式和移動指派運算子。

### 建立 C++ 類別的移動建構函式

- 定義空的建構函式方法，該方法會接受以類別類型的右值參考做為其參數，如下列範例所示範：

```

MemoryBlock(MemoryBlock&& other)
: _data(nullptr)
, _length(0)
{
}

```

- 在移動建構函式中，將類別資料成員從來源物件指派給將建構的物件：

```

_data = other._data;
_length = other._length;

```

- 將來源物件的資料成員指派為預設值。這樣可防止解構函式多次釋放資源（例如記憶體）：

```

other._data = nullptr;
other._length = 0;

```

### 建立 C++ 類別的移動指派運算子

- 定義空的指派運算子，該運算子會接受以類別類型的右值參考做為其參數，並傳回對類別類型的參考，如下列範例所示範：

```

MemoryBlock& operator=(MemoryBlock&& other)
{
}

```

- 在移動指派運算子中，如果您嘗試將物件指派給其本身，請新增不執行任何作業的條件陳述式。

```

if (this != &other)
{
}

```

- 在條件陳述式中，從所指派的物件釋放所有資源（例如記憶體）。

下列範例會從所指派的物件釋放 `_data` 成員：

```
// Free the existing resource.  
delete[] _data;
```

依照第一個程序的步驟 2 和步驟 3，將資料成員從來源物件傳送至將建構的物件：

```
// 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. 傳回目前物件的參考，如下列範例所示範：

```
return *this;
```

## 範例：完成移動函式和指派運算子

下列範例示範 `MemoryBlock` 類別的完整移動建構函式和移動指派運算子：

```

// 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;
}

```

## 範例使用 move 語義來改善效能

下列範例示範移動語意如何改善應用程式的效能。此範例會在向量物件中加入兩個元素，然後在兩個現有元素之間插入新的元素。`vector` 類別會使用 move 語義，藉由移動向量的元素而非複製它們來有效率地執行插入作業。

```

// 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));
}

```

這個範例會產生下列輸出：

```

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，此範例會產生下列輸出：

```

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.

```

這個使用移動語意的範例版本比不使用移動語意的版本更有效率，因為前者執行較少的複製、記憶體配置和記憶體解除配置作業。

## 穩固程式設計

為避免資源流失，請一律釋放移動指派運算子中的資源（例如記憶體、檔案控制代碼和通訊端）。

為避免解構資源後無法復原，請適當地處理移動指派運算子中的自我指派。

如果您同時為類別提供移動建構函式和移動指派運算子，可以撰寫移動建構函式來呼叫移動指派運算子，藉此去除冗餘碼。下列範例示範呼叫移動指派運算子的修訂版移動建構函式。

```
// Move constructor.  
MemoryBlock(MemoryBlock&& other) noexcept  
    : _data(nullptr)  
    , _length(0)  
{  
    *this = std::move(other);  
}
```

Std:: move函數會將左值轉換 `other` 成右值。

## 另請參閱

[右值參考宣告子:&&](#)

[std:: move](#)

# 委派建構函式

2020/4/15 • [Edit Online](#)

許多類別有多個執行類似操作的建構函數,例如,驗證參數:

```
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;  
    }  
};
```

您可以通過添加執行所有驗證的函數來減少重複代碼,但如果一個構造函數可以將某些工作委託給 `class_c` 另一個構造函數,則代碼更易於理解和維護。要新增委派建構函數,請使用語法 `constructor ( . . . ) : constructor ( . . . )`:

```
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 };  
}
```

在遍歷前面的範例中,請注意建構函數 `class_c(int, int, int)` 首先呼叫式函數,而建構函數 `class_c(int, int)` 又呼 `class_c(int)` 叫。每個構造函數只執行其他構造函數未執行的工作。

調用物件的第一個構造函數將初始化該物件,以便此時為其所有成員初始化。不能在委託給其他構造函數的構造函數中執行成員初始化,如下所示:

```

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;
};


```

下一個範例顯示了非靜態數據成員初始化器的使用。請注意,如果建構函數還初始化給定的資料成員,則成員初始化器將被覆蓋:

```

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;
}

```

建構函數委派語法不會阻止意外創建構造函數遞歸 — 構造函數1 調用構造函數2,調用構造函數 1 — 並且在存在堆疊溢出之前不會引發任何錯誤。你有責任避免迴圈。

```

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() { }
};

```

# 解構函式 (C++)

2020/11/2 • [Edit Online](#)

「析構函式」是一種成員函式，會在物件超出範圍或由的呼叫明確終結時自動叫用 `delete`。析構函式的名稱與類別相同，前面加上波狀符號 (`~`)。例如，`String` 類別的解構函式宣告為：`~String()`。

如果您未定義析構函式，編譯器會提供預設值，對於許多類別而言，這就已足夠。當類別儲存必須釋放的系統資源控制碼時，或是擁有所指向之記憶體的指標時，您只需要定義自訂的析構函式。

請考慮下面的 `String` 類別宣告：

```
// spec1_destructors.cpp
#include <string>

class String {
public:
    String( char *ch ); // Declare constructor
    ~String();           // and destructor.
private:
    char     *_text;
    size_t   sizeOfText;
};

// Define the constructor.
String::String( char *ch ) {
    sizeOfText = strlen( ch ) + 1;

    // Dynamically allocate the correct amount of memory.
    _text = new char[ sizeOfText ];

    // If the allocation succeeds, copy the initialization string.
    if( _text )
        strcpy_s( _text, sizeOfText, ch );
}

// Define the destructor.
String::~String() {
    // Deallocate the memory that was previously reserved
    // for this string.
    delete[] _text;
}

int main() {
    String str("The piper in the glen...");
}
```

在上述範例中，函式會 `String::~String` 使用 `delete` 運算子來解除配置動態配置給文字儲存區的空間。

## 宣告析構函數

解構函式是名稱與類別相同的函式，但其名稱前面會加上波狀符號 (`~`)。

有數種規則用於管理解構函式的宣告。解構函式：

- 不接受引數。
- 不會傳回值(或 `void`)。

- 不可以宣告為 `const`、`volatile` 或 `static`。不過，它們可以在被宣告為、或的物件遭到破壞時叫用 `const` `volatile` `static`。
- 可以宣告為 `virtual`。使用虛擬解構函式可以終結物件而不需要知道它們的類型，其會使用虛擬函式機制為物件叫用正確的解構函式。請注意，解構函式也可以宣告為抽象類別的純虛擬函式。

## 使用解構函式

在下列任一事件發生時，會呼叫解構函式：

- 區塊範圍內的區域（自動）物件會超出範圍。
- 使用運算子配置的物件 `new` 會使用明確地解除配置 `delete`。
- 暫存物件的存留期結束。
- 程式結束，而全域或靜態物件存在。
- 使用解構函式的函式完整名稱明確地呼叫解構函式。

解構函式可以自由呼叫類別成員函式和存取類別成員資料。

使用析構函數有兩項限制：

- 您無法接受其位址。
- 衍生類別不會繼承其基類的析構函式。

## 解構順序

當物件超出範圍或被刪除時，事件完整解構的順序如下所示：

1. 呼叫類別的解構函式，並且執行解構函式的主體。
2. 按照非靜態成員物件出現在類別解構函式中的順序反向呼叫其解構函式。用於結構化這些成員的選擇性成員初始化清單不會影響結構或損毀的順序。
3. 非虛擬基類的析構函數是以宣告的反向順序呼叫。
4. 按照宣告的相反順序呼叫虛擬基底類別的解構函式。

```

// order_of_destruction.cpp
#include <cstdio>

struct A1      { virtual ~A1() { printf("A1 dtor\n"); } };
struct A2 : A1 { virtual ~A2() { printf("A2 dtor\n"); } };
struct A3 : A2 { virtual ~A3() { printf("A3 dtor\n"); } };

struct B1      { ~B1() { printf("B1 dtor\n"); } };
struct B2 : B1 { ~B2() { printf("B2 dtor\n"); } };
struct B3 : B2 { ~B3() { printf("B3 dtor\n"); } };

int main() {
    A1 * a = new A3;
    delete a;
    printf("\n");

    B1 * b = new B3;
    delete b;
    printf("\n");

    B3 * b2 = new B3;
    delete b2;
}

```

Output: A3 dtor

A2 dtor

A1 dtor

B1 dtor

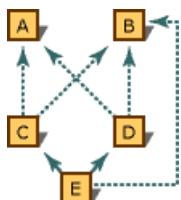
B3 dtor

B2 dtor

B1 dtor

## 虛擬基底類別

虛擬基底類別的解構函式會依它們出現在導向非循環圖中的反向順序呼叫 (深度優先、由左至右、後序走訪)。下圖將說明繼承圖表。



顯示虛擬基底類別的繼承圖形

以下列出圖中所顯示類別的類別開頭。

```

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

```

為了判斷 **E** 類型物件之虛擬基底類別的解構順序，編譯器會套用下列演算法來建置清單：

1. 周遊左側圖表，從圖中的最深點開始 (在這個案例中為 **E**)。
2. 向左周遊，直到瀏覽過所有節點。記下目前節點的名稱。
3. 再次瀏覽上一個節點 (右下方)，確認記住的節點是否為虛擬基底類別。
4. 如果記住的節點是虛擬基底類別，請掃描清單，查看該節點是否已輸入。如果不是虛擬基底類別，則會忽略

它。

5. 如果記住的節點不在清單中，請將它加入清單的底部。
6. 向上並沿著下一個向右的路徑周遊圖表。
7. 移至步驟 2。
8. 到達最後一個向上路徑時，請記下目前節點的名稱。
9. 移至步驟 3。
10. 繼續這個程序，到底部節點再次成為目前節點為止。

因此，**E** 類別的解構順序如下：

1. 非虛擬基類 **E**。
2. 非虛擬基類 **D**。
3. 非虛擬基類 **C**。
4. 虛擬基底類別 **B**。
5. 虛擬基底類別 **A**。

這個程序會產生已排序的唯一項目清單。類別名稱不會重複出現。清單建構之後，會依反向順序查看該清單，而清單中每個類別的解構函式會依最後一個到第一個的順序呼叫。

如果某一個類別中的建構函式或解構函式要求其他元件必須先建立或保存更長時間 (例如，如果 **A** 的解構函式 (如上圖) 要求其程式碼執行時，**B** 必須持續存在 (反之亦然))，建構或解構的順序就會非常重要。

繼承圖表中類別之間的這種相依性原本就存在危險性，因為之後衍生的類別可以修改最左邊的路徑，藉此變更建構和解構的順序。

### 非虛擬基類

非虛擬基類的析構函數是以宣告基類名稱的反向順序來呼叫。請考慮下列類別宣告：

```
class MultInherit : public Base1, public Base2
...
```

在上述範例中，**Base2** 的解構函式是在 **Base1** 的解構函式之前呼叫。

## 明確解構函式呼叫

明確呼叫解構函式不是必要的步驟。不過，這對放置於絕對位址的物件執行清除作業可能會很有用。這些物件通常會使用 **new** 接受位置引數的使用者定義運算子進行配置。**delete** 操作員無法解除配置此記憶體，因為它不是從免費儲存區配置的(如需詳細資訊，請參閱[new 和 delete 運算子](#))。不過，呼叫解構函式時，可以執行適當的清除作業。若要明確呼叫物件的解構函式 (**s** 類別的 **String**)，請使用下列其中一種陳述式：

```
s.String::~String();      // non-virtual call
ps->String::~String();  // non-virtual call

s.~String();            // Virtual call
ps->~String();         // Virtual call
```

您可以使用明確呼叫解構函式的標註法 (如先前所示)，不論該類型是否定義了解構函式。這可讓您進行這類明確呼叫，而不需要知道是否已為該類型定義解構函式。明確呼叫未定義的解構函式不會有任何作用。

## 穩固程式設計

如果類別取得資源，就需要有一個析構函式，而若要安全地管理資源，它可能必須執行複製的處理常式和複製作業。

如果這些特殊函式不是由使用者定義，則由編譯器隱含定義。隱含產生的函式和指派運算子會執行淺層的成員複製，如果物件正在管理資源，這幾乎肯定會錯誤。

在下一個範例中，隱含產生的複製程式將會建立指標 `str1.text` 並 `str2.text` 參考相同的記憶體，而當我們從傳回時，`copy_strings()` 該記憶體會被刪除兩次，這是未定義的行為：

```
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
```

明確定義的「析構函式」、「複製」「函數」或「複製指派運算子」，可防止移動函數和移動指派運算子的隱含定義。在此情況下，如果複製的成本很高，則無法提供移動作業通常是遺失的優化機會。

## 另請參閱

[複製構造函式和複製指派運算子](#)

[移動構造函式和移動指派運算子](#)

# 成員函式概觀

2020/11/2 • [Edit Online](#)

成員函式不是靜態就是非靜態。靜態成員函式的行為與其他成員函式不同，因為靜態成員函數沒有隱含 `this` 引數。非靜態成員函式具有 `this` 指標。無論是靜態或非靜態的成員函式，都可以在類別宣告之內或之外定義。

如果成員函式是在類別宣告內定義，則會將它視為內嵌函式，而且不需要使用其類別名稱限定函式名稱。雖然在類別宣告內定義的函式已視為內嵌函式，但您可以使用 `inline` 關鍵字來記錄程式碼。

以下是在類別宣告內宣告函式的範例：

```
// 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()
{
}
```

如果成員函式的定義是在類別宣告之外，則只有在明確宣告為時，才會將它視為內嵌函數 `inline`。此外，定義中的函式名稱必須使用範圍解析運算子 (`::`) 以其類別名稱加以限定。

下列範例與上述 `Account` 類別宣告相同，唯一的差異在於 `Deposit` 函式是在類別宣告之外定義：

```
// overview_of_member_functions2.cpp
class Account
{
public:
    // Declare the member function Deposit but do not define it.
    double Deposit( double HowMuch );
private:
    double balance;
};

inline double Account::Deposit( double HowMuch )
{
    balance += HowMuch;
    return balance;
}

int main()
{
}
```

**NOTE**

雖然成員函式可以在類別宣告內或分開定義，但是在類別定義之後，成員函式不可加入至該類別。

包含成員函式的類別可以具有多個宣告，不過成員函式本身在程式中只能有一個定義。若有多個定義，則會在連結時產生錯誤訊息。如果類別包含內嵌函式定義，則函式定義必須一致，以遵守這項「一個定義」的規則。

# virtual 規範

2020/3/25 • [Edit Online](#)

**Virtual**關鍵字只能套用至非靜態類別成員函式。它表示將函式呼叫的繫結延後到執行階段。如需詳細資訊，請參閱[虛擬函式](#)。

# override 規範

2020/3/25 • [Edit Online](#)

您可以使用**override**關鍵字來指定覆寫基類中虛擬函式的成員函式。

## 語法

```
function-declaration override;
```

## 備註

覆寫會區分內容，而且只有在成員函式宣告之後使用時才具有特殊意義；否則，它不是保留的關鍵字。

## 範例

使用覆寫有助於防止程式碼中發生意外的繼承行為。下列範例示範不使用覆寫的 where，衍生類別的成員函式行為可能不是預期的。編譯器不會對這個程式碼發出任何錯誤。

```
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
};
```

當您使用**override**時，編譯器會產生錯誤，而不是以無訊息方式建立新的成員函式。

```
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()
};


```

若要指定無法覆寫函數，而且無法繼承類別，請使用[final](#)關鍵字。

## 另請參閱

[final 規範](#)

[關鍵字](#)

# final 規範

2020/3/25 • [Edit Online](#)

您可以使用final關鍵字來指定無法在衍生類別中覆寫的虛擬函式。您也可以用它來指定無法被繼承的類別。

## 語法

```
function-declaration final;
class class-name final base-classes
```

## 備註

final是內容相關的，只有在函式宣告或類別名稱之後使用時才具有特殊意義；否則，它不是保留的關鍵字。

當在類別宣告中使用final時，`base-classes` 是宣告的選擇性部分。

## 範例

下列範例會使用final關鍵字來指定無法覆寫虛擬函式。

```
class BaseClass
{
    virtual void func() final;
};

class DerivedClass: public BaseClass
{
    virtual void func(); // compiler error: attempting to
                        // override a final function
};
```

如需如何指定可覆寫成員函式的詳細資訊，請參閱[覆寫規範](#)。

下一個範例會使用final關鍵字來指定無法繼承類別。

```
class BaseClass final
{
};

class DerivedClass: public BaseClass // compiler error: BaseClass is
                                    // marked as non-inheritable
{
```

## 另請參閱

[關鍵字](#)

[override 規範](#)

# 繼承 (C++)

2020/11/2 • [Edit Online](#)

本節將說明如何使用衍生類別產生可擴充程式。

## 概觀

使用稱為「繼承」的機制，可以從現有類別衍生新的類別（請參閱從[單一繼承](#)開始的資訊）。供衍生使用的類別稱為特定衍生類別的「基底類別」。衍生類別會使用下列語法宣告：

```
class Derived : [virtual] [access-specifier] Base
{
    // member list
};

class Derived : [virtual] [access-specifier] Base1,
    [virtual] [access-specifier] Base2, . . .
{
    // member list
};
```

在類別的標記（名稱）後方會有一個冒號後面接著基底規格的清單。基底類別必須先宣告，因此如此命名。基底規格可能包含存取規範，也就是其中一個關鍵字 `public`、`protected` 或 `private`。這些存取指定名稱會出現在基底類別名稱的前方，並且只會套用至該基底類別。這些指定名稱可控制衍生類別使用基底類別成員的權限。如需存取基類成員的資訊，請參閱[成員存取控制](#)。如果省略存取規範，則會考慮對該基底的存取權 `private`。基底規格可能包含關鍵字 `virtual` 來表示虛擬繼承。這個關鍵字會顯示在存取指定名稱的前方或後方（如果有的話）。如果使用虛擬繼承，則會將基底類別稱為是虛擬基底類別。

您可以指定多個基底類別（以逗號分隔）。如果指定了單一基類，繼承模型就是[單一繼承](#)。如果指定了三個以上的基類，則繼承模型稱為[多重繼承](#)。

本文包含下列主題：

- [單一繼承](#)
- [多個基底類別](#)
- [虛擬函式](#)
- [明確覆寫](#)
- [抽象類別](#)
- [範圍規則摘要](#)

`_Super`和`_interface`關鍵字記載于本節。

## 另請參閱

[C++ 語言參考](#)

# 虛擬函式

2020/11/2 • [Edit Online](#)

虛擬函式是您必須在衍生類別中重新定義的成員函式。當您使用基底類別的參考或指標參考衍生類別物件時，可以呼叫該物件的虛擬函式，並執行函式的衍生類別版本。

不論呼叫函式時使用的運算式為何，虛擬函式都可確保針對物件呼叫正確的函式。

假設基類包含宣告為 `virtual` 的函式，且衍生類別定義相同的函式。針對衍生類別的物件會叫用來自衍生類別的函式（即使是使用基底類別的指標或參考呼叫）。下列範例將示範基底類別，它會提供 `PrintBalance` 函式和兩個衍生類別的實作。

```
// 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();
}
```

在上述程式碼中，除了針對 `PrintBalance` 所指向的物件之外，`pAccount` 的呼叫都相同。由於 `PrintBalance` 是虛擬的，因此會呼叫為每個物件定義的函式版本。衍生類別 `PrintBalance` 和 `CheckingAccount` 中的 `SavingsAccount` 函式會「覆寫」基底類別 `Account` 中的函式。

如果宣告的類別未提供 `PrintBalance` 函式的覆寫實作，則會使用基底類別 `Account` 的預設實作。

衍生類別中的函式只有在類型相同時，才會覆寫基底類別中的虛擬函式。衍生類別內的函式不能只使用傳回類型與基底類別內的虛擬函式作區別，引數清單也必須加以區別。

使用指標或參考呼叫函式時，適用下列規則：

- 對虛擬函式的呼叫會根據為其進行呼叫之物件的基礎類型進行解析。
- 對非虛擬函式的呼叫是根據指標或參考的類型進行解析。

下列範例將示範透過指標呼叫時，虛擬和非虛擬函式的行為：

```
// deriv_VirtualFunctions2.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

class Base {
public:
    virtual void NameOf(); // Virtual function.
    void InvokingClass(); // Nonvirtual function.
};

// Implement the two functions.
void Base::NameOf() {
    cout << "Base::NameOf\n";
}

void Base::InvokingClass() {
    cout << "Invoked by Base\n";
}

class Derived : public Base {
public:
    void NameOf(); // Virtual function.
    void InvokingClass(); // Nonvirtual function.
};

// Implement the two functions.
void Derived::NameOf() {
    cout << "Derived::NameOf\n";
}

void Derived::InvokingClass() {
    cout << "Invoked by Derived\n";
}

int main() {
    // Declare an object of type Derived.
    Derived aDerived;

    // Declare two pointers, one of type Derived * and the other
    // of type Base *, and initialize them to point to aDerived.
    Derived *pDerived = &aDerived;
    Base     *pBase   = &aDerived;

    // Call the functions.
    pBase->NameOf();           // Call virtual function.
    pBase->InvokingClass();    // Call nonvirtual function.
    pDerived->NameOf();         // Call virtual function.
    pDerived->InvokingClass(); // Call nonvirtual function.
}
```

```
Derived::NameOf  
Invoked by Base  
Derived::NameOf  
Invoked by Derived
```

請注意，無論 `NameOf` 函式是經由 `Base` 的指標或 `Derived` 的指標叫用，都會呼叫 `Derived` 的函式。它會呼叫 `Derived` 的函式是因為 `NameOf` 是虛擬函式，而且 `pBase` 和 `pDerived` 都指向 `Derived` 類型的物件。

因為只會針對類別類型的物件呼叫虛擬函式，所以您無法將全域或靜態函式宣告為 `virtual`。

在 `virtual` 衍生類別中宣告覆寫函式時，可以使用關鍵字，但這是不必要的；虛擬函式的覆寫一律是虛擬的。

必須定義基類中的虛擬函式，除非它們是使用純規範來宣告。（如需純虛擬函式的詳細資訊，請參閱[抽象類別](#)）。

使用範圍解析運算子 (`::`) 就可透過明確限定函式名稱的方式隱藏虛擬函式呼叫機制。請參考先前包含 `Account` 類別的範例。若要呼叫基底類別中的 `PrintBalance`，請使用下列範例程式碼：

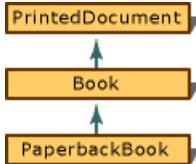
```
CheckingAccount *pChecking = new CheckingAccount( 100.00 );  
  
pChecking->Account::PrintBalance(); // Explicit qualification.  
  
Account *pAccount = pChecking; // Call Account::PrintBalance  
  
pAccount->Account::PrintBalance(); // Explicit qualification.
```

上述範例中對 `PrintBalance` 的兩個呼叫都會隱藏虛擬函式呼叫機制。

# 單一繼承

2020/11/2 • [Edit Online](#)

在「單一繼承」(一種常見的繼承形式) 中，任何類別都只能有一個基底類別。請考慮下圖中說明的關係。



簡單的單一繼承圖形

請注意圖中從一般到特定的進展。在大部分類別階層架構設計中，另一種常見的屬性是衍生類別，其與基底類別具有一種「種類」(kind of) 的關聯性。在圖中，`Book` 是一種 `PrintedDocument`，而 `PaperbackBook` 是一種 `book`。

請注意圖中的其他項目：`Book` 同時為衍生類別 (來自 `PrintedDocument`) 和基底類別 (`PaperbackBook` 衍生自 `Book`)。這種類別階層架構的基本架構宣告如下列範例所示：

```
// deriv_SingleInheritance.cpp
// compile with: /LD
class PrintedDocument {};

// Book is derived from PrintedDocument.
class Book : public PrintedDocument {};

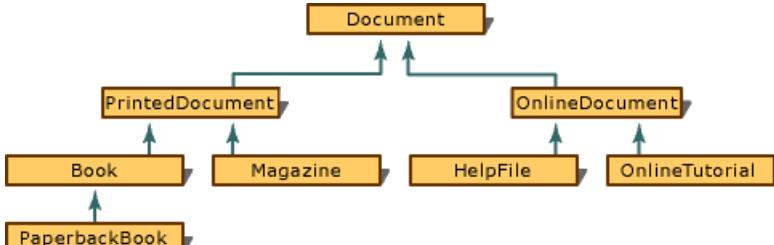
// PaperbackBook is derived from Book.
class PaperbackBook : public Book {};
```

`PrintedDocument` 被視為 `Book` 的「直接基底」類別，它是 `PaperbackBook` 的「間接基底」類別。其中的差異是直接基底類別會出現在類別宣告的基底清單中，而間接基底則沒有。

每個類別的衍生基底類別都是在宣告衍生類別之前宣告。它並不足以提供基底類別的向前參考宣告，且必須是完整的宣告。

在上述範例中，`public` 會使用存取規範。公開、受保護和私用繼承的意義在成員存取控制中有所描述。

類別可做為許多特定類別的基底類別，如下圖所示。



導向非循環圖的範例

上圖中 (稱為「導向非循環圖」或 "DAG") 的某些類別是多個衍生類別的基底類別。不過，反向並不成立：其中只有一個指定衍生類別的直接基底類別。此圖中描述了一個「單一繼承」結構。

## NOTE

導向非循環圖不是唯一的單一繼承。它們也可用來描述多重繼承圖形。

在繼承中，衍生類別包含基底類別的成員以及您加入的新成員。因此，除非在衍生類別中重新定義這些成員，否則衍生類別可能會參考基底類別的成員。當這些成員在衍生類別中重新定義時，可以使用範圍解析運算子 ( :: ) 來參考直接或間接基底類別的成員。請思考此範例：

```
// 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;
};
```

請注意，`Book` 的建構函式 (即 `Book::Book`) 可以存取資料成員 `Name`。在程式中可以建立及使用 `Book` 類型的物件，如下所示：

```
// 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();
```

如上述範例中所示，類別成員和繼承的資料和函式的使用方式是相同的。如果類別 `Book` 的實作呼叫 `PrintNameOf` 函式的重新實作，則只能使用範圍解析 (`Document`) 運算子呼叫屬於 :: 類別中的函式：

```

// 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();
}

```

如果有一個可存取且明確的基底類別，衍生類別的指標和參考便可以隱含轉換為其基底類別的指標和參考。下列程式碼使用指標示範這個概念 (相同原則適用於參考)：

```

// 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;
}

```

上述範例中建立了不同的類型。不過，由於這些類型都是衍生自 `Document` 類別，所以會產生 `Document *` 的隱含轉換。因此，`DocLib` 是一個「異質性清單」(在此清單中，並非所有物件都屬於相同類型)，其中包含不同類型的物件。

由於 `Document` 類別有一個 `PrintNameOf` 函式，它可以列印圖書館中每本書的名稱，不過，它可能會忽略文件的某些特定類型資訊 (`Book` 的頁面計數、`HelpFile` 的位元組數，等等)。

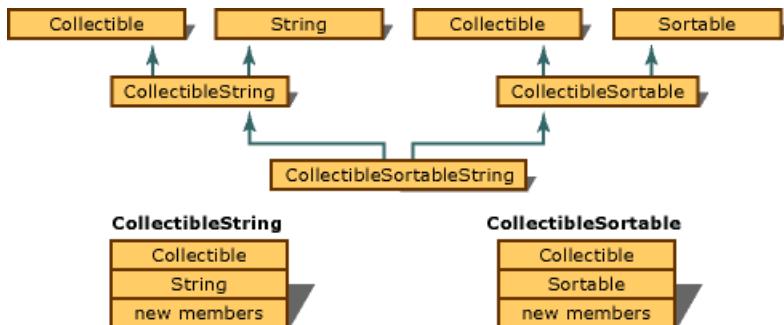
#### NOTE

強制基底類別實作如 `PrintNameOf` 等函式通常不是最好的設計。[虛擬](#)函式提供其他設計替代方案。

# 基底類別

2020/3/25 • [Edit Online](#)

繼承的程序會建立新的衍生類別，該類別是由基底類別的成員再加上由衍生類別加入的任何新成員所組成。在多重繼承中可以建構繼承圖形，其中相同的基底類別是屬於多個衍生類別的一部分。下圖顯示這類圖形。



## 單一基類的多個實例

圖中以圖示表示 `CollectibleString` 和 `CollectibleSortable` 的各個元件。不過，位於 `Collectible` 中的基底類別 `CollectibleSortableString` 是透過 `CollectibleString` 路徑和 `CollectibleSortable` 路徑存取。若要消除這種冗餘狀況，可以在繼承時將這些類別宣告為虛擬基底類別。

# 多個基底類別

2020/11/2 • [Edit Online](#)

類別可以衍生自一個以上的基類。在多重繼承模型中(其中類別衍生自一個以上的基類)，會使用基底清單文法專案來指定基類。例如，可以指定衍生自 `CollectionOfBook` 和 `Collection` 的 `Book` 類別宣告：

```
// deriv_MultipleBaseClasses.cpp
// compile with: /LD
class Collection {
};
class Book {};
class CollectionOfBook : public Book, public Collection {
    // New members
};
```

除非是在叫用建構函式和解構函式的特定情況下，否則指定基底類別的順序並不重要。在這些情況下，指定基底類別的順序會影響下列各項：

- 建構函式進行初始化的順序。如果程式碼要求 `Book` 的 `CollectionOfBook` 部分必須在 `Collection` 部分之前初始化，則指定的順序就很重要。初始化會依照在基底清單中指定類別的順序進行。
- 叫用解構函式進行清除的順序。同樣地，如果類別的特定「部分」必須在其他部分終結時存在，則順序就很重要。系統會以基底清單中所指定之類別的反向順序來呼叫析構函數。

## NOTE

基底類別的指定順序可能會影響類別的記憶體配置。請不要依據記憶體中基底成員的順序做出任何程式設計上的決策。

指定基底清單時，您無法多次指定相同的類別名稱。不過，類別可以多次做為衍生類別的間接基底。

## 虛擬基底類別

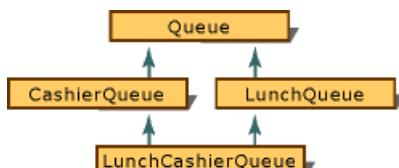
由於類別可以多次做為衍生類別的間接基底類別，因此 C++ 對此類的基底類別工作提供一種最佳化的方式。虛擬基底類別可節省空間，並在使用多重繼承的類別階層架構時避免出現模稜兩可的問題。

每個非虛擬物件都包含基底類別中定義的資料成員。重複項目不僅佔用空間，您還必須在存取時指定您需要的基底類別成員複本。

將基底類別指定為虛擬基底類別時，可多次將其當做間接基底，而不需要使用其資料成員的複本。其資料成員的單一複本會由所有基底類別共用 (這些類別會將其當做虛擬基底使用)。

當宣告虛擬基底類時，`virtual` 關鍵字會出現在衍生類別的基底清單中。

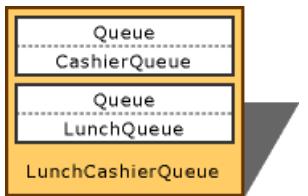
考慮下圖中的類別階層架構，其中示範模擬的午餐供應線。



模擬的午餐-折線圖

在圖中，`Queue` 為 `CashierQueue` 和 `LunchQueue` 的基底類別。不過，在將這兩個類別合併為 `LunchCashierQueue`

時會發生下列問題：新的類別會內含兩個來自 `Queue` 類型的子物件，其中一個來自 `CashierQueue`，另一個則是來自 `LunchQueue`。下圖顯示概念性的記憶體配置（實際的記憶體配置可能會進行最佳化）。

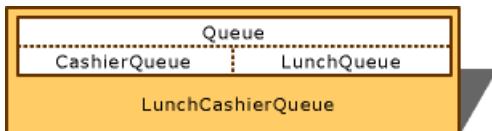


模擬的午餐流程物件

請注意，`Queue` 物件中有兩個 `LunchCashierQueue` 子物件。下列程式碼會將 `Queue` 壓告為虛擬基底類別：

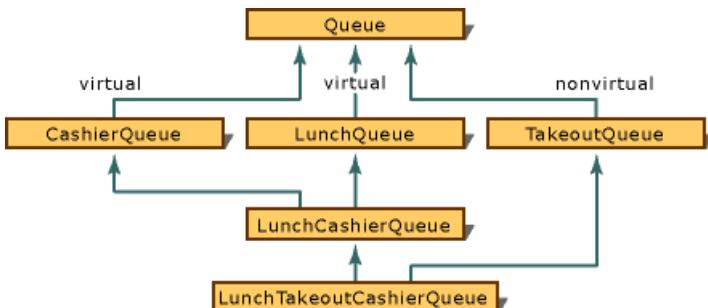
```
// deriv_VirtualBaseClasses.cpp  
// compile with: /LD  
class Queue {};  
class CashierQueue : virtual public Queue {};  
class LunchQueue : virtual public Queue {};  
class LunchCashierQueue : public LunchQueue, public CashierQueue {};
```

`virtual` 關鍵字可確保只包含一個子物件的複本 `Queue`（請參閱下圖）。



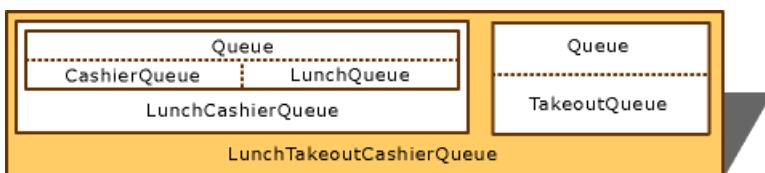
模擬的午餐線物件與虛擬基類

一個類別可以擁有一個虛擬元件和一個特定類型的非虛擬元件。此情況會發生在如下圖中示範的情況下。



相同類別的虛擬和非虛擬元件

在圖中，`CashierQueue` 和 `LunchQueue` 使用 `Queue` 做為虛擬基底類別。不過，`TakeoutQueue` 指定 `Queue` 做為基底類別，而不是虛擬基底類別。因此，`LunchTakeoutCashierQueue` 內含兩個類型為 `Queue` 的子物件：一個是來自包含 `LunchCashierQueue` 的繼承路徑，另一個是來自包含 `TakeoutQueue` 的路徑。下圖中說明此情形。



具有虛擬和非虛擬繼承的物件版面配置

#### NOTE

與使用非虛擬繼承相比較，使用虛擬繼承在大小方面提供相當大的優勢。不過，它可能會增加額外的處理負擔。

如果衍生類別會覆寫從虛擬基底類別繼承的虛擬函式，且衍生基底類別的建構函式或解構函式使用虛擬基底類別的指標呼叫函式，編譯器可能會在內含虛擬基底的類別中採用額外的隱藏 [vtordisp] 欄位。`/vd0` 編譯器選項會抑制隱藏的 vtordisp 函數/析構函式位移成員的加入。`/vd1` 編譯器選項（預設值）會在必要時啟用它們。請只有在您

確定所有類別建構函式和解構函式都會實際呼叫虛擬函式時，才關閉 vtordisps。

/vd 編譯器選項會影響整個編譯模組。使用 `vtordisp` pragma 依類別來隱藏和重新啟用 `vtordisp` 欄位：

```
#pragma vtordisp( off )
class GetReal : virtual public { ... };
\#pragma vtordisp( on )
```

## 名稱語意模糊

多重繼承實現了依循多個路徑繼承名稱的可能性。來自這些路徑的類別成員名稱不一定是唯一的。這些名稱衝突稱為「模稜兩可」。

參考類別成員的所有運算式都必須進行明確參考。下列範例將示範如何發展出模稜兩可的情況：

```
// deriv_NameAmbiguities.cpp
// compile with: /LD
// Declare two base classes, A and B.
class A {
public:
    unsigned a;
    unsigned b();
};

class B {
public:
    unsigned a(); // Note that class A also has a member "a"
    int b();      // and a member "b".
    char c;
};

// Define class C as derived from A and B.
class C : public A, public B {};
```

以上述類別宣告為例，下面這類程式碼就是模稜兩可，因為不清楚 `b` 是參考 `B` 還是 `A` 中的 `B`：

```
C *pc = new C;
pc->b();
```

請參考上述範例。由於名稱 `a` 同時是 `A` 類別和 `B` 類別的成員，因此編譯器無法分辨哪一個 `a` 指定要呼叫的函式。如果成員可以參考多個函式、物件、類型或列舉程式，則對該成員的存取就是模稜兩可的情況。

編譯器會依照下列順序執行測試來偵測模稜兩可的情況：

1. 如果對名稱的存取是模稜兩可的情況（如上所述），則會產生錯誤訊息。
2. 如果多載函式並非模稜兩可，就會進行解析
3. 如果對名稱的存取違反成員存取的權限，則會產生錯誤訊息（如需詳細資訊，請參閱[成員存取控制](#)）。

當運算式透過繼承產生模稜兩可的情況時，您可以使用類別名稱限定所指的名稱，藉此手動解析該名稱。若要讓上述範例正確編譯而不發生模稜兩可的情況，請使用如下所示的程式碼：

```
C *pc = new C;
pc->B::a();
```

#### NOTE

當 `C` 告示時，它可能會在 `B` 於 `C` 的範圍中參考時造成錯誤。不過，只有 `B` 範圍中實際參考未限定的 `C` 時，才會發出錯誤。

### 支配

透過繼承圖形可能會到達多個名稱 (函式、物件或列舉程式)。這種情況視為與非虛擬基底類別模稜兩可。除非其中一個名稱為其他名稱的「主要」名稱，否則它們也是視為與虛擬基底類別模稜兩可。

如果兩個類別中皆定義了該名稱，且某個類別是衍生自另一個類別，則其名稱優先於另一個名稱。主要名稱是衍生類別中的名稱，這個名稱會在出現模稜兩可時使用，如下列範例所示：

```
// deriv_Dominance.cpp
// compile with: /LD
class A {
public:
    int a;
};

class B : public virtual A {
public:
    int a();
};

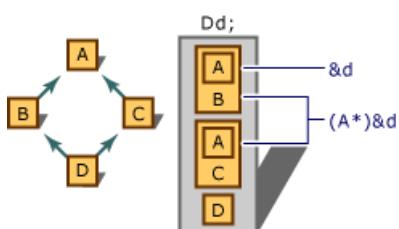
class C : public virtual A {};

class D : public B, public C {
public:
    D() { a(); } // Not ambiguous. B::a() dominates A::a.
};
```

### 模稜兩可的轉換

從指標或參考明確或隱含轉換為類別類型，可能會造成模稜兩可的情況。下圖「指標不明確地轉換為基底類別」顯示下列項目：

- 壓告類型為 `D` 的物件。
- 將通訊運算子 (`&`) 套用至該物件的效果。請注意，傳址運算子一律會提供該物件的基底位址。
- 將使用傳址運算子取得的指標明確轉換為基底類別類型 `A` 的作用。請注意，若強制將物件位址設為 `A*` 類型，編譯器永遠無法得到足以判斷應選取哪一個 `A` 類型之子物件的資訊；在這種情況下，會同時存在兩個子物件。



指標不明確地轉換為基底類別

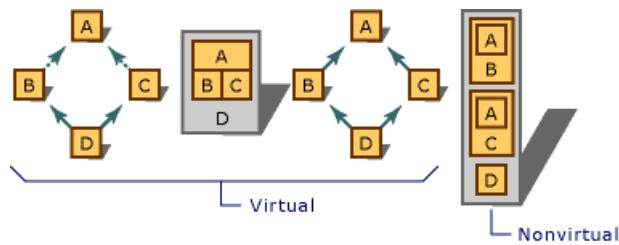
類型 `A*` (`A` 的指標) 的轉換是模稜兩可的，因為沒有辦法分辨哪一個 `A` 類型的子物件是正確的。請注意，您可以明確指定要使用的子物件，以避免模稜兩可的情況，如下所示：

```
(A*)(B*)&d      // Use B subobject.
(A*)(C*)&d      // Use C subobject.
```

## 語意模糊和虛擬基底類別

如果使用虛擬基底類別，則可以透過多重繼承路徑存取函式、物件、類型和列舉值。因為只有一個基底類別的執行個體，在存取這些名稱時不會出現模稜兩可的情形。

下圖顯示物件使用虛擬和非虛擬繼承的組成方式。



### 虛擬與非虛擬衍生

在圖中，透過非虛擬基底類別存取類別 **A** 的所有成員會產生模稜兩可的情況，編譯器不會提供是否要使用與 **B** 關聯的子物件，或使用與 **C** 關聯之子物件的資訊。不過，當 **A** 指定為虛擬基底類別時，沒有存取子物件的問題。

## 另請參閱

[繼承](#)

# 明確覆寫 (C++)

2020/3/25 • • [Edit Online](#)

## Microsoft 專屬

如果在兩個或多個[介面](#)中宣告相同的虛擬函式，而且如果類別衍生自這些介面，您可以明確覆寫每個虛擬函式。

如需使用C++/cli 在 managed 程式碼中明確覆寫的詳細資訊，請參閱[明確覆寫](#)。

## END Microsoft 特定的

## 範例

下列程式碼範例示範如何使用明確覆寫：

```
// deriv_ExplicitOverrides.cpp
// compile with: /GR
extern "C" int printf_s(const char *, ...);

__interface IMyInt1 {
    void mf1();
    void mf1(int);
    void mf2();
    void mf2(int);
};

__interface IMyInt2 {
    void mf1();
    void mf1(int);
    void mf2();
    void mf2(int);
};

class CMyClass : public IMyInt1, public IMyInt2 {
public:
    void IMyInt1::mf1() {
        printf_s("In CMyClass::IMyInt1::mf1()\n");
    }

    void IMyInt1::mf1(int) {
        printf_s("In CMyClass::IMyInt1::mf1(int)\n");
    }

    void IMyInt1::mf2();
    void IMyInt1::mf2(int);

    void IMyInt2::mf1() {
        printf_s("In CMyClass::IMyInt2::mf1()\n");
    }

    void IMyInt2::mf1(int) {
        printf_s("In CMyClass::IMyInt2::mf1(int)\n");
    }

    void IMyInt2::mf2();
    void IMyInt2::mf2(int);
};

void CMyClass::IMyInt1::mf2() {
    printf_s("In CMyClass::IMyInt1::mf2()\n");
}
```

```

void CMyClass::IMyInt1::mf2(int) {
    printf_s("In CMyClass::IMyInt1::mf2(int)\n");
}

void CMyClass::IMyInt2::mf2() {
    printf_s("In CMyClass::IMyInt2::mf2()\n");
}

void CMyClass::IMyInt2::mf2(int) {
    printf_s("In CMyClass::IMyInt2::mf2(int)\n");
}

int main() {
    IMyInt1 *pIMyInt1 = new CMyClass();
    IMyInt2 *pIMyInt2 = dynamic_cast<IMyInt2 *>(pIMyInt1);

    pIMyInt1->mf1();
    pIMyInt1->mf1(1);
    pIMyInt1->mf2();
    pIMyInt1->mf2(2);
    pIMyInt2->mf1();
    pIMyInt2->mf1(3);
    pIMyInt2->mf2();
    pIMyInt2->mf2(4);

    // Cast to a CMyClass pointer so that the destructor gets called
    CMyClass *p = dynamic_cast<CMyClass *>(pIMyInt1);
    delete p;
}

```

```

In CMyClass::IMyInt1::mf1()
In CMyClass::IMyInt1::mf1(int)
In CMyClass::IMyInt1::mf2()
In CMyClass::IMyInt1::mf2(int)
In CMyClass::IMyInt2::mf1()
In CMyClass::IMyInt2::mf1(int)
In CMyClass::IMyInt2::mf2()
In CMyClass::IMyInt2::mf2(int)

```

## 另請參閱

[繼承](#)

# 抽象類別 (C++)

2020/3/25 • [Edit Online](#)

抽象類別用於表示可衍生更明確類別的一般概念。您無法建立抽象類別類型的物件，但是可以使用抽象類別類型的指標和參考。

至少包含一個純虛擬函式的類別會被視為抽象類別。衍生自抽象類別的類別必須實作純虛擬函式，否則這些類別也是抽象類別。

請考慮在虛擬函式中呈現的範例。`Account` 類別的目的是要提供一般功能，但 `Account` 類型的物件則過於籠統，實用性不高。因此，`Account` 非常適合作為抽象類別：

```
// 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;
};
```

此宣告和上一個宣告之間唯一的差別，在於 `PrintBalance` 是使用純指定名稱 (`= 0`) 壓告的。

## 抽象類別的限制

抽象類別不能用於：

- 變數或成員資料
- 引數類型
- 函式傳回型別
- 明確轉換的類型

另一項限制是，如果抽象類別的建構函式直接或間接呼叫純虛擬函式，則結果會是未定義。不過，抽象類別的建構函式和解構函式可以呼叫其他成員函式。

您可以為抽象類別定義純虛擬函式，但是只能使用下列語法直接呼叫：

`抽象類別-name::function-name()`

這在設計基底類別包含純虛擬解構函式的類別階層架構時很有幫助，因為基底類別解構函式會一律在終結物件的處理序中呼叫。請考慮下列範例：

```
// Declare an abstract base class with a pure virtual destructor.  
// deriv_RestrictionsOnUsingAbstractClasses.cpp  
class base {  
public:  
    base() {}  
    virtual ~base()=0;  
};  
  
// Provide a definition for destructor.  
base::~base() {}  
  
class derived:public base {  
public:  
    derived() {}  
    ~derived(){}  
};  
  
int main() {  
    derived *pDerived = new derived;  
    delete pDerived;  
}
```

當 `pDerived` 所指向的物件已刪除時，就會呼叫 `derived` 類別的解構函式，然後呼叫 `base` 類別的解構函式。純虛擬函式的空白實作可確保函式至少有某種實作存在。

#### NOTE

在上述範例中，純虛擬函式 `base::~base` 是從 `derived::~derived` 隱含呼叫。另外也可以使用完整限定的成員函式名稱明確呼叫純虛擬函式。

## 另請參閱

[繼承](#)

# 範圍規則摘要

2020/11/2 • • [Edit Online](#)

使用的名稱在其範圍內不可以模稜兩可 (其位置由多載決定)。如果名稱表示一個函式，該函式必須明確指定參數的數目和類型。如果名稱保持不明確，則會套用[成員存取](#)規則。

## 建構函式初始設定式

函式[初始化運算式](#)會在所指定之函式的最外層區塊範圍中進行評估。因此，它們可以使用建構函式的參數名稱。

## 全域名稱

物件、函式或列舉程式若是在任何函式或類別之外引入，或是加上全域一元範圍運算子 ( :: ) 前置詞，且未與下列任何二元運算子搭配使用，其名稱即為全域名稱：

- 範圍解析 ( :: )
- 物件和參考的成員選取 ( . )
- 指標的成員選取 ( -> )

## 限定名稱

搭配二進位範圍解析運算子 ( :: ) 使用的名稱亦稱為「限定名稱」(Qualified Name)。在二進位範圍解析運算子後面指定的名稱，必須是運算子左方所指定類別的成員，或是其基底類別的成員。

成員選取運算子 ( . ) 之後指定的名稱。或 -> ) 必須是運算子左邊所指定物件之類別類型的成員，或是其基類的成員。在成員選取運算子 ( . ) 的右邊指定的名稱 -> 也可以是另一個類別類型的物件，前提是的左邊 -> 是類別物件，且該類別定義的多載成員選取運算子 ( -> ) 會評估為其他類別類型的指標。(在[類別成員存取](#)中，將會更詳細地討論此布建)。

編譯器會依照下列順序搜尋名稱，並且在找到名稱時停止：

1. 如果名稱是在函式內使用，則為目前區塊範圍，否則為全域範圍。
2. 向外至每個封閉區塊範圍，包括最外層的函式範圍 (包括函式參數)。
3. 如果名稱是在成員函式內使用，則會在類別的範圍內搜尋名稱。
4. 在類別的基底類別內搜尋名稱。
5. 在封閉巢狀類別範圍 (如果有的話) 及其基底內搜尋。搜尋會繼續，直到搜尋至最外層封閉類別範圍為止。
6. 在全域範圍內搜尋。

不過，您可以依照下述方式修改這個搜尋順序：

1. 前面加上 :: 的名稱會強制從全域範圍開始搜尋。
2. 前面加 class 上、和關鍵字的名稱，會 struct union 強制編譯器只搜尋 class 、 struct 或 union 名稱。
3. 範圍解析運算子 () 左邊的名稱只能 :: 是 class 、 struct 、 namespace 或 union 名稱。

如果名稱參考非靜態成員，但是在靜態成員函式中使用，則會產生錯誤訊息。同樣地，如果名稱參考封入類別中的任何非靜態成員，就會產生錯誤訊息，因為括住的類別沒有封閉式類別 this 指標。

## 函式參數名稱

函式定義中的函式參數名稱會視為在函式最外層區塊的範圍內。因此，它們會是區域名稱且當函式結束時會超出範圍。

函式宣告 (原型) 中的函式參數名稱會位於宣告的區域範圍，而在宣告的結尾會超出範圍。

預設參數會位於其所預設的參數範圍內，如上兩個段落中所述。但是，它們無法存取區域變數或非靜態類別成員。預設參數是在函式呼叫時進行評估，但是它們是在函式宣告的原始範圍中進行評估。因此，成員函式的預設參數一定是在類別範圍中進行評估。

## 另請參閱

[繼承](#)

# 繼承關鍵字

2020/11/2 • [Edit Online](#)

Microsoft 特定的

```
class [__single_inheritance] class-name;
class [__multiple_inheritance] class-name;
class [__virtual_inheritance] class-name;
```

其中：

## 類別名稱

所要宣告類別的名稱。

C++ 可讓您在定義類別之前宣告類別成員的指標。例如：

```
class S;
int S::*p;
```

在上述程式碼中，會宣告為 `p` 類別 S 的整數成員指標。不過，尚未 `class S` 在這個程式碼中定義，只會宣告。當編譯器遇到這類指標時，必須將指標的表示法一般化。表示法的大小取決於指定的繼承模型。有四種方式可將繼承模型指定至編譯器：

- 在 IDE 的 [成員指標標記法] 底下
- 在命令列中使用 `/vmg` 參數
- 使用 `pointers_to_members` pragma
- 使用繼承關鍵字 `__single_inheritance`、`__multiple_inheritance` 和 `__virtual_inheritance`。這項技術能夠以每個類別為基礎控制繼承模型。

## NOTE

如果您總是在定義類別之後宣告類別成員的指標，就不需要使用上述任何選項。

在類別定義之前宣告類別成員的指標，會影響所產生可執行檔的大小和速度。類別使用的繼承越複雜，用來表示類別成員指標所需的位元組數目就越多，而且用來解譯指標的程式碼也會越大。單一繼承最不複雜，而虛擬繼承最為複雜。

如果上述範例變更為：

```
class __single_inheritance S;
int S::*p;
```

無論命令列選項或 pragma 為何，`class S` 成員的指標都會盡可能使用最小的表示法。

**NOTE**

類別成員指標表示法的同一個向前宣告應出現在宣告該類別成員指標的每一個轉譯單位中，而且宣告應在成員指標宣告之前發生。

為了與舊版相容，`_single_inheritance`、`_multiple_inheritance`和`_virtual_inheritance`是、和的同義字  
`_single_inheritance`，`_multiple_inheritance` `_virtual_inheritance` 除非指定了編譯器選項/za (停用語言擴充功能)。

結束 Microsoft 專有

另請參閱

[關鍵字](#)

# virtual (C++)

2020/11/2 • [Edit Online](#)

關鍵字會宣告虛擬函式 `virtual` 或虛擬基類。

## 語法

```
virtual [type-specifiers] member-function-declarator  
virtual [access-specifier] base-class-name
```

參數

類型規範

指定虛擬成員函式的傳回型別。

成員函式-宣告子

宣告成員函式。

存取規範

定義基類、或的存取層級 `public` `protected` `private`。可以出現在關鍵字之前或之後 `virtual`。

基本類別-名稱

識別先前宣告的類別類型。

## 備註

如需詳細資訊，請參閱虛擬函式。

另請參閱下列關鍵字：[class](#)、[private](#)、[public](#)和[protected](#)。

## 另請參閱

[關鍵字](#)

\_\_super

2020/11/2 • [Edit Online](#)

## Microsoft 特定的

可讓您明確陳述您要呼叫將覆寫之函式的基底類別實作。

## 語法

```
__super::member_function();
```

## 備註

多載解析階段會考量所有可存取的基底類別方法，並且提供最佳相符結果的函式即為將會呼叫的函式。

`__super` 只能出現在成員函式的主體內。

`__super` 不能與 `using` 声明搭配使用。如需詳細資訊，請參閱[using 声明](#)。

隨著引入了插入程式碼的[屬性](#)，您的程式碼可能會包含一個或多個基類，而您可能不知道這些類別的名稱，但其中包含您想要呼叫的方法。

## 範例

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

結束 Microsoft 專有

## 另請參閱

[關鍵字](#)

# \_\_interface

2020/11/2 • [Edit Online](#)

## Microsoft 特定的

Microsoft c + + 介面可以定義如下：

- 能夠繼承自零個或多個基底介面。
- 不能繼承自基底類別。
- 只能包含公用的純虛擬方法。
- 不能包含建構函式、解構函式或運算子。
- 不能包含靜態方法。
- 不能包含資料成員；可以使用屬性。

## 語法

```
modifier __interface interface-name {interface-definition};
```

## 備註

C + + 類別或結構可以使用這些規則來執行，但會 `__interface` 加以強制。

例如，以下是介面定義範例：

```
__interface IMyInterface {
    HRESULT CommitX();
    HRESULT get_X(BSTR* pbstrName);
};
```

如需 managed 介面的詳細資訊，請參閱[介面類別](#)。

請注意，您不需要明確表明 `CommitX` 和 `get_X` 函式是純虛擬函式。第一個函式的同等宣告如下：

```
virtual HRESULT CommitX() = 0;
```

`__interface` 意指[novtable](#) `__declspec` 修飾詞。

## 範例

下列範例示範如何使用在介面中宣告的屬性。

```
// deriv_interface.cpp
#define _ATL_ATTRIBUTES 1
#include <atlbase.h>
#include <atlcom.h>
#include <string.h>
#include <comdef.h>
#include <stdio.h>
```

```

[module(name="test")];

[ object, uuid("00000000-0000-0000-0000-000000000001"), library_block ]
__interface IFace {
    [ id(0) ] int int_data;
    [ id(5) ] BSTR bstr_data;
};

[ coclass, uuid("00000000-0000-0000-0000-000000000002") ]
class MyClass : public IFace {
private:
    int m_i;
    BSTR m_bstr;

public:
    MyClass()
    {
        m_i = 0;
        m_bstr = 0;
    }

    ~MyClass()
    {
        if (m_bstr)
            ::SysFreeString(m_bstr);
    }

    int get_int_data()
    {
        return m_i;
    }

    void put_int_data(int _i)
    {
        m_i = _i;
    }

    BSTR get_bstr_data()
    {
        BSTR bstr = ::SysAllocString(m_bstr);
        return bstr;
    }

    void put_bstr_data(BSTR bstr)
    {
        if (m_bstr)
            ::SysFreeString(m_bstr);
        m_bstr = ::SysAllocString(bstr);
    }
};

int main()
{
    _bstr_t bstr("Testing");
    CoInitialize(NULL);
    CComObject<MyClass>* p;
    CComObject<MyClass>::CreateInstance(&p);
    p->int_data = 100;
    printf_s("p->int_data = %d\n", p->int_data);
    p->bstr_data = bstr;
    printf_s("bstr_data = %S\n", p->bstr_data);
}

```

```

p->int_data = 100
bstr_data = Testing

```

結束 Microsoft 專有

另請參閱

[關鍵字](#)

[介面屬性](#)

# 特殊成員函式

2020/3/25 • [Edit Online](#)

特殊成員函式是類別(或結構)成員函式，在某些情況下，編譯器會自動為您產生。這些函數是預設的函式、「析構函數」、「複製」「作業」和「複製指派運算子」，以及「移動」和「移動指派運算子」。如果您的類別未定義一或多個特殊成員函式，則編譯器可能會隱含宣告並定義所使用的函數。編譯器產生的實值稱為預設的特殊成員函式。如果不需要，編譯器不會產生函數。

您可以使用 = `default` 關鍵字，明確宣告預設的特殊成員函式。如此一來，編譯器就只會在必要時定義函式，就像函式完全沒有宣告一樣。

在某些情況下，編譯器可能會產生已刪除的特殊成員函式，這些函式尚未定義，因此無法呼叫。如果類別的其他屬性在類別上呼叫特定特殊成員函式沒有意義，就可能發生這種情況。若要明確避免自動產生特殊成員函式，您可以使用 = `delete` 關鍵字將它宣告為 deleted。

編譯器會產生預設的函式，也就是不使用任何引數的函式，只有在您尚未宣告任何其他的函式時。如果您只宣告了接受參數的函式，則嘗試呼叫預設的函式的程式碼會導致編譯器產生錯誤訊息。編譯器產生的預設函式會針對物件執行簡單的成員預設初始化。預設初始化會將所有成員變數保留為不定狀態。

預設的析構函式會執行物件的成員的析構。只有當基類的析構函式為虛擬時，它才會是虛擬的。

預設的複製和移動結構和指派作業會執行成員取向的位模式複製或非靜態資料成員的移動。只有在未宣告任何析構函數或移動或複製作業時，才會產生移動作業。只有在未宣告任何複製的函式時，才會產生預設的複製函數。如果已宣告移動作業，則會隱含地刪除它。只有在未明確宣告任何複製指派運算子時，才會產生預設複製指派運算子。如果已宣告移動作業，則會隱含地刪除它。

## 另請參閱

[C++ 語言參考](#)

# 靜態成員 (C++)

2020/11/2 • [Edit Online](#)

類別可以包含靜態資料成員和成員函式。當資料成員宣告為時 `static`，只會針對類別的所有物件維護一個資料複本。

靜態資料成員不是特定類別類型之物件的一部分。因此，靜態資料成員的宣告不視為定義。資料成員是在類別範圍中宣告的，不過，定義是在檔案範圍執行。這些靜態成員具有外部連結。下列範例會加以說明：

```
// 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()
{
```

在上述程式碼中，成員 `bytecount` 是在類別 `BufferedOutput` 中宣告，但必須在類別宣告之外加以定義。

靜態資料成員可以在不參考類別類型物件的情況下參考。使用 `BufferedOutput` 物件撰寫的位元組數目可以如下取得：

```
long nBytes = BufferedOutput::bytecount;
```

若要靜態成員存在，不需要有類別類型的物件存在。也可以使用成員選取()來存取靜態成員。and -> )運算子。例如：

```
BufferedOutput Console;

long nBytes = Console.bytecount;
```

在上述情況中，不會評估物件 (`Console`) 的參考；傳回值是靜態物件 `bytecount` 的傳回值。

靜態資料成員是受類別成員存取規則規範，因此，只有類別成員函式和 friend 才能進行靜態資料成員的私用存取。[成員存取控制](#)中會描述這些規則。例外狀況是不管其存取限制，靜態資料成員必須在檔案範圍中定義。如果資料成員將明確初始化，定義中必須提供初始設定式。

靜態成員的類型未以類別名稱限定。因此，的類型 `BufferedOutput::bytecount` 為 `long`。

## 另請參閱

[類別和結構](#)

# 用為實值型別的 C++ 類別

2019/12/2 • [Edit Online](#)

C++ 類別預設為實數值型別。您可以將它們指定為參考型別，讓多型行為能夠支援物件導向程式設計。數值型別有時可以從記憶體和版面配置控制項的角度來查看，而參考型別則是關於基類和虛擬函式，以進行多型的用途。根據預設，實值型別為可複製，這表示一定會有複製的函式和複製指派運算子。針對參考型別，您可以將類別設為非可複製（停用複製程式設計函數和複製指派運算子），並使用支援其所需多型的虛擬析構函式。實值型別也是內容的相關資訊，當複製它們時，一律會提供兩個獨立的值供您分別修改。參考型別是關於身分識別—這是什麼類型的物件？因此，「參考型別」也稱為「多型類型」。

如果您真的想要類似參考的型別（基類、虛擬函式），您必須明確停用複製，如下列程式碼中的 `MyRefType` 類別所示。

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

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

public:
    MyRefType () {}
};

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

編譯上述程式碼將會產生下列錯誤：

```
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'
```

## 實數值型別和移動效率

由於新的複製優化，因此可避免複製配置額外負荷。例如，當您在字串向量的中間插入字串時，將不會有任何複製重新配置額外負荷，只有移動—即使它導致向量本身增加也一樣。這也適用於其他作業，例如在兩個非常大型的物件上執行 add 運算。如何啟用這些值作業優化？在某些C++編譯器中，編譯器會隱含地為您啟用此功能，就像複製函式可以由編譯器自動產生一樣。不過，在C++中，您的類別必須在類別定義中宣告，以「加入宣告」來移動指派和函數。在適當的成員函式宣告中使用 double 符號(&&)右值參考，並定義移動函數和移動指派方法，即可完成這項作業。您也必須插入正確的程式碼，以「竊取 getmembers」來源物件。

您要如何決定是否需要啟用移動？如果您已經知道您需要啟用「複製結構」，您可能會想要在可能比深層複本更便宜的情況下，啟用移動。不過，如果您知道需要移動支援，則不一定表示您想要啟用複製。第二種情況則稱為「僅限移動類型」。標準程式庫中已有一個範例 `unique_ptr`。請注意，舊的 `auto_ptr` 已被取代，而且已 `unique_ptr` 精確地取代，因為舊版中缺少 move 語義支援C++。

藉由使用移動語義，您可以傳回傳值或插入中間。Move 是複製的優化。需要堆積配置做為因應措施。請考慮下列虛擬程式碼：

```

#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

```

## 啟用適用於適當數值型別的移動

針對類似值的類別，移動可以比深層複本便宜。針對效率啟用移動結構和移動指派。請考慮下列虛擬程式碼：

```

#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!");
    }
};

```

如果您啟用 [複製結構/指派]，也可以啟用 [移動結構/指派]（如果它比深層複本便宜）。

某些非實值類型為僅限移動，例如當您無法複製資源時，只會傳送擁有權。範例：`unique_ptr`。

## 另請參閱

[C++ 類型系統](#)

[歡迎回到C++](#)

[C++ 語言參考](#)

[C++ 標準程式庫](#)

# 使用者定義類型轉換 (C++)

2020/11/2 • [Edit Online](#)

轉換會從不同類型的值產生某種類型的新值。標準轉換內建于 C++ 語言，並支援其內建類型，您可以建立使用者定義的轉換，以執行使用者定義類型之間的轉換。

標準轉換可以執行內建類型之間的轉換、有繼承關係之類型的指標或參考之間的轉換、轉換成 Void 指標或從 Void 指標轉換，以及轉換成 null 指標。如需詳細資訊，請參閱[標準轉換](#)。使用者定義轉換可執行使用者定義類型之間的轉換，或是使用者定義類型與內建類型之間的轉換。您可以將它們實作為轉換函式，或當做轉換函數來執行。

轉換可以是明確的（當程式設計人員呼叫一種類型以轉換成另一種類型，例如在轉型或直接初始化作業中）或隱含的（當語言或程式呼叫的類型不同於程式設計人員指定的類型時）。

在下列情況會嘗試隱含轉換：

- 提供給函式的引數沒有與對應參數相同的類型。
- 從函式傳回的值沒有與函式傳回型別相同的類型。
- 初始設定式運算式沒有與其初始化之物件相同的類型。
- 用來控制條件陳述式、迴圈建構或參數的運算式，沒有加以控制所需的結果類型。
- 提供給運算子的運算元沒有與對應運算元參數相同的類型。若為內建運算子，這兩種運算元必須具有相同的類型，並轉換成可代表兩者的共同類型。如需詳細資訊，請參閱[標準轉換](#)。若為使用者定義運算子，每個運算元都必須要有與對應運算元參數相同的類型。

當一個標準轉換無法完成隱含轉換時，編譯器可以使用使用者定義轉換，後面接選擇的其他標準轉換，以完成轉換。

當轉換網站上有兩個以上執行相同轉換的使用者定義轉換時，則將該轉換視為模稜兩可。這種模稜兩可的情況是種錯誤，因為編譯器無法判斷應選擇哪個可用的轉換。不過，這並不只是定義多種方式來執行相同轉換的錯誤，因為在原始程式碼不同位置的可用轉換集可能會不同，例如，取決於原始程式檔內含哪些標頭檔。只要轉換網站上只有一個可用的轉換，就不會模稜兩可。發生模稜兩可轉換的方式有好幾種，最常見的方式如下：

- 多重繼承。在多個基底類別中定義該轉換。
- 模稜兩可函式呼叫。轉換定義為目標類型的轉換建構函式，以及來源類型的轉換函式。如需詳細資訊，請參閱[轉換函式](#)。

只要更完整地限定相關類型的名稱，或是執行明確的轉型來釐清您的意圖，通常就能解決模稜兩可的問題。

轉換建構函式和轉換函式都遵循成員存取控制規則，但是只有在能夠判定明確轉換時，才會考慮轉換的存取範圍。這表示，即使競爭轉換的存取層級會防止轉換被使用，轉換還是可能會模稜兩可。如需成員存取範圍的詳細資訊，請參閱[成員存取控制](#)。

## 明確的關鍵字以及隱含轉換的問題

依預設，當您建立使用者定義的轉換時，編譯器可以用它來執行隱含的轉換。有時候您就是想這麼做，但有時候在進行隱含轉換時，用來指引編譯器的簡單規則會引導它接受您不想要讓它接受的程式碼。

轉換成的一個可能會造成問題的隱含轉換，就是其中一個知名的例子 `bool`。有許多原因會讓您建立可用於布林內容的類別型別（例如，它可以用來控制 `if` 語句或迴圈），但是當編譯器對內建型別執行使用者定義的轉換時，編譯器可以在之後套用其他標準轉換。這項額外標準轉換的目的是要允許從提升至的專案 `short` `int`，但

也會開啟較不明顯的轉換(例如從 `bool` 到 `int`)，讓您的類別類型可用於您從未預期的整數內容中。這個特定的問題就是所謂的安全 `Bool` 問題。這種問題是 `explicit` 關鍵字可以提供協助的地方。

`explicit` 關鍵字會告訴編譯器，指定的轉換無法用來執行隱含轉換。如果您想要在引進關鍵字之前隱含轉換的語法便利性 `explicit`，您必須接受隱含轉換有時候會建立的非預期結果，或使用較不方便的命名轉換函式做為因應措施。現在，藉由使用 `explicit` 關鍵字，您可以建立僅用於執行明確轉換或直接初始化的方便轉換，而不會導致安全的 `Bool` 問題所一例的問題種類。

`explicit` 關鍵字可以套用至 `c++98` 之後的轉換函式，以及自 `c++11` 起的轉換函式。下列各節包含如何使用關鍵字的詳細資訊 `explicit`。

## 轉換構造函式

轉換建構函式可定義從使用者定義或內建類型轉換成使用者定義類型的作業。下列範例示範從內建類型轉換 `double` 成使用者定義類型的轉換函式 `Money`。

```
#include <iostream>

class Money
{
public:
    Money() : amount{ 0.0 } {};
    Money(double _amount) : amount{ _amount } {};

    double amount;
};

void display_balance(const Money balance)
{
    std::cout << "The balance is: " << balance.amount << std::endl;
}

int main(int argc, char* argv[])
{
    Money payable{ 79.99 };

    display_balance(payable);
    display_balance(49.95);
    display_balance(9.99f);

    return 0;
}
```

請注意，第一次呼叫函式 `display_balance` (採用類型 `Money` 的引數) 時不需要轉換，因為其引數是正確的類型。不過，在第二次呼叫時 `display_balance`，需要進行轉換，因為引數的類型 (`double` 值為 `49.95`) 不是函數所預期。函式不能直接使用此值，但因為有從引數類型轉換 `double` 為相符參數的類型—`Money`-類型的暫存值 `Money` 是從引數所構成，並用來完成函式呼叫。在第三個呼叫中 `display_balance`，請注意，引數不是，而 `double` 是值為 `float` 的，`9.99` 而且函式呼叫仍然可以完成，因為編譯器可以執行標準轉換(在此案例中為)，然後從執行使用者定義的轉換，`float` `double` `double` `Money` 以完成必要的轉換。

### 宣告轉換建構函式

下列規則適用於宣告轉換建構函式：

- 轉換的目標類型是所建構的使用者定義類型。
- 轉換建構函式通常只會採用一個引數，屬於此來源類型。不過，如果其他每個參數都有預設值，轉換建構函式就可以指定其他參數。來源類型仍是第一個參數的類型。
- 轉換建構函式就像所有建構函式一樣，並沒有指定傳回型別。在宣告中指定傳回類型是一種錯誤。

- 轉換建構函式可以是明確的。

### 明確轉換建構函式

藉由將轉換的函式宣告為 `explicit`，它只能用來執行物件的直接初始化，或執行明確的轉換。如此可以防止接受類別型別引數的函式，不會也隱含地接受轉換建構函式來源類型的引數，並防止類別類型從來源類型的值以複製的方式初始化。下列範例示範如何定義明確的轉換建構函式，並示範它對何種程式碼產生格式正確的影響。

```
#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;
}
```

在此範例中，請注意，您仍可使用明確的轉換建構函式來執行 `payable` 的直接初始化。如果您要以複製方式初始化 `Money payable = 79.99;`，則會是個錯誤。第一次呼叫 `display_balance` 時並不受影響，因為引數是正確的類型。第二次呼叫 `display_balance` 時則是個錯誤，因為轉換建構函式不能用來執行隱含轉換。第三個對的呼叫 `display_balance` 是合法的，因為明確轉換為 `Money`，但請注意，編譯器仍然會藉由插入從到的隱含轉換，協助完成轉換 `float` `double`。

雖然允許隱含轉換的便利性很吸引人，但是這麼做會造成難以找出的 Bug。經驗告訴我們，除非您確定想要讓特定的轉換隱含發生，否則最好是讓所有轉換建構函式都很明確。

## 轉換函式

轉換函式可定義從使用者定義類型到其他類型的轉換作業。這些函式有時候是指「轉型運算子」，因為當值轉型為不同類型時，會將其連同轉換建構函式一起呼叫。下列範例示範從使用者自訂類型轉換 `Money` 成內建類型的轉換函數 `double`：

```

#include <iostream>

class Money
{
public:
    Money() : amount{ 0.0 } {};
    Money(double _amount) : amount{ _amount } {};

    operator double() const { return amount; }
private:
    double amount;
};

void display_balance(const Money balance)
{
    std::cout << "The balance is: " << balance << std::endl;
}

```

請注意，成員變數 `amount` 設為私用，而且會引進類型的公用轉換函式，`double` 只是為了傳回的值 `amount`。在函式 `display_balance` 中，使用資料流插入運算子 `<<` 將 `<<` 的值以資料流傳送至標準輸出時，會發生隱含轉換。因為沒有為使用者定義型別定義資料流程插入運算子 `Money`，但內建型別有一個，所以 `double` 編譯器可以使用從到的轉換函 `Money` `double` 式來滿足資料流程插入運算子。

衍生類別會繼承轉換函式。當衍生類別中的轉換函式轉換成完全相同的類型時，只會覆寫繼承的轉換函式。例如，衍生類別運算子 `int` 的使用者定義轉換函式不會覆寫(或甚至影響)基類運算子的使用者定義轉換函式，即使標準轉換定義和之間的轉換關聯性也一樣 `int` `short`。

## 宣告轉換函式

下列規則適用於宣告轉換函式：

- 必須先宣告轉換的目標類型，才能宣告轉換函式。不能在轉換函式的宣告中宣告類別、結構、列舉和 `typedef`。

```
operator struct String { char string_storage; }() // illegal
```

- 轉換函式不接受引數。在宣告中指定任何參數都是錯誤。
- 轉換函式具有以轉換函式名稱(也是轉換的目標類型名稱)指定的傳回型別。在宣告中指定傳回類型是一種錯誤。
- 轉換函式可以是虛擬的。
- 轉換函式可以是明確的。

## 明確轉換函數

當轉換函式宣告為明確時，就只能用來執行明確轉型。如此可以防止接受轉換函式目標型別引數的函式，不會也隱含地接受類別類型的引數，並防止目標類型的執行個體從類別類型的值以複製方式初始化。下列範例示範如何定義明確的轉換函式，並示範它對何種程式碼產生格式正確的影響。

```
#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;
}
```

在這裡，轉換函式運算子 `double` 已明確地進行，而且函式中已引進明確轉換成類型 `double` `display_balance` 來執行轉換。如果省略此轉型，編譯器會找不到適合類型 `<<` 的資料流插入運算子 `Money`，並且會發生錯誤。

# 可變動的資料成員 (C++)

2020/11/2 • [Edit Online](#)

這個關鍵字只能套用至非靜態和非常數類別的資料成員。如果宣告了資料成員 `mutable`，則從成員函式將值指派給這個資料成員是合法的 `const`。

## 語法

```
mutable member-variable-declaration;
```

## 備註

例如，下列程式碼會編譯而不會發生錯誤 `m_accessCount`，因為已宣告為 `mutable`，因此 `GetFlag` 即使 `GetFlag` 是 `const` 成員函式，也可以修改。

```
// mutable.cpp
class X
{
public:
    bool GetFlag() const
    {
        m_accessCount++;
        return m_flag;
    }
private:
    bool m_flag;
    mutable int m_accessCount;
};

int main()
{}
```

## 另請參閱

[關鍵字](#)

# 巢狀類別宣告

2020/11/2 • [Edit Online](#)

可以在某個類別的範圍內宣告另一個類別。這種類別稱為「巢狀類別」。巢狀類別被視為在封入類別的範圍內，可在該範圍內使用。若要在直接封入範圍以外參考巢狀類別，則必須使用完整名稱。

下列範例顯示如何宣告巢狀類別：

```
// 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` 會在中宣告 `BufferedIO`。這些類別名稱在類別 `BufferedIO` 的範圍外不會顯示。不過，類型為 `BufferedIO` 的物件不包含任何類型為 `BufferedInput` 或 `BufferedOutput` 的物件。

巢狀類別只能直接使用封入類別中的名稱、類型名稱、靜態成員的名稱及列舉程式。若要使用其他類別成員的名稱，您必須使用指標、參考或物件名稱。

在上述 `BufferedIO` 範例中，可以在巢狀類別、`IOError` 或 `BufferedIO::BufferedInput` 中直接存取列舉 `BufferedIO::BufferedOutput`，如函式 `good` 中所示。

## NOTE

巢狀類別只能宣告類別範圍中的類型。它們不會導致建立巢狀類別中所包含的物件。上述範例宣告兩個巢狀類別，但不會宣告這些類別類型的任何物件。

巢狀類別宣告範圍可視性的例外是同時宣告類型名稱和向前宣告。在這種情況下，向前宣告所宣告的類別名稱會在封入類別以外顯示，且其範圍定義為最小的封入非類別範圍。例如：

```
// 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;
}
```

## 巢狀類別中的存取權限

在某一個類別中將另一個類別設為巢狀，並不會對巢狀類別的成員函式提供特殊存取權限。同樣地，封入類別的成員函式對於巢狀類別的成員不具有特殊存取權限。

## 巢狀類別中的成員函式

在巢狀類別中宣告的成員函式可以在檔案範圍中定義。上一個範例可能已撰寫：

```

// 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()
{
}

```

在上述範例中，會使用限定型別名稱語法來宣告函數名稱。該宣告：

```
BufferedIO::BufferedInput::read()
```

表示「`read` 函式是 `BufferedInput` 類別的成員，而該類別在 `BufferedIO` 類別的範圍中」。因為此宣告使用了**限定型別名稱語法**，所以可能會有下列形式的結構：

```

typedef BufferedIO::BufferedInput BIO_INPUT;

int BIO_INPUT::read()

```

上述宣告相當於前一個宣告，但它會使用名稱來 `typedef` 取代類別名稱。

## 巢狀類別中的 friend 函式

在巢狀類別中宣告的 friend 函式會被視為在巢狀類別的範圍內，而非封入類別的範圍內。因此，friend 函式不會取得對封入類別之成員或成員函式的特殊存取權限。如果您想要使用在 friend 函式的巢狀類別中宣告的名稱，而且 friend 函式是在檔案範圍中定義，請使用**限定類型名稱**，如下所示：

```

// 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()
{
}

```

下列程式碼示範宣告為 friend 函式的 `GetExtendedErrorStatus` 函式。在檔案範圍內定義的函式中，訊息會從靜態陣列複製到類別成員。請注意，較理想的 `GetExtendedErrorStatus` 實作是將其宣告為：

```
int GetExtendedErrorStatus( char *message )
```

前述介面可讓數個類別藉由傳遞要將錯誤訊息複製到其中的記憶體位置，使用這個函式的服務。

## 另請參閱

[類別和結構](#)

# 匿名類別類型

2020/11/2 • [Edit Online](#)

類別可以是匿名的，也就是說，它們可以不使用識別碼來宣告。當您將類別名稱取代為名稱時，這會很有用 `typedef`，如下所示：

```
typedef struct
{
    unsigned x;
    unsigned y;
} POINT;
```

## NOTE

上述範例中的匿名類別用法對於保留與現有 C 程式碼的相容性而言很實用。在某些 C 程式碼中，使用 `typedef` 搭配匿名結構是普遍的。

另外，當您想要讓類別成員的參考出現，就像該參考不是包含在另一個類別中一般時，匿名類別也相當實用，如下所示：

```
struct PTValue
{
    POINT ptLoc;
    union
    {
        int iValue;
        long lValue;
    };
};

PTValue ptv;
```

在上述程式碼中，`iValue` 可以使用物件成員選取運算子`(.)`來存取，如下所示：

```
int i = ptv.iValue;
```

匿名類別會受到某些限制（如需匿名等位的詳細資訊，[請參閱等位](#)）。匿名類別：

- 不能具有建構函式或解構函式。
- 無法當做引數傳遞至函式（除非使用省略號來失效類型檢查）。
- 不可做為函式的傳回值傳回。

## 匿名結構

### Microsoft 特定的

Microsoft C 擴充功能可讓您在另一個結構內宣告結構變數，而不需為它命名。這些巢狀結構稱為匿名結構。C++ 不允許匿名結構。

您可以存取匿名結構的成員，就如同它們是包含結構中的成員。

```
// 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
```

結束 Microsoft 專有

# 成員的指標

2020/11/2 • [Edit Online](#)

成員指標宣告是特殊的指標宣告。它們是使用下列順序來宣告：

儲存類別指定名稱<sub>opt</sub>  $cv$  限定詞<sub>opt</sub> 類型規範<sub>ms</sub>-修飾詞<sub>opt</sub> 限定名稱<sub>opt</sub>  $::*$   $cv$  限定詞<sub>opt</sub> 識別碼<sub>pm</sub>-初始化運算式<sub>opt</sub> ;

## 1. 宣告規範：

- 選擇性的儲存類別規範。
- 選擇性 `const` 和規範 `volatile`。
- 類型指定名稱：類型的名稱。這是要指向的成員類型，而不是類別。

## 2. 宣告子：

- 選擇性的 Microsoft 專有修飾詞。如需詳細資訊，請參閱[Microsoft 專有的修飾詞](#)。
- 包含指向成員之類別的限定名稱。
- `::` 運算子。
- `*` 運算子。
- 選擇性 `const` 和規範 `volatile`。
- 為成員指標命名的識別項。

## 3. 選擇性的成員指標初始化運算式：

- `=` 運算子。
- `&` 運算子。
- 類別的限定名稱。
- `::` 運算子。
- 適當型別之類別的非靜態成員名稱。

一如往常，單一宣告允許使用多個宣告子（和任何關聯的初始設定式）。成員的指標可能不會指向類別的靜態成員、參考型別的成員，或 `void`。

類別成員的指標與一般指標不同：它具有成員類型的類型資訊，以及該成員所屬的類別。一般指標只能識別記憶體中的單一物件（能取得其位址）。類別的成員指標可以在類別的任何執行個體中識別該成員。下列範例宣告類別、`Window` 和某些成員資料指標。

```

// 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()
{
}

```

在上述範例中，`pwCaption` 是型別之任何類別成員的指標 `Window char*`。`pwCaption` 的類型是 `char * Window::*`。下一個程式碼片段會宣告 `SetCaption` 和 `GetCaption` 成員函式的指標。

```

const char * (Window::* pfnwGC)() = &Window::GetCaption;
bool (Window::* pfnwSC)( const char * ) = &Window::SetCaption;

```

`pfnwGC` 和 `pfnwSC` 指標分別指向 `GetCaption` 類別的 `SetCaption` 和 `Window`。程式碼會使用成員指標 `pwCaption`，將資訊直接複製到視窗標題：

```

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 `->*` 運算子(成員指標運算子)之間的差異在於，`.*` 運算子會選取指定物件或物件參考的成員，而 `->*` 運算子會透過指標選取成員。如需這些運算子的詳細資訊，請參閱[使用成員指標運算子的運算式](#)。

成員指標運算子的結果是成員的類型。在此案例中，此名稱為 `char *`。

下一個程式碼片段會使用成員指標叫用 `GetCaption` 和 `SetCaption` 成員函式：

```

// 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 );

```

## 成員指標的限制

靜態成員的位址不是成員的指標。這是一個靜態成員實例的一般指標。只有一個靜態成員的實例存在於指定類別的所有物件中。這表示您可以使用一般的位址( & )和引用( \* )運算子。

## 成員指標和虛擬函式

透過成員指標函式叫用虛擬函式的運作方式，如同已直接呼叫函式一樣。在 v 資料表中查閱正確的函式，並加以叫用。

就如以往一般，虛擬函式運作的關鍵在於透過基底類別的指標叫用（如需虛擬函式的詳細資訊，請參閱[虛擬函式](#)）。

下列程式碼將示範如何透過成員指標函式叫用虛擬函式：

```
// 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
```

# this 指標

2020/12/10 • [Edit Online](#)

`this` 指標是只能在 `class`、或類型的非靜態成員函式中存取的指標 `struct` `union`。它會指向針對其呼叫成員函式的物件。靜態成員函式沒有 `this` 指標。

## 語法

```
this  
this->member-identifier
```

## 備註

物件的 `this` 指標不是物件本身的一部分。它不會反映在 `sizeof` 物件上的語句結果中。針對物件呼叫非靜態成員函式時，編譯器會將物件的位址以隱藏引數的形式傳遞至函數。例如，下列函式呼叫：

```
myDate.setMonth( 3 );
```

可以解讀為：

```
setMonth( &myDate, 3 );
```

物件的位址可從成員函式中當做指標來取得 `this`。大部分 `this` 的指標使用都是隱含的。雖然不必要，但是 `this` 在參考的成員時，還是要使用明確的 `class`。例如：

```
void Date::setMonth( int mn )  
{  
    month = mn;           // These three statements  
    this->month = mn;     // are equivalent  
    (*this).month = mn;  
}
```

運算式 `*this` 通常用來從成員函式傳回目前的物件：

```
return *this;
```

`this` 指標也用來防範自我參考：

```
if (&Object != this) {  
// do not execute in cases of self-reference
```

### NOTE

因為指標是不可修改的 `this`，所以 `this` 不允許對指標的指派。舊版 C++ 允許指派至 `this`。

有時候，`this` 指標會直接使用，例如，若要操控自我參考資料 `structures`，其中需要目前物件的位址。

## 範例

```
// this_pointer.cpp
// compile with: /EHsc

#include <iostream>
#include <string.h>

using namespace std;

class Buf
{
public:
    Buf( char* szBuffer, size_t sizeOfBuffer );
    Buf& operator=( const Buf & );
    void Display() { cout << buffer << endl; }

private:
    char*    buffer;
    size_t   sizeOfBuffer;
};

Buf::Buf( char* szBuffer, size_t sizeOfBuffer )
{
    sizeOfBuffer++; // account for a NULL terminator

    buffer = new char[ sizeOfBuffer ];
    if (buffer)
    {
        strcpy_s( buffer, sizeOfBuffer, szBuffer );
        sizeOfBuffer = sizeOfBuffer;
    }
}

Buf& Buf::operator=( const Buf &otherbuf )
{
    if( &otherbuf != this )
    {
        if (buffer)
            delete [] buffer;

        sizeOfBuffer = strlen( otherbuf.buffer ) + 1;
        buffer = new char[sizeOfBuffer];
        strcpy_s( buffer, sizeOfBuffer, otherbuf.buffer );
    }
    return *this;
}

int main()
{
    Buf myBuf( "my buffer", 10 );
    Buf yourBuf( "your buffer", 12 );

    // Display 'my buffer'
    myBuf.Display();

    // assignment operator
    myBuf = yourBuf;

    // Display 'your buffer'
    myBuf.Display();
}
```

```
my buffer  
your buffer
```

## 指標的類型 this

`this` 指標的型別可以在函式宣告中，由 `const` 和關鍵字修改 `volatile`。若要宣告具有其中一個屬性的函式，請在函式引數清單之後新增關鍵字。

假設有一個範例：

```
// type_of_this_pointer1.cpp  
class Point  
{  
    unsigned X() const;  
};  
int main()  
{  
}
```

上述程式碼會宣告成員函式，`x` 其中 `this` 指標會被視為 `const` 物件的指標 `const`。您可以使用 *cv-mod-list* 選項的組合，但它們一律會修改指標所指向的物件 `this`，而不是指標本身。下列宣告會宣告 function `x`，其中 `this` 指標是物件的 `const` 指標 `const`：

```
// type_of_this_pointer2.cpp  
class Point  
{  
    unsigned X() const;  
};  
int main()  
{  
}
```

成員函式 `this` 中的類型是由下列語法所描述。*Cv 限定詞清單*是由成員函式的宣告子所決定。它可以是 `const` 或 `volatile`（或兩者）。\* class -type\* 是的名稱 class：

[*CV-辨識符號-清單*]\* class -類型\* \*\*\* `const this` \*\*

換句話說，`this` 指標一律是 `const` 指標。無法重新指派。成員函式宣告 `const` `volatile` 中使用的或限定詞適用於 class 指標指向的實例 `this`（在該函式的範圍內）。

下表將進一步說明這些修飾詞的運作方式。

### 修飾詞 this 的語法

III	II
<code>const</code>	無法變更成員資料;無法叫用不是的成員函式 <code>const</code> 。
<code>volatile</code>	成員資料會在每次存取時從記憶體載入;停用特定的優化。

將物件傳遞至不是的成員函式是錯誤的 `const` `const`。

同樣地，將物件傳遞至不是的成員函式也是錯誤 `volatile` `volatile`。

宣告為 `const` 無法變更成員資料的成員函式-在這類函式中，`this` 指標是 `const` 物件的指標。

#### NOTE

Con struct or 和 de struct or 不能宣告為 `const` 或 `volatile`。不過，它們可以在或物件上叫用 `const` `volatile`。

## 另請參閱

[關鍵字](#)

# C++ 位元欄位

2020/11/2 • [Edit Online](#)

類別和結構可包含比整數類型佔用較少儲存區的成員。這些成員指定為位元欄位。位欄位成員宣告子規格的語法如下：

## 語法

*declarator*宣告子：常數運算式

## 備註

(選擇性)宣告子是在程式中用來存取成員的名稱。它必須是整數類資料類型(包括列舉類型)。常數運算式會指定成員在結構中佔用的位數。匿名位元欄位(亦即沒有識別項的位元欄位成員)可用於填補。

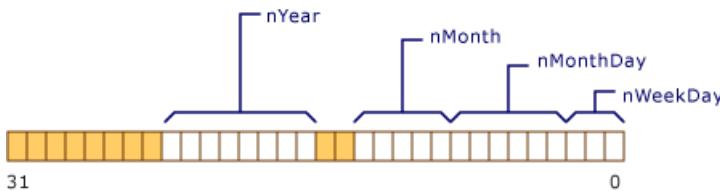
### NOTE

未命名的位欄位寬度0會強制將下一個位欄位對齊下一個■界限，其中*type*是成員的類型。

下列範例宣告包含位元欄位的結構：

```
// bit_fields1.cpp
// compile with: /LD
struct Date {
    unsigned short nYear : 8;      // 0..100 (8 bits)
    unsigned short nMonth : 5;     // 0..12 (5 bits)
    unsigned short nMonthDay : 6;   // 0..31 (6 bits)
    unsigned short nWeekDay : 3;    // 0..7 (3 bits)
};
```

Date 類型物件的概念性記憶體配置如下圖所示。



Date 物件的記憶體配置

請注意，`nYear`為8位長，且會溢位已宣告類型的文字界限 `unsigned short`。因此，它會在新的開頭開始 `unsigned short`。不需要所有位元欄位都調整至基礎類型的一個物件中；新的單位儲存根據宣告所要求的位元數目進行配置。

### Microsoft 特定的

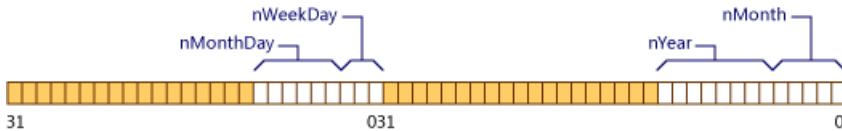
宣告為位元欄位的資料之順序是由低至高位元，如上面的圖所示。

### 結束 Microsoft 專有

如果結構的宣告包含未命名且長度為0的欄位，如下列範例所示：

```
// 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)
};
```

之後，記憶體配置如下圖所示：



具有長度為零之位元欄位的 Date 物件配置

位欄位的基礎類型必須是整數類資料類型，如內建類型中所述。

如果類型之參考的初始化運算式 `const T&` 是參考型別之位欄位的左值 `T`，則參考不會直接系結至位欄位。相反地，參考會系結至已初始化的暫存，以保存位欄位的值。

## 位元欄位的限制

下列清單詳細說明位元欄位的錯誤作業：

- 取得位元欄位的位址。
- 初始化 `const` 具有位欄位的非參考。

## 另請參閱

[類別和結構](#)

# C++ 中的 Lambda 運算式

2020/11/2 • [Edit Online](#)

在 C++ 11 和更新版本中, lambda 運算式(通常稱為 *lambda*)是一種便利的方式, 可在叫用或將它當做引數傳遞至函式的位置定義匿名函式物件(關閉)。Lambda 通常用來封裝要傳遞給演算法或非同步方法的數行程式碼。本文定義什麼是 Lambda、Lambda 與其他程式設計技術的比較、描述 Lamdba 的優點並提供基本範例。

## 相關主題

- [Lambda 運算式與函數物件的比較](#)
- [使用 lambda 運算式](#)
- [constexpr Lambda 運算式](#)

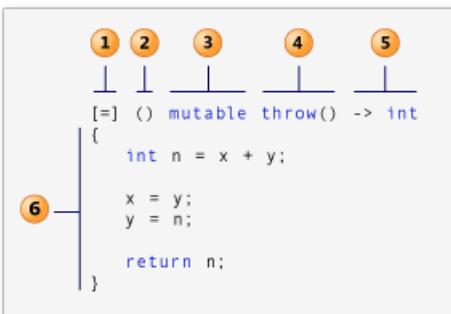
## Lambda 運算式的組件

ISO C++ 標準顯示將傳遞為 `std::sort()` 函式第三個引數的簡單 Lambda:

```
#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
    );
}
```

下圖顯示 Lambda 的組件:



1. *capture 子句*(在 C++ 規格中也稱為 *lambda introducer*)。
2. *參數清單選擇性*. (也稱為 *lambda 告白子*)
3. *可變規格選擇性*.
4. *例外狀況-規格選擇性*.
5. *尾端-傳回類型選擇性*.
6. *lambda 主體*。

## 擷取子句

Lambda 可以在其主體中引進新的變數(在c + + 14中),而且它也可以從周圍的範圍存取或**捕捉**變數。Lambda 的開頭是 capture 子句(標準語法中的*lambda-introducer*),它會指定要捕捉的變數,以及捕捉是以傳值或傳址方式。含有 `&` 前置詞的變數會以傳址方式來存取,而不含 `&` 前置詞的變數會以傳值方式來存取。

空白的擷取子句 `[]` 表示 Lambda 運算式主體不存取封閉範圍中的任何變數。

您可以使用預設的「捕捉模式」(標準語法中的「**捕捉-預設值**」)來指示如何捕捉 lambda 中所參考的任何外部變數：`[&]` 表示您參考的所有變數都是以傳址方式來捕捉，`[=]` 表示它們是以傳值方式來捕捉。您可以使用預設擷取模式，然後明確指定特定變數的相反模式。例如，如果 Lambda 主體以傳址方式存取外部變數 `total`，並以傳值方式存取外部變數 `factor`，則下列擷取子句相等：

```
[&total, factor]
[factor, &total]
[&, factor]
[factor, &]
[=, &total]
[&total, =]
```

使用 `capture-default` 時，只會捕捉 lambda 中提到的變數。

如果 capture 子句包含 `capture-default &`，則 `identifier capture` 該 capture 子句中的 no 會有此格式 `& identifier`。同樣地，如果 capture 子句包含 `capture-default =`，則該 capture 子句的任何一個都不 `capture` 能有此格式 `= identifier`。`this` 在 capture 子句中，識別碼或不能出現一次以上。下列程式碼片段說明部分範例。

```
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
}
```

後面接著省略號的 capture 就是套件擴充，如下列[variadic 範本](#)範例所示：

```
template<class... Args>
void f(Args... args) {
    auto x = [args...] { return g(args...); };
    x();
}
```

若要在類別方法主體中使用 lambda 運算式，請將 `this` 指標傳遞至 capture 子句，以提供對封入類別之方法和資料成員的存取權。

Visual Studio 2017 15.3 和更新版本(適用於[/std: c + + 17](#))：藉 `this` 由 `*this` 在 capture 子句中指定，可以藉由值來捕捉指標。`[以值方式捕捉]` 表示整個關閉(也就是 encapsulates lambda 運算式的匿名函式物件)會複製到叫用 lambda 的每一個呼叫位置。當 lambda 將以平行或非同步作業的方式執行時，Capture 會很有用，特別是在某些硬體架構(例如 NUMA)上。

如需示範如何使用 lambda 運算式搭配類別方法的範例，請參閱[Lambda 運算式的範例](#)中的「範例：在方法中使用 lambda 運算式」。

使用擷取子句時，建議您記住這幾個重點，特別是同時使用 Lambda 與多執行緒時：

- 傳址擷取可用來修改外部變數，但傳值擷取方式不能。( `mutable` 允許修改複本，而不是原件)。

- 傳址擷取方式會反映對外部變數的更新，但傳值擷取方式不會。
- 傳址擷取方式採用存留期相依性，但傳值擷取方式沒有存留期相依性。非同步執行 Lambda 時，這點特別重要。如果您在非同步 Lambda 中以傳址方式擷取區域變數，該區域變數極可能會在 Lambda 執行之前消失，導致在執行階段存取違規。

### 一般化擷取 (C++ 14)

在 C++14，您可以在擷取子句中引進並初始化新的變數，而這些變數不需要存在於 Lambda 函式的封閉範圍中。初始化可以表示為任何任意運算式；新變數的類型是透過運算式所產生的類型推斷而來。這項功能的其中一個優點是在 C++14 中，您可以從周圍範圍擷取僅移動變數（例如 std::unique\_ptr）並在 Lambda 中使用它們。

```
pNums = make_unique<vector<int>>(nums);
//...
auto a = [ptr = move(pNums)]()
{
    // use ptr
};
```

### 參數清單

除了擷取變數之外，Lambda 還可以接受輸入參數。參數清單（標準語法中的 *lambda* 告白子）是選擇性的，而且在大部分的方面類似于函式的參數清單。

```
auto y = [] (int first, int second)
{
    return first + second;
};
```

在 C++14 中，如果參數類型是泛型，您可以使用 `auto` 關鍵字做為類型規範。這會告訴編譯器建立函式呼叫運算子做為樣板。參數清單中的每個實例 `auto` 都相當於不同的類型參數。

```
auto y = [] (auto first, auto second)
{
    return first + second;
};
```

Lambda 運算式可接受另一個 Lambda 運算式當做其引數。如需詳細資訊，請參閱 [Lambda 運算式的範例](#) 主題中的「高階 Lambda 運算式」。

由於參數清單是選擇性的，因此如果您未將引數傳遞至 lambda 運算式，而且其 lambda 告白子不包含 `例外狀況規格`、`尾端傳回類型` 或 `mutable`，則可以省略空括弧 `mutable`。

### 可變動規格

一般來說，lambda 的函式呼叫運算子是逐值的，但使用 `mutable` 關鍵字會取消這項操作。它不會產生可變動的資料成員。可變動規格可讓 Lambda 運算式主體修改以傳值方式擷取的變數。本文稍後的部分範例會示範如何使用 `mutable`。

### 例外狀況規格

您可以使用 `noexcept` 例外狀況規格來表示 lambda 運算式不會擲回任何例外狀況。如同一般函式，如果 lambda 運算式宣告例外狀況規格，而且 lambda 主體擲回例外狀況，Microsoft C++ 編譯器就會產生警告 C4297 `noexcept`，如下所示：

```
// throw_lambda_expression.cpp
// compile with: /W4 /EHsc
int main() // C4297 expected
{
    []() noexcept { throw 5; }();
}
```

如需詳細資訊，請參閱[例外狀況規格 \(throw\)](#)。

## 傳回類型

Lambda 運算式的傳回型別會自動推算出來。您不需要使用關鍵字，`auto` 除非您指定尾端傳回類型。尾端傳回型別與一般方法或函式的傳回型別部分類似。不過，傳回型別必須接在參數清單之後，而且您必須在傳回型別前面加上尾端傳回型別關鍵字 `->`。

如果 Lambda 主體包含單一 return 陳述式，或者運算式不會傳回值，您可以省略 Lambda 運算式的傳回類型部分。如果 Lambda 主體包含單一 return 陳述式，編譯器會從傳回運算式的類型推算傳回型別。否則，編譯器會將傳回型別會推算為 `void`。請考慮下列說明此原則的範例程式碼片段。

```
auto x1 = [](int i){ return i; }; // OK: return type is int
auto x2 = []{ return{ 1, 2 }; }; // ERROR: return type is void, deducing
                                // return type from braced-init-list is not valid
```

Lambda 運算式可能會產生另一個 Lambda 運算式當做其傳回值。如需詳細資訊，請參閱[Lambda 運算式的範例中的「高階 Lambda 運算式」](#)。

## Lambda 主體

Lambda 運算式的 lambda 主體(標準語法中的複合陳述式)可以包含一般方法或函式主體可包含的任何內容。一般函式和 Lambda 運算式的主體都可以存取下列類型的變數：

- 摷取自封閉範圍的變數 (如先前所述)。
- 參數
- 區域宣告變數
- 類別資料成員 (在類別內部宣告並加以 `this` 捕捉)
- 有靜態儲存期的任何變數 (例如，全域變數)

下列範例包含 Lambda 運算式，這個運算式會以傳值方式明確摷取變數 `n`，並以傳址方式隱含摷取變數 `m`：

```
// captures_lambda_expression.cpp
// compile with: /W4 /EHsc
#include <iostream>
using namespace std;

int main()
{
    int m = 0;
    int n = 0;
    [&, n] (int a) mutable { m = ++n + a; }(4);
    cout << m << endl << n << endl;
}
```

```
5
0
```

因為變數 `n` 是以傳值方式擷取，其值在 Lambda 運算式呼叫之後會保持為 `0`。此 `mutable` 規格允許在 `n` lambda 中修改。

雖然 Lambda 運算式只能擷取有自動儲存期的變數，但是您可以在 Lambda 運算式的主體中使用有靜態儲存期的變數。下列範例使用 `generate` 函式和 Lambda 運算式，將值指派給 `vector` 物件的每個項目。Lambda 運算式會修改這個靜態變數以產生下一個項目的值。

```
void fillVector(vector<int>& v)
{
    // A local static variable.
    static int nextValue = 1;

    // The lambda expression that appears in the following call to
    // the generate function modifies and uses the local static
    // variable nextValue.
    generate(v.begin(), v.end(), [] { return nextValue++; });
    //WARNING: this is not thread-safe and is shown for illustration only
}
```

如需詳細資訊，請參閱[產生](#)。

下列程式碼範例會使用上一個範例中的函式，並加入使用 C++ 標準程式庫演算法的 lambda 運算式範例 `generate_n`。此 Lambda 運算式會將 `vector` 物件的元素指派給前兩個元素的總和。`mutable` 使用關鍵字，因此 lambda 運算式的主體可以修改其外部變數和的複本，而 lambda 運算式會以傳 `x` `y` 值方式來捕捉。由於 Lambda 運算式會以傳值方式擷取原始變數 `x` 和 `y`，在 Lambda 執行後，它們的值仍然保持 `1`。

```
// compile with: /W4 /EHsc
#include <algorithm>
#include <iostream>
#include <vector>
#include <string>

using namespace std;

template <typename C> void print(const string& s, const C& c) {
    cout << s;

    for (const auto& e : c) {
        cout << e << " ";
    }

    cout << endl;
}

void fillVector(vector<int>& v)
{
    // A local static variable.
    static int nextValue = 1;

    // The lambda expression that appears in the following call to
    // the generate function modifies and uses the local static
    // variable nextValue.
    generate(v.begin(), v.end(), [] { return nextValue++; });
    //WARNING: this is not thread-safe and is shown for illustration only
}

int main()
{
    // The number of elements in the vector.
    const int elementCount = 9;

    // Create a vector object with each element set to 1.
    vector<int> v(elementCount, 1);
```

```

// These variables hold the previous two elements of the vector.
int x = 1;
int y = 1;

// Sets each element in the vector to the sum of the
// previous two elements.
generate_n(v.begin() + 2,
    elementCount - 2,
    [=]() mutable throw() -> int { // lambda is the 3rd parameter
        // Generate current value.
        int n = x + y;
        // Update previous two values.
        x = y;
        y = n;
        return n;
    });
print("vector v after call to generate_n() with lambda: ", v);

// Print the local variables x and y.
// The values of x and y hold their initial values because
// they are captured by value.
cout << "x: " << x << " y: " << y << endl;

// Fill the vector with a sequence of numbers
fillVector(v);
print("vector v after 1st call to fillVector(): ", v);
// Fill the vector with the next sequence of numbers
fillVector(v);
print("vector v after 2nd call to fillVector(): ", v);
}

```

```

vector v after call to generate_n() with lambda: 1 1 2 3 5 8 13 21 34
x: 1 y: 1
vector v after 1st call to fillVector(): 1 2 3 4 5 6 7 8 9
vector v after 2nd call to fillVector(): 10 11 12 13 14 15 16 17 18

```

如需詳細資訊，請參閱[generate\\_n](#)。

## constexpr lambda 運算式

Visual Studio 2017 15.3 版和更新版本(可與搭配使用 [/std:c++17](#))：在常數運算式中允許將 lambda 運算式宣告為，或在常數運算式中 `constexpr` 使用它所捕捉或引進之每個資料成員的初始化時，可能會將它宣告為或用來。

```

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 的結果符合函式的需求，則會隱含地 `constexpr` 執行此動作 `constexpr`：

```
auto answer = [](int n)
{
    return 32 + n;
};

constexpr int response = answer(10);
```

如果 lambda 隱含或明確 `constexpr`，轉換成函式指標會產生 `constexpr` 函數：

```
auto Increment = [](int n)
{
    return n + 1;
};

constexpr int(*inc)(int) = Increment;
```

## Microsoft 專有

下列 common language runtime (CLR) 受管理的實體不支援 lambda: `ref class`、`ref struct`、`value class` 或 `value struct`。

如果您使用的是 Microsoft 專有的修飾詞(例如 `__declspec`)，您可以將其插入 lambda 運算式中的後面，例如 `parameter-declaration-clause`：

```
auto Sqr = [](int t) __declspec(code_seg("PagedMem")) -> int { return t*t; };
```

若要判斷 lambda 是否支援修飾詞，請參閱檔中[Microsoft 專有的修飾詞](#)一節中的相關文章。

除了 c++ 11 標準 lambda 功能之外，Visual Studio 支援無狀態 lambda，其可全轉換為使用任意呼叫慣例的函式指標。

## 另請參閱

[C++ 語言參考](#)

[C++ 標準程式庫中的函式物件](#)

[函式呼叫](#)

[for\\_each](#)

# Lambda 運算式語法

2020/11/2 • [Edit Online](#)

本文示範 Lambda 運算式的語法和結構化項目。如需 lambda 運算式的描述，請參閱 [Lambda 運算式](#)。

## 函式物件與 Lambda

當您撰寫程式碼時，您可能會使用函式指標和函式物件來解決問題和執行計算，尤其是當您使用 [C++ 標準程式庫演算法](#) 時。函式指標和函式物件各有優缺點。例如，函式指標的語法額外負荷最小，但是無法在範圍內保留狀態，而函式物件則可以維護狀態但無法避免需要定義類別的語法額外負荷。

Lambda 結合了函式指標和函式物件的優點並避免其缺點。Lambda 像函式物件一樣具有彈性並可以維護狀態，但不同於函式物件，由於語法精簡，因此不需要明確類別定義。使用 Lambda 時，您可以撰寫相對函式物件而言較不麻煩且較不容易發生錯誤的程式碼。

下列範例比較使用函式物件與使用 Lambda。第一個範例使用 Lambda 將 `vector` 物件中每個偶數和奇數項目列印到主控台。第二個範例使用函式物件完成相同的工作。

## 範例 1：使用 Lambda

此範例會將 lambda 傳遞至 `for_each` 函數。Lambda 會列印結果，其陳述 `vector` 物件中的每個項目是偶數還是奇數。

### 程式碼

```
// even_lambda.cpp
// compile with: cl /EHsc /nologo /W4 /MTd
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    // Create a vector object that contains 9 elements.
    vector<int> v;
    for (int i = 1; i < 10; ++i) {
        v.push_back(i);
    }

    // Count the number of even numbers in the vector by
    // using the for_each function and a lambda.
    int evenCount = 0;
    for_each(v.begin(), v.end(), [&evenCount] (int n) {
        cout << n;
        if (n % 2 == 0) {
            cout << " is even " << endl;
            ++evenCount;
        } else {
            cout << " is odd " << endl;
        }
    });

    // Print the count of even numbers to the console.
    cout << "There are " << evenCount
        << " even numbers in the vector." << endl;
}
```

```
1 is odd
2 is even
3 is odd
4 is even
5 is odd
6 is even
7 is odd
8 is even
9 is odd
There are 4 even numbers in the vector.
```

## 註解

在此範例中，`for_each` 函數的第三個引數是 lambda。`[&evenCount]` 組件指定運算式的擷取子句、`(int n)` 指定參數清單，而其餘組件則指定運算式的主體。

## 範例 2：使用函式物件

有時候，Lambda 的靈巧度較差，因此擴充程度無法超越上一個範例。下一個範例會使用函式物件（而不是 lambda）搭配 `for_each` 函式來產生與範例 1 相同的結果。這兩個範例都會將偶數計數儲存在 `vector` 物件中。為了維護作業的狀態，`FunctorClass` 類別會以傳址方式儲存 `m_evenCount` 變數做為成員變數。若要執行此作業，請 `FunctorClass` 將函式呼叫運算子運算子 `**( # B1 **)`。Microsoft C++ 編譯器產生的程式碼，與範例 1 中的 lambda 程式碼大小和效能相當類似。就本文所述此類基礎問題而言，較簡單的 Lambda 設計應該比函式物件的設計好。不過，如果您認為未來可能會需要大幅擴充，那麼使用函式物件設計會讓程式碼較容易維護。

如需 `**` 運算子 (`# B1 **`) 的詳細資訊，請參閱 [函式呼叫](#)。如需 `for_each` 函數的詳細資訊，請參閱 [for\\_each](#)。

## 程式碼

```

// even_functor.cpp
// compile with: /EHsc
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;

class FunctorClass
{
public:
    // The required constructor for this example.
    explicit FunctorClass(int& evenCount)
        : m_evenCount(evenCount) { }

    // The function-call operator prints whether the number is
    // even or odd. If the number is even, this method updates
    // the counter.
    void operator()(int n) const {
        cout << n;

        if (n % 2 == 0) {
            cout << " is even " << endl;
            ++m_evenCount;
        } else {
            cout << " is odd " << endl;
        }
    }

private:
    // Default assignment operator to silence warning C4512.
    FunctorClass& operator=(const FunctorClass&);

    int& m_evenCount; // the number of even variables in the vector.
};

int main()
{
    // Create a vector object that contains 9 elements.
    vector<int> v;
    for (int i = 1; i < 10; ++i) {
        v.push_back(i);
    }

    // Count the number of even numbers in the vector by
    // using the for_each function and a function object.
    int evenCount = 0;
    for_each(v.begin(), v.end(), FunctorClass(evenCount));

    // Print the count of even numbers to the console.
    cout << "There are " << evenCount
        << " even numbers in the vector." << endl;
}

```

```

1 is odd
2 is even
3 is odd
4 is even
5 is odd
6 is even
7 is odd
8 is even
9 is odd
There are 4 even numbers in the vector.

```

## 另請參閱

[Lambda 運算式](#)

[Lambda 運算式的範例](#)

[生成](#)

[generate\\_n](#)

[for\\_each](#)

[例外狀況規格 \(擲回\)](#)

[編譯器警告 \(層級 1\) C4297](#)

[Microsoft 專有的修飾詞](#)

# Lambda 運算式的範例

2020/11/2 • [Edit Online](#)

本文說明如何在您的程式中使用 Lambda 運算式。如需 lambda 運算式的總覽，請參閱[Lambda 運算式](#)。如需 lambda 運算式結構的詳細資訊，請參閱[Lambda 運算式語法](#)。

## 宣告 Lambda 運算式

### 範例 1

因為 lambda 運算式具有類型，所以您可以將它指派給 `auto` 變數或 `function` 物件，如下所示：

#### 程式碼

```
// 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;
}
```

#### 輸出

```
5
7
```

#### 備註

如需詳細資訊，請參閱 `auto`、`function` 類別和函式呼叫。

雖然 Lambda 運算式最常在函式的主體中宣告，但您也可以在可初始化變數的任何位置宣告 Lambda 運算式。

### 範例 2

當宣告運算式時，Microsoft C++ 編譯器會將 lambda 運算式系結至其已捕捉的變數，而不是呼叫運算式時。下列範例示範 Lambda 運算式以傳值方式擷取區域變數 `i`，以及以傳址方式擷取區域變數 `j`：因為 Lambda 運算式是以傳值方式擷取 `i` 的值，因此之後在程式中重新指派 `i` 的值並不會影響運算式的結果。不過，因為 Lambda 運算式是以傳址方式擷取 `j` 的值，因此之後重新指派 `j` 的值會影響運算式的結果。

#### 程式碼

```
// 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;
}
```

## 輸出

```
47
```

[\[在本文中\]](#)

## 呼叫 Lambda 運算式

如下程式碼片段所示，您可以立即呼叫 Lambda 運算式。第二個程式碼片段顯示如何將 lambda 當做引數傳遞至 C++ 標準程式庫演算法，例如 `find_if`。

### 範例 1

此範例會宣告一個 Lambda 運算式，此運算式會傳回兩個整數相加的總和並立即以引數 5 和 4 呼叫運算式：

#### 程式碼

```
// 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;
}
```

## 輸出

```
9
```

### 範例 2

此範例將 Lambda 運算式當做引數傳遞至 `find_if` 函式。如果 lambda 運算式的 `true` 參數是偶數，則會傳回此值。

## 程式碼

```
// 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;
    }
}
```

## 輸出

```
The first even number in the list is 42.
```

## 備註

如需函式的詳細資訊 `find_if`，請參閱 [find\\_if](#)。如需執行一般演算法之 C++ 標準程式庫函式的詳細資訊，請參閱 [<algorithm>](#)。

[[在本文中](#)]

## 嵌套 Lambda 運算式

### 範例

如本範例所示，您可以在 Lambda 運算式中與另一個 Lambda 運算式形成巢狀。內部的 Lambda 運算式會將其引數乘以 2 並傳回結果。外部的 Lambda 運算式會以內部 Lambda 運算式的引數呼叫該運算式並將結果加上 3。

## 程式碼

```
// 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;
}
```

## 輸出

```
13
```

## 備註

在此範例中, `[](int y) { return y * 2; }` 是巢狀 Lambda 運算式。

[\[在本文中\]](#)

## 高階 Lambda 函數

### 範例

許多程式設計語言都支援更高順序函式的概念。高階函式是以另一個 Lambda 運算式為其引數或傳回 Lambda 運算式的 Lambda 運算式。您可以使用 `function` 類別，讓 C++ lambda 運算式的行為類似于高階函數。下列範例說明傳回 `function` 物件的 Lambda 運算式，以及使用 `function` 物件做為其引數的 Lambda 運算式。

### 程式碼

```

// 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;
}

```

## 輸出

30

[\[在本文中\]](#)

## 在函式中使用 Lambda 運算式

### 範例

您可以在函式的主體中使用 Lambda 運算式。Lambda 運算式可以存取封入函式能夠存取的任何函式或資料成員。您可以明確或隱含地捕捉 `this` 指標，以提供封入類別之函式和資料成員的存取權。Visual Studio 2017 15.3 版和更新版本(適用于 `/std:c++17`)：`this` `[*this]` 當 lambda 將用於非同步或並行作業，而該程式碼可能會在原始物件超出範圍之後執行時，以值 `Capture()`。

您可以 `this` 在函式中明確使用指標，如下所示：

```

// 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; });
}

```

您也可以 `this` 隱含地捕捉指標：

```
void ApplyScale(const vector<int>& v) const
{
    for_each(v.begin(), v.end(),
        [=](int n) { cout << n * _scale << endl; });
}
```

下列範例示範封裝小數位數值的 `Scale` 類別。

```
// function_lambda_expression.cpp
// compile with: /EHsc /W4
#include <algorithm>
#include <iostream>
#include <vector>

using namespace std;

class Scale
{
public:
    // The constructor.
    explicit Scale(int scale) : _scale(scale) {}

    // Prints the product of each element in a vector object
    // and the scale value to the console.
    void ApplyScale(const vector<int>& v) const
    {
        for_each(v.begin(), v.end(), [=](int n) { cout << n * _scale << endl; });
    }

private:
    int _scale;
};

int main()
{
    vector<int> values;
    values.push_back(1);
    values.push_back(2);
    values.push_back(3);
    values.push_back(4);

    // Create a Scale object that scales elements by 3 and apply
    // it to the vector object. Does not modify the vector.
    Scale s(3);
    s.ApplyScale(values);
}
```

## 輸出

```
3
6
9
12
```

## 備註

`ApplyScale` 函式使用 Lambda 運算式列印小數位數值和 `vector` 物件中每個元素的乘積。Lambda 運算式會隱含地捕捉 `this` 讓它可以存取 `_scale` 成員。

[[在本文中](#)]

# 搭配範本使用 Lambda 運算式

## 範例

因為 Lambda 運算式具有類型，因此您可以搭配 C++ 範本使用。下列範例顯示 `negate_all` 和 `print_all` 函式。函式會 `negate_all` 將一元套用 `operator-` 至物件中的每個元素 `vector`。`print_all` 函式會將 `vector` 物件中的每個元素印出至主控台。

## 程式碼

```
// 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);
}
```

## 輸出

```
34
-43
56
After negate_all():
-34
43
-56
```

## 備註

如需 C++ 範本的詳細資訊，請參閱 [範本](#)。

[\[在本文中\]](#)

## 處理例外狀況

## 範例

Lambda 運算式的主體遵循結構化例外狀況處理(SEH)和 C++ 例外狀況處理這兩種規則。您可以處理在 Lambda 運算式主體中引發的例外狀況，也可以延後至封閉範圍再處理例外狀況。下列範例會使用函式 `for_each` 和 lambda 運算式來填入 `vector` 具有另一個值的物件。它會使用 `try / catch` 區塊來處理第一個向量的無效存取。

## 程式碼

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

## 輸出

```
Caught 'invalid vector<T> subscript'.
```

## 備註

如需例外狀況處理的詳細資訊，請參閱[例外狀況處理](#)。

[\[在本文中\]](#)

## 使用 Lambda 運算式搭配 Managed 類型(c + +/CLI)

## 範例

Lambda 運算式的擷取子句不能包含屬於 Managed 類型的變數。不過，您可以將屬於 Managed 類型的引數傳遞至 Lambda 運算式的參數清單。下列範例包含以傳值方式擷取區域 Unmanaged 變數 `ch` 並以 `System.String` 物件做為其參數的 Lambda 運算式：

## 程式碼

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

## 輸出

```
Hello!
```

## 備註

您也可以使用 Lambda 運算式搭配 STL/CLR 程式庫。如需詳細資訊，請參閱[STL/CLR 程式庫參考](#)。

### IMPORTANT

在這些 common language runtime (CLR)受管理的實體中不支援 lambda: `ref class`、`ref struct`、`value class` 和 `value struct`。

[[在本文中](#)]

## 另請參閱

[Lambda 運算式](#)

[Lambda 運算式語法](#)

`auto`

`function` 課堂

`find_if`

`<algorithm>`

[函式呼叫](#)

[範本](#)

[例外狀況處理](#)

[STL/CLR 程式庫參考](#)

# C++ 中的 constexpr lambda 運算式

2020/11/2 • [Edit Online](#)

Visual Studio 2017 15.3 版和更新版本 (適用於 [/std: c++17](#)) : lambda 運算式可以在常數運算式中被宣告為 `constexpr` 或在常數運算式中使用的每個資料成員的初始化時，將其宣告為或用於常數運算式中。

```
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 的結果符合函式的需求，則會隱含地 `constexpr` 執行此動作 `constexpr` :

```
auto answer = [](int n)
{
    return 32 + n;
};

constexpr int response = answer(10);
```

如果 lambda 是隱含或明確的 `constexpr`，而且您將它轉換為函式指標，則產生的函數也會是 `constexpr` :

```
auto Increment = [](int n)
{
    return n + 1;
};

constexpr int(*inc)(int) = Increment;
```

## 另請參閱

[C++ 語言參考](#)

[C++ 標準程式庫中的函式物件](#)

[函式呼叫](#)

[for\\_each](#)

# 陣列 (C++)

2020/12/10 • [Edit Online](#)

陣列是相同類型的物件序列，其佔用連續的記憶體區域。傳統的 C 樣式陣列是許多 bug 的來源，但仍然很常見，尤其是在較舊的程式碼基底中。在新式 C++ 中，我們強烈建議使用 `std::vector` 或 `std::array`，而不是本節所述的 C 樣式陣列。這兩個標準程式庫類型都會將其元素儲存為連續的記憶體區塊。不過，它們提供更高的型別安全，以及保證指向序列內有效位置的支援反覆運算器。如需詳細資訊，請參閱 [容器](#)。

## 堆疊宣告

在 C++ 陣列宣告中，陣列大小是指定在變數名稱之後，而不是在類型名稱之後，如同某些其他語言。下列範例會宣告 1000 的陣列，這是在堆疊上配置的雙精度浮點數。元素數目必須以整數常值的形式提供，或做為常數運算式。這是因為編譯器必須知道要配置多少堆疊空間；它無法使用在執行時間計算的值。陣列中的每個元素都會被指派預設值 0。如果您沒有指派預設值，每個專案一開始都會包含在該記憶體位置發生的任何隨機值。

```
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] << " ";
```

陣列中的第一個元素是第零個元素。最後一個元素是 ( $n-1$ ) 元素，其中  $n$  是陣列可包含的元素數目。宣告中的元素數目必須是整數類資料類型，而且必須大於 0。您必須負責確保程式永遠不會將值傳遞給大於的注標運算子 `(size - 1)`。

只有當陣列是或中的最後一個欄位 `struct` `union`，而且已啟用 Microsoft 擴充功能 (`/Za` 或 `/permissive-` 未設為) 時，才會有合法的陣列。

以堆疊為基礎的陣列比堆積型陣列更快配置和存取。不過，堆疊空間有限。陣列元素的數目不能太大，以致于使用太多的堆疊記憶體。根據您的程式而定，有多少太多。您可以使用程式碼剖析工具判斷陣列是否太大。

## 堆積宣告

您可能需要的陣列太大而無法在堆疊上配置，或其大小在編譯時期不知道。您可以使用運算式，在堆積上配置此陣列 `new[]`。運算子會傳回第一個元素的指標。注標運算子在指標變數上的運作方式，與在堆疊型陣列上的運作方式相同。您也可以使用 [指標算術](#)，將指標移至陣列中的任何任意元素。您必須負責確保：

- 您永遠會保留原始指標位址的複本，以便您可以在不再需要陣列時刪除記憶體。
- 您不會遞增或遞減超出陣列界限的指標位址。

下列範例顯示如何在執行時間定義堆積上的陣列。它會顯示如何使用注標運算子和指標算術來存取陣列元素：

```
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);
}
```

## 初始化陣列

您可以在迴圈中初始化陣列、一次一個元素，或在單一語句中初始化。下列兩個數組的內容相同：

```

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 };

```

## 將陣列傳遞至函式

當陣列傳遞至函式時，它會以指標的形式傳遞至第一個專案，不論它是堆疊型或堆積型的陣列。指標不包含其他大小或類型資訊。這種行為稱為 **指標衰減**。當您將陣列傳遞至函式時，您必須一律在個別參數中指定專案數。此行為也表示當陣列傳遞至函數時，不會複製陣列元素。若要防止函式修改元素，請將參數指定為元素的指標 `const`。

下列範例顯示接受陣列和長度的函式。指標指向原始陣列，而不是複本。因為參數不是，所以函式 `const` 可以修改陣列元素。

```

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]
    }
}

```

宣告並定義陣列參數，`p` `const` 使其成為函式區塊內的唯讀：

```
void process(const double *p, const size_t len);
```

相同的函式也可以用這些方式宣告，而不會有任何行為變更。陣列仍會以指標的形式傳遞至第一個元素：

```

// 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);

```

## 多維陣列

從其他陣列建構的陣列是多維陣列。這些多維陣列是藉由依序放置多個包含括號的常數運算式所指定。例如，以下列宣告為例：

```
int i2[5][7];
```

它會指定類型的陣列 `int`，並在概念上排列于五個數據列的二維矩陣和七個數據行，如下圖所示：

0, 0	0, 1	0, 2	0, 3	0, 4	0, 5	0, 6
1, 0	1, 1	1, 2	1, 3	1, 4	1, 5	1, 6
2, 0	2, 1	2, 2	2, 3	2, 4	2, 5	2, 6
3, 0	3, 1	3, 2	3, 3	3, 4	3, 5	3, 6
4, 0	4, 1	4, 2	4, 3	4, 4	4, 5	4, 6

多維陣列的概念性配置

您可以宣告具有初始化運算式清單（的多維陣列，如 [初始化運算式](#)）中所述。在這些宣告中，指定第一個維度之界限

的常數運算式可以省略。例如：

```
// 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 }
};
```

上述宣告會定義一個三列四行的陣列。資料列代表工廠，資料行則代表工廠出貨的市場。值是從工廠到市場的運費。陣列的第一個維度會被省略，但編譯器會藉由檢查初始設定式填入該維度。

在  $n$  維陣列類型上使用間接運算子 (\*) 會產生  $n-1$  維度陣列。如果  $n$  是 1，則會產生純量 (或陣列元素)。

C++ 陣列是以列為主要順序進行儲存。以列為主要順序表示最後一個註標變更最快速。

## 範例

您也可以在函式宣告中省略多維陣列第一個維度的界限規格，如下所示：

```

// multidimensional_arrays.cpp
// compile with: /EHsc
// arguments: 3
#include <limits>    // Includes DBL_MAX
#include <iostream>

const int cMkts = 4, cFacts = 2;

// Declare a float that represents the transportation costs
double TransportCosts[][][cMkts] = {
    { 32.19, 47.29, 31.99, 19.11 },
    { 11.29, 22.49, 33.47, 17.29 },
    { 41.97, 22.09, 9.76, 22.55 }
};

// Calculate size of unspecified dimension
const int cFactories = sizeof TransportCosts /
    sizeof( double[cMkts] );

double FindMinToMkt( int Mkt, double myTransportCosts[][cMkts], int mycFacts);

using namespace std;

int main( int argc, char *argv[] ) {
    double MinCost;

    if ( argv[1] == 0 ) {
        cout << "You must specify the number of markets." << endl;
        exit(0);
    }
    MinCost = FindMinToMkt( *argv[1] - '0', TransportCosts, cFacts );
    cout << "The minimum cost to Market " << argv[1] << " is: "
        << MinCost << "\n";
}

double FindMinToMkt(int Mkt, double myTransportCosts[][cMkts], int mycFacts) {
    double MinCost = DBL_MAX;

    for( size_t i = 0; i < cFacts; ++i )
        MinCost = (MinCost < TransportCosts[i][Mkt]) ?
            MinCost : TransportCosts[i][Mkt];

    return MinCost;
}

```

The minimum cost to Market 3 is: 17.29

撰寫函數 `FindMinToMkt` 是為了讓新的處理站不需要變更任何程式碼，只要重新編譯就可以了。

## 初始化陣列

具有類別的函式的物件陣列會由函式初始化。當初始化運算式清單中的專案數少於陣列中的元素時，預設的函式會用於其餘的元素。如果未針對類別定義預設的函式，則初始化運算式清單必須完整，也就是陣列中的每個元素都必須有一個初始化運算式。

以定義兩個建構函式的 `Point` 類別為例：

```

// 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` 的第一個元素是使用 `Point( int, int )` 建構函式建構，其餘兩個元素則是使用預設建構函式建構。

靜態成員陣列 (是否 `const` 可以在類別宣告) 以外的 (定義中初始化)。例如：

```

// 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()
{
}

```

## 存取陣列元素

使用陣列註標運算子 (`[ ]`)，您可以存取陣列的個別項目。如果您使用一維陣列的名稱，但沒有注標，則會評估為數組第一個元素的指標。

```

// 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.
}

```

當您使用多維陣列時，您可以在運算式中使用各種組合。

```

// 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.
}

```

在上述程式碼中，`multi` 是類型的三維陣列 `double`。`p2multi` 指標指向大小為三的陣列類型 `double`。在此範例中，陣列搭配使用的註標有一個、兩個和三個。雖然指定所有註標(如同在語句中一樣常見)，但 `cout` 有時候選取特定的陣列元素子集會很有用，如下列語句所示 `cout`。

## 多載注標運算子

和其他運算子一樣，可由使用者重新定義 () 的注標運算子 `[]`。註標運算子的預設行為(如果未多載)是使用下列方法結合陣列名稱和註標：

```
*((array_name) + (subscript))
```

如同所有包含指標類型的加法，會自動進行調整以調整為類型的大小。結果值不是來自來源的  $n$  個位元組 `array_name`，而是陣列的第  $n$  個元素。如需這種轉換的詳細資訊，請參閱 [加法類運算子](#)。

同樣地，對於多維陣列而言，位址是使用下列方法衍生：

```
((array_name) + (subscript1 * max2 * max3 * ... * maxn) + (subscript2 * max3 * ... * maxn) + ... + subscriptn))
```

## 運算式中的陣列

當陣列類型的識別碼出現在以外的運算式中時 `sizeof`，位址 (`&`) 或參考的初始化，會轉換成第一個陣列元素的指標。例如：

```

char szError1[] = "Error: Disk drive not ready.";
char *psz = szError1;

```

指標 `psz` 會指向陣列 `szError1` 的第一個元素。與指標不同的陣列不是可修改的左值。這就是為什麼下列指派不合法的原因：

```
szError1 = psz;
```

## 另請參閱

[std::array](#)

# 參考 (C++)

2020/11/2 • [Edit Online](#)

參考 (例如指標) 會儲存位在記憶體中其他位置之物件的位址。與指標不同，參考在初始化之後不能參照不同的物件，或設為 null。有兩種類型的參考：左值參考，參考名為的變數，以及參考暫存物件的右值參考。& 運算子表示左值參考，而 && 運算子表示右值參考，或是根據內容而定的通用參考 (右值或左值)。

參考可能是使用下列語法來宣告：

```
[儲存類別規範][ cv 限定詞] 型別規範 [ ms-修飾詞 declarator [= 運算式];
```

可能會使用任何有效的宣告子來指定參考。除非參考是函式或陣列類型的參考，否則便會套用下列簡化的語法：

```
[儲存類別規範][ cv 限定詞] 類型規範 [& 或 [&& [ cv 限定詞] 識別碼 [= 運算式];
```

參考是使用下列序列宣告：

1. 宣告指定名稱：

- 選擇性的儲存類別規範。
- 選擇性 `const` 和(或) `volatile` 限定詞。
- 類型指定名稱：類型的名稱。

2. 宣告子：

- 選擇性的 Microsoft 專有修飾詞。如需詳細資訊，請參閱[Microsoft 專有的修飾詞](#)。
- & 運算子或 && 運算子。
- 選擇性 `const` 和/或 `volatile` qualifiers。
- 識別碼。

3. 選擇性的初始設定式。

陣列和函式指標的更複雜宣告子形式也適用於陣列和函式的參考。如需詳細資訊，請參閱[指標](#)。

多個宣告子和初始設定式可能會出現在逗號分隔清單中，後面接著一個宣告指定名稱。例如：

```
int &i;  
int &i, &j;
```

參考、指標和物件可以同時宣告：

```
int &ref, *ptr, k;
```

參考會保存物件的位址，不過其運作方式與物件相同。

在下列程式中，請注意物件的名稱 `s` 和物件的參考 `SRef` 在程式中的使用方式是相同的：

## 範例

```
// references.cpp
#include <stdio.h>
struct S {
    short i;
};

int main() {
    S s;      // Declare the object.
    S& SRef = s;  // Declare the reference.
    s.i = 3;

    printf_s("%d\n", s.i);
    printf_s("%d\n", SRef.i);

    SRef.i = 4;
    printf_s("%d\n", s.i);
    printf_s("%d\n", SRef.i);
}
```

```
3
3
4
4
```

## 另請參閱

[參考型別函式引數](#)

[參考型別函式傳回](#)

[指標的參考](#)

# 左值參考宣告子 : &

2020/11/2 • [Edit Online](#)

保存物件的位址，但語法上的行為與物件相似。

## 語法

```
type-id & cast-expression
```

## 備註

您可以將左值參考當做物件的另一個名稱。左值參考宣告包含選擇性的指定名稱清單加上參考宣告符號。參考必須經過初始化，而且不能變更。

只要物件的位址可以轉換為指定的指標類型，則該物件也可轉換為類似的參考類型。例如，只要物件的位址可以轉換成 `char *` 類型，該物件也可轉換成 `char &` 類型。

請勿將參考宣告與使用[位址運算子](#)混淆。當 `&` 識別碼前面加上類型(例如 `int` 或 `char`)時，會將識別碼宣告為類型的參考。當 `& identifier`前面沒有一個型別時，其使用方式就是傳址運算子的用法。

## 範例

下列範例示範宣告 `Person` 物件和該物件之參考的參考宣告子。因為 `rFriend` 是 `myFriend` 的參考，無論更新哪一個變數都會變更同一個物件。

```
// reference_declarator.cpp
// compile with: /EHsc
// Demonstrates the reference declarator.
#include <iostream>
using namespace std;

struct Person
{
    char* Name;
    short Age;
};

int main()
{
    // Declare a Person object.
    Person myFriend;

    // Declare a reference to the Person object.
    Person& rFriend = myFriend;

    // Set the fields of the Person object.
    // Updating either variable changes the same object.
    myFriend.Name = "Bill";
    rFriend.Age = 40;

    // Print the fields of the Person object to the console.
    cout << rFriend.Name << " is " << myFriend.Age << endl;
}
```

Bill is 40

## 另請參閱

[參考](#)

[參考型別函式引數](#)

[參考型別函式傳回](#)

[指標的參考](#)

# 右值參考宣告子 : &&

2020/11/2 • [Edit Online](#)

保留右值運算式的參考。

## 語法

```
type-id && cast-expression
```

## 備註

右值參考可讓您區分左值與右值。左值參考和右值參考的語法和語意很相似，但是兩者遵循的規則稍有不同。如需左值和右值的詳細資訊，請參閱 [左值和右值](#)。如需左值參考的詳細資訊，請參閱左值參考宣告子 : &。

下列各節說明右值參考如何支援 [移動語義](#) 和 [完美轉送](#) 的實值。

## 移動語意

右值參考支援 [移動語義](#) 的執行，這可能會大幅增加應用程式的效能。移動語意可讓您撰寫將資源從一個物件轉移到另一個物件 (例如動態配置記憶體) 的程式碼。移動語意可用於轉移暫存物件中無法供程式其他位置參考的資源，因此可以發揮功效。

若要執行移動語法，您通常會提供移動的函式，並選擇性地將移動指派運算子 (operator =) 提供給您的類別。然後，來源為右值的複製和指派作業會自動利用移動語意。不同於預設複製建構函式，編譯器不提供預設移動建構函式。如需如何撰寫移動函式以及如何在應用程式中使用它的詳細資訊，請參閱 [移動函式](#) 和 [移動指派運算子 \(C++\)](#)。

您也可以多載一般函式和運算子，以使用移動語意。Visual Studio 2010 引進將語義轉換成 C++ 標準程式庫。例如，`string` 類別會實作執行移動語意的作業。請考慮下列串連數個字串並列印結果的範例：

```
// 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，運算子 + 的每個呼叫都會配置並傳回新的暫存 `string` 物件 (右值)。運算子 + 無法將一個字串附加至另一個字串，因為它不知道來源字串是否為左值或右值。如果原始字串都是左值，程式中其他位置可能會參考它們，因此不得加以修改。藉由使用右值參考，可以修改運算子 + 以採用右值，而不會在程式中的其他位置參考。因此，operator + 現在可以將一個字串附加至另一個字串。這可以大幅減少 `string` 類別必須執行的動態記憶體配置數目。如需類別的詳細資訊 `string`，請參閱 [basic\\_string 類別](#)。

當編譯器無法使用傳回值最佳化 (RVO) 或具名傳回值最佳化 (NRVO) 時，移動語意也能有所幫助。在這些情況下，如果類型定義了移動建構函式，則編譯器會加以呼叫。

若要進一步了解移動語意，請參考將項目插入至 `vector` 物件的範例。如果超出 `vector` 物件的容量，`vector` 物

件必須重新配置其項目的記憶體，然後將每個項目複製到另一個記憶體位置，以便為插入的項目騰出空間。當插入作業複製項目時，會建立新的項目，呼叫複製建構函式複製上一個項目的資料到新項目，然後終結上一個項目。移動語意可讓您直接移動物件，而不必執行耗費大量資源的記憶體配置和複製作業。

若要利用 `vector` 範例中的移動語意，您可以寫入移動建構函式，將資料從某個物件移動至另一個。

如需在 Visual Studio 2010 的 C++ 標準程式庫中引進 move 語義的詳細資訊，請參閱 [C++ 標準程式庫](#)。

## 完美轉送

完美轉送可減少對多載函式的需求，有助於避免轉送問題。當您撰寫的泛型函式採用參考做為其參數，且傳遞（或轉送）這些參數至另一個函式時，可能會發生轉送問題。例如，如果泛型函式採用 `const T&` 類型的參數，則呼叫的函式無法修改該參數的值。如果泛型函式接受 `T&` 類型的參數，則無法使用右值（例如暫存物件或整數常值）呼叫函式。

若要解決這個問題，您通常必須提供每個參數採用 `T&` 和 `const T&` 的泛型函式多載版本。因此，多載函式數目會隨著參數數目以指數方式增加。右值參考可讓您撰寫接受任意引數並轉送至另一個函式的函式版本，就如同直接呼叫另一個函式。

請考慮宣告 `W`、`X`、`Y` 和 `Z` 這四種類型的下列範例。每個類型的函式會採用 `const` 與非左值參考的不同組合 `const` 做為其參數。

```
struct W
{
    W(int&, int&) {}
};

struct X
{
    X(const int&, int&) {}
};

struct Y
{
    Y(int&, const int&) {}
};

struct Z
{
    Z(const int&, const int&) {}
};
```

假設您要撰寫產生物件的泛型函式。下列範例示範撰寫此函式的其中一種方法：

```
template <typename T, typename A1, typename A2>
T* factory(A1& a1, A2& a2)
{
    return new T(a1, a2);
}
```

下列範例示範對 `factory` 函式的有效呼叫：

```
int a = 4, b = 5;
W* pw = factory<W>(a, b);
```

不過，下列範例未包含對 `factory` 函式的有效呼叫，因為 `factory` 會接受可修改的左值參考做為其參數，但它是使用右值呼叫：

```
Z* pz = factory<Z>(2, 2);
```

若要解決這個問題，您通常必須為每個 `factory` 與 `A&` 參數的組合建立 `const A&` 函式的多載版本。右值參考可讓您撰寫 `factory` 函式的一個版本，如下列範例所示：

```
template <typename T, typename A1, typename A2>
T* factory(A1&& a1, A2&& a2)
{
    return new T(std::forward<A1>(a1), std::forward<A2>(a2));
}
```

這個範例使用右值參考做為 `factory` 函式的參數。`std::forward`函式的目的將 `factory` 函式的參數轉寄給樣板類別的函式。

下列範例示範 `main` 函式，這個函式會使用修改的 `factory` 函式來建立 `W`、`X`、`Y` 和 `Z` 類別的執行個體。修改的 `factory` 函式會將其參數 (左值或右值) 轉送至適當的類別建構函式。

```
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;
}
```

## 右值參考的其他屬性

您可以多載函式以接受左值參考和右值參考。

藉由多載函式來採用 `const` 左值參考或右值參考，您可以撰寫程式碼來區分不可修改的物件 (左值) 和可修改的暫存值 (右值)。您可以將物件傳遞至接受右值參考的函式，除非物件標示為 `const`。下列範例說明多載之後可接受左值參考和右值參考的 `f` 函式。`main` 函式會使用左值和右值呼叫 `f`。

```

// reference-overload.cpp
// Compile with: /EHsc
#include <iostream>
using namespace std;

// A class that contains a memory resource.
class MemoryBlock
{
    // TODO: Add resources for the class here.
};

void f(const MemoryBlock&)
{
    cout << "In f(const MemoryBlock&). This version cannot modify the parameter." << endl;
}

void f(MemoryBlock&&)
{
    cout << "In f(MemoryBlock&&). This version can modify the parameter." << endl;
}

int main()
{
    MemoryBlock block;
    f(block);
    f(MemoryBlock());
}

```

這個範例會產生下列輸出：

```

In f(const MemoryBlock&). This version cannot modify the parameter.
In f(MemoryBlock&&). This version can modify the parameter.

```

在此範例中，第一個對 `f` 的呼叫會傳遞區域變數（左值）做為其引數。對 `f` 的第二個呼叫會傳遞暫存物件做為其引數。由於程式中的其他位置無法參考暫存物件，呼叫會繫結至接受右值參考的 `f` 多載版本，此版本可以自由修改物件。

編譯器會將具名右值參考視為左值，而將未具名右值參考當做右值。

當您撰寫接受右值參考做為其參數的函式時，該參數會被視為函式主體中的左值。由於程式的多個部分都可以參考具名物件，編譯器會將具名右值參考視為左值；因此，允許程式的多個部分從該物件修改或移除資源是相當危險的。例如，如果程式中有許多個部分嘗試從相同物件傳輸資源，則只有第一個部分會成功傳輸資源。

下列範例說明多載之後可接受左值參考和右值參考的 `g` 函式。`f` 函式接受右值參考做為其參數（具名右值參考），並傳回右值參考（未具名右值參考）。從 `g` 對 `f` 的呼叫中，多載解析會選取採用左值參考的 `g` 版本，因為 `f` 主體會將其參數視為左值。從 `g` 對 `main` 的呼叫中，多載解析會選取採用右值參考的 `g` 版本，因為 `f` 會傳回右值參考。

```

// 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()));
}

```

這個範例會產生下列輸出：

```

In g(const MemoryBlock&).
In g(MemoryBlock&&).

```

在這個範例中，`main` 函式將右值傳遞給 `f`。`f` 主體會將其具名參數視為左值。從 `f` 對 `g` 的呼叫會將參數繫結至左值參考 (`g` 的第一個多載版本)。

- 您可以將左值轉換成右值參考。

C++ 標準程式庫 `std::move` 函數可讓您將物件轉換為該物件的右值參考。或者，您可以使用 `static_cast` 關鍵字將左值轉換成右值參考，如下列範例所示：

```

// 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));
}

```

這個範例會產生下列輸出：

```

In g(const MemoryBlock&).
In g(MemoryBlock&&).

```

**函式樣板會推算其樣板引數類型，然後使用參考摺疊規則。**

常見的方式是撰寫函式範本以傳遞 (, 或將) 其參數 轉送至另一個函式。請務必了解樣板類型推算對於接受右值參考之函式樣板的作用。

如果函式引數為右值，編譯器會推算引數為右值參考。例如，如果您將  $x$  類型物件的右值參考傳遞至採用  $T\&\&$  類型做為其參數的樣板函式，則樣板引數推算會將  $T$  推算為  $x$ 。因此，該參數會具有  $x\&\&$  類型。如果函式引數為左值或  $const$  左值，則編譯器會推算其類型為該類型的左值參考或  $const$  左值參考。

下列範例宣告某個結構樣板，然後針對各種參考類型特製化此樣板。`print_type_and_value` 函式接受右值參考做為其參數，並將其轉送給 `S::print` 方法的適當特製化版本。`main` 函式示範各種呼叫 `S::print` 方法的方式。

```

// 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\&\&)." << endl;
    }
}

```

```

        cout << print<T>::print(t);
    }

};

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());
}

```

這個範例會產生下列輸出：

```

print<T&>: first
print<const T &>: second
print<T&&>: third
print<const T&&>: fourth

```

為了解析對 `print_type_and_value` 函式的每個呼叫，編譯器會先執行樣板引數推算。編譯器以推算出的樣板引數取代參數類型時，會接著套用參考摺疊規則。例如，將區域變數 `s1` 傳遞至 `print_type_and_value` 函式會造成編譯器產生下列函式簽章：

```
print_type_and_value<string&>(string& && t)
```

編譯器會使用參考摺疊規則，將簽章縮減如下：

```
print_type_and_value<string&>(string& t)
```

`print_type_and_value` 函式的這個版本接著會將其參數轉送給 `s::print` 方法的正確特製化版本。

下表摘要說明樣板引數類型推算的參考摺疊規則：

左值參考	右值參考
<code>T&amp; &amp;</code>	<code>T&amp;</code>
<code>T&amp; &amp;&amp;</code>	<code>T&amp;</code>
<code>T&amp;&amp; &amp;</code>	<code>T&amp;</code>
<code>T&amp;&amp; &amp;&amp;</code>	<code>T&amp;&amp;</code>

樣板引數推算是實作完美轉送的要素。本主題先前的〈完美轉送〉一節更詳細說明完美轉送。

## 摘要

右值參考會區分左值與右值。它們可以消除不必要的記憶體配置和複製作業，因此能協助您改善應用程式效能。此外，也可以讓您撰寫接受任意引數並轉送至另一個函式的函式版本，就如同直接呼叫另一個函式。

## 另請參閱

[具有一元運算子的運算式](#)

[左值參考宣告子:&](#)

[左值和右值](#)

[移動建構函式和移動指派運算子 \(C++\)](#)

[C++ 標準程式庫](#)

# 參考類型函式引數

2020/11/2 • [Edit Online](#)

通常更有效率的方式是傳遞參考，而不是傳遞大型物件給函式。這可讓編譯器傳遞物件位址，同時又可維護將用來存取物件的語法。請考慮下列會使用 `Date` 結構的範例。

```
// 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;
}
```

上述程式碼顯示使用成員選取運算子`(.)`而非指標成員選取運算子`(*)`存取以傳址方式傳遞之結構的成員`->`。

雖然當做參考型別傳遞的引數會觀察到非指標類型的語法，但它們會保留指標類型的一個重要特性：除非宣告為`const`，否則它們是可修改的`const`。由於上述程式碼的目的不是要修改物件`date`，因此更適當的函式原型為：

```
long DateOfYear( const Date& date );
```

此原型可確保 `DateOfYear` 函式不會變更其引數。

任何採用參考型別的函式原型都可以接受其位置中相同類型的物件，因為從 *typename* 到 *typename* 的標準轉換 & 。

## 另請參閱

[參考](#)

# 參考類型函式傳回

2020/11/2 • [Edit Online](#)

函式可宣告為傳回參考類型。進行此類宣告有兩個原因：

- 傳回的資訊是一個夠大的物件，因此傳回參考比傳回複本更有效率。
- 函式的類型必須是左值。
- 傳回函式時，referred-to 物件不會超出範圍。

就像是以傳址方式將大型物件傳遞至函式一樣有效率，從函式依參考傳回大型物件也會更有效率。使用參考傳回通訊協定就不需要在傳回之前將物件複製到暫存位置。

當函式必須評估為左值時，參考傳回型別可能也會很有用。大部分的多載運算子都屬於此類，特別是指派運算子。多載的運算子涵蓋在多載的運算子中。

## 範例

請考量 **Point** 範例：

```

// 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";
}

```

## 輸出

```

x = 7
y = 9

```

請注意，函式 `x` 和 `y` 已宣告為傳回參考類型。這些函式可在指派陳述式的任一邊使用。

也請注意，在 `main` 中，`Point` 物件保持在範圍內，因此其參考成員仍然存在且可以安全地存取。

除了下列情況下之外，參考類型的宣告必須包含初始設定式：

- 明確 `extern` 宣告
- 類別成員的宣告
- 類別中的宣告
- 函式引數或函式傳回型別的宣告

## 傳回區域變數位址注意事項

如果您宣告區域範圍的物件，則會在傳回函式時終結該物件。如果函式傳回該物件的參考，則呼叫端嘗試使用 null

參考時，該參考可能會在執行階段造成存取違規。

```
// C4172 means Don't do this!!!
Foo& GetFoo()
{
    Foo f;
    ...
    return f;
} // f is destroyed here
```

在此情況下，編譯器會發出警告：`warning C4172: returning address of local variable or temporary`。在簡單程式中，如果呼叫端在覆寫記憶體位置之前存取參考，則可能偶爾不會發生存取違規。這純粹只是幸運。請留意警告。

## 另請參閱

[參考](#)

# 指標的參考

2019/12/2 • [Edit Online](#)

指標的參考可以透過與物件的參考類似的方式來宣告。指標的參考是可修改的值，可像一般的指標。

## 範例

此程式碼範例顯示使用指標的指標以及指標的參考之間的差異。

函式 `Add1` 和 `Add2` 的功能相同，雖然它們不呼叫相同的方式。其差異在於 `Add1` 使用 `double` 的間接取值，但 `Add2` 指標參考的便利性。

```
// 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 BTTree {
    char *szText;
    BTTree *Left;
    BTTree *Right;
};

// Define a pointer to the root of the tree.
BTTree *btRoot = 0;

int Add1( BTTree **Root, char *szToAdd );
int Add2( BTTree*& Root, char *szToAdd );
void PrintTree( BTTree* 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' );
        Add1( &btRoot, szBuf );
        Add2( btRoot, szBuf );
    }
}
```

```

    cin.get();

    if ( strlen( szBuf ) ) {
        switch ( *argv[1] ) {
            // Method 1: Use double indirection.
            case '1':
                Add1( &btRoot, szBuf );
                break;
            // Method 2: Use reference to a pointer.
            case '2':
                Add2( btRoot, szBuf );
                break;
            default:
                cerr << "Illegal value "
                    << *argv[1]
                    << "' supplied for add method.\n"
                    << "Choose 1 or 2.\n";
                return -1;
        }
    }
}

// Display the sorted list.
PrintTree( btRoot );
}

// PrintTree: Display the binary tree in order.
void PrintTree( BTREE* MybtRoot ) {
    // Traverse the left branch of the tree recursively.
    if ( MybtRoot->Left )
        PrintTree( MybtRoot->Left );

    // Print the current node.
    cout << MybtRoot->szText << "\n";

    // Traverse the right branch of the tree recursively.
    if ( MybtRoot->Right )
        PrintTree( MybtRoot->Right );
}

// Add1: Add a node to the binary tree.
//      Uses double indirection.
int Add1( BTREE **Root, char *szToAdd ) {
    if ( (*Root) == 0 ) {
        (*Root) = new BTREE;
        (*Root)->Left = 0;
        (*Root)->Right = 0;
        (*Root)->szText = new char[strlen( szToAdd ) + 1];
        strcpy_s((*Root)->szText, (strlen( szToAdd ) + 1), szToAdd );
        return 1;
    }
    else {
        if ( strcmp( (*Root)->szText, szToAdd ) > 0 )
            return Add1( &((*Root)->Left), szToAdd );
        else
            return Add1( &((*Root)->Right), szToAdd );
    }
}

// Add2: Add a node to the binary tree.
//      Uses reference to pointer
int Add2( BTREE*& Root, char *szToAdd ) {
    if ( Root == 0 ) {
        Root = new BTREE;
        Root->Left = 0;
        Root->Right = 0;
        Root->szText = new char[strlen( szToAdd ) + 1];
        strcpy_s( Root->szText, (strlen( szToAdd ) + 1), szToAdd );
        return 1;
    }
}

```

```
        }
    else {
        if ( strcmp( Root->szText, szToAdd ) > 0 )
            return Add2( Root->Left, szToAdd );
        else
            return Add2( Root->Right, szToAdd );
    }
}
```

```
Usage: references_to_pointers.exe [1 | 2]
```

where:

1 uses double indirection

2 uses a reference to a pointer.

```
Input is from stdin. Use ^Z to terminate input.
```

## 另請參閱

### 參考

# 指標 (C++)

2020/4/15 • [Edit Online](#)

指標是存儲物件的記憶體位址的變數。指標在 C 和 C++ 中廣泛使用，主要有三個目的：

- 在堆上分配新物件，
- 將函數傳遞給其他函式
- 反覆運算陣列或其他數據結構中的元素。

在 C 樣式的編程中，原始指標用於所有這些方案。但是，原始指標是許多嚴重程式設計錯誤的根源。因此，強烈建議不使用它們，除非它們提供了顯著的性能優勢，並且對於哪個指標是負責刪除對象的擁有指標沒有歧義。現代 C++ 提供用於分配物件的智慧指標、用於遍歷資料結構的迭代器以及用於傳遞函數的 lambda 運算式。通過使用這些語言和庫工具而不是原始指標，可以使程式更安全、更易於調試以及更易於理解和維護。有關詳細資訊，請參閱 [智慧指標](#)、[反覆運算器](#) 和 [Lambda 運算式](#)。

## 本節內容

- [原始指標](#)
- [Const 和易失性指標](#)
- [新增運算子與刪除運算子](#)
- [智慧指標](#)
- [如何建立和使用 unique\\_ptr 實體](#)
- [如何建立及使用 shared\\_ptr 實體](#)
- [如何建立與使用 weak\\_ptr 實體](#)
- [如何建立與使用 CComPtr 與 CComQIPtr 實體](#)

## 另請參閱

[迭代器](#)

[Lambda 運算式](#)

# 原始指標 ( C + + )

2020/11/2 • [Edit Online](#)

指標是一種變數類型。它會將物件的位址儲存在記憶體中，並用來存取該物件。原始指標是一種指標，其存留期不是由封裝物件所控制，例如智慧型指標。原始指標可以被指派另一個非指標變數的位址，或者可以指派的值 `nullptr`。尚未指派值的指標包含亂數據。

指標也可以被取值，以抓取它所指向之物件的值。成員存取運算子可讓您存取物件的成員。

```
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
```

指標可以指向具類型的物件或 `void`。當程式在記憶體中的堆積上設定物件時，它會以指標的形式接收該物件的位址。這類指標稱為「擁有指標」。當不再需要所需的堆積設定物件時，必須使用擁有指標(或其複本)來明確釋放該物件。無法釋放記憶體會導致記憶體流失，並將該記憶體位置轉譯為電腦上的任何其他程式無法使用。使用配置 `new` 的記憶體必須使用 `delete` (或`** delete []**`)釋放。如需詳細資訊，請參閱 [new 和 delete 運算子](#)。

```
MyClass* mc = new MyClass(); // allocate object on the heap
mc->print(); // access class member
delete mc; // delete object (please don't forget!)
```

指標(如果未宣告為 `const`)可以遞增或遞減，以指向記憶體中的另一個位置。這種運算稱為指標算術。它用於 C 樣式的程式設計中，以反復查看陣列或其他資料結構中的元素。`const` 指標無法指向不同的記憶體位置，而且在這個意義上類似于 [參考](#)。如需詳細資訊，請參閱 [const 和 volatile 指標](#)。

```
// 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位的作業系統上，指標的大小為64位。系統的指標大小會決定它可以有多少可定址的記憶體。指標的所有複本都會指向相同的記憶體位置。指標(連同參考)廣泛用於 C + +，以便在函式之間傳遞更大的物件。這是因為複製物件的位址通常比複製整個物件更有效率。定義函式時，請將指標參數指定為，`const` 除非您想要函式來修改物件。一般而言，`const` 參考是將物件傳遞給函式的慣用方式，除非物件的值可能是 `nullptr`。

函式的指標可讓函數傳遞至其他函數，並用於 C 樣式程式設計中的「回呼」。基於此目的，新式 C + + 會使用 [lambda 運算式](#)。

## 初始化和成員存取

下列範例顯示如何宣告、初始化和使用原始指標。它會使用 `new` 進行初始化，以指向在堆積上配置的物件，您必須明確地進行此操作 `delete`。此範例也會顯示與原始指標相關聯的幾個危險。(請記住，此範例是 C 樣式程式設計，而非現代 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
}
```

## 指標算術和陣列

指標與陣列密切相關。當陣列以傳值方式傳遞至函式時，會將它當做第一個元素的指標傳遞。下列範例示範指標和陣列的下列重要屬性：

- 運算子會傳回 `sizeof` 陣列的總大小(以位元組為單位)
- 若要判斷元素的數目，請將總位元組除以一個元素的大小
- 將陣列傳遞至函式時，會 *decays* 至指標類型
- 套用 `sizeof` 至指標時的運算子會傳回指標大小、x86 上的4個位元組或 x64 上的8個位元組

```
#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);
}
```

某些算數運算可以用於非 `const` 指標，使其指向另一個記憶體位置。指標會使用 `++`、`+ = - =` 和運算子來遞增和遞減 `--`。這項技術可用於陣列中，特別適用於不具類型的資料緩衝區。會以 `void* char` (1個位元組)的大小遞增。具類型的指標會以其指向的類型大小遞增。

下列範例會示範如何使用指標算術來存取 Windows 點陣圖中的個別圖元。請注意 `and` 和取值運算子的用法 `new` `delete`。

```

#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\* 指標

**void** 僅指向原始記憶體位置的指標。有時必須使用 **void\*** 指標，例如，在 C++ 程式碼和 C 函式之間傳遞時。

當具類型的指標轉換成 void 指標時，記憶體位置的內容不會變更。不過，類型資訊會遺失，因此您無法執行遞增或遞減作業。記憶體位置可以轉型，例如從轉換成，`MyClass*` `void*` 再轉換回 `MyClass*`。這類作業原本就很容易出錯，而且需要特別注意 void 錯誤。新式 c + + void 在幾乎所有情況下都不鼓勵使用指標。

```
//func.c
void func(void* data, int length)
{
    char* c = (char*)(data);

    // fill in the buffer with data
    for (int i = 0; i < length; ++i)
    {
        *c = 0x41;
        ++c;
    }
}

// main.cpp
#include <iostream>

extern "C"
{
    void func(void* data, int length);
}

class MyClass
{
public:
    int num;
    std::string name;
    void print() { std::cout << name << ":" << num << std::endl; }
};

int main()
{
    MyClass* mc = new MyClass{10, "Marian"};
    void* p = static_cast<void*>(mc);
    MyClass* mc2 = static_cast<MyClass*>(p);
    std::cout << mc2->name << std::endl; // "Marian"

    // use operator new to allocate untyped memory block
    void* pvoid = operator new(1000);
    char* pchar = static_cast<char*>(pvoid);
    for(char* c = pchar; c < pchar + 1000; ++c)
    {
        *c = 0x00;
    }
    func(pvoid, 1000);
    char ch = static_cast<char*>(pvoid)[0];
    std::cout << ch << std::endl; // 'A'
    operator delete(p);
}
```

## 函式的指標

在 C 樣式程式設計中，函式指標主要是用來將函式傳遞給其他函數。這項技術可讓呼叫者自訂函式的行為，而不需要修改函式。在現代 c + + 中，[Lambda 運算式](#)提供的功能具有更高的型別安全和其他優點。

函式指標宣告會指定所指向函數必須具有的簽章：

```
// 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);
```

下列範例會顯示一個函式 `combine`，該函式會採用接受的任何函式做為參數 `std::string`，並傳回 `std::string`。根據傳遞至的函式 `combine`，它會在字串前面加上或附加。

```
#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";
}
```

## 另請參閱

[智慧型指標 間接取值運算子: \\*](#)

[傳址運算子:&](#)

[歡迎回到 C++](#)

# const 和 volatile 指標

2020/11/2 • [Edit Online](#)

Const和volatile關鍵字會變更指標的處理方式。`const` 關鍵字指定在初始化之後無法修改指標; 之後, 指標會受到保護而無法修改。

`volatile` 關鍵字指定與後面的名稱相關聯的值可以由使用者應用程式以外的動作來修改。因此, `volatile` 關鍵字適用於宣告共用記憶體中的物件, 可供多個進程或用於與插斷服務常式通訊的全域資料區域存取。

當名稱宣告為時 `volatile`, 編譯器會在每次由程式存取時, 從記憶體重載值。這將可大幅減少進行最佳化的次數。不過, 當物件的狀態可能遭到意外變更時, 它仍是確保可預測程式效能的唯一方式。

若要將指標所指向的物件宣告為 `const` 或 `volatile`, 請使用下列格式的宣告:

```
const char *cpch;
volatile char *vpch;
```

若要宣告指標的值(也就是儲存在指標中的實際位址), `const` `volatile` 請使用下列格式的宣告:

```
char * const pchc;
char * volatile pchv;
```

C++ 語言會防止允許修改宣告為之物件或指標的指派 `const`。這類指派會移除用來宣告物件或指標的資訊, 因此違反了原始宣告的用意。請考慮下列宣告:

```
const char cch = 'A';
char ch = 'B';
```

假設有兩個物件的前面宣告(`cch`, 類型為`const char`, 而 `ch`, 類型為`char`), 下列宣告/初始化是有效的:

```
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;
```

下列宣告/初始化是錯誤的。

```
char *pch2 = &cch; // Error
char *const pch3 = &cch; // Error
```

`pch2` 的宣告會宣告一項指標, 但是常數物件可能會透過該指標而遭到修改, 因此不允許此宣告。的宣告 `pch3` 會指定指標為常數, 而不是物件; 因為不允許宣告, 所以不允許宣告 `pch2`。

下列八個指派顯示透過指標進行的指派, 以及變更上述宣告的指標值; 現在, 我們可以假設透過 `pch1` 進行 `pch8` 的初始化是正確的。

```
*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
```

宣告為 `volatile` 或混合使用的指標會 `const volatile` 遵守相同的規則。

`const` 物件的指標通常用於函式宣告，如下所示：

```
errno_t strcpy_s( char *strDestination, size_t numberOfElements, const char *strSource );
```

上述語句會宣告函式，`strcpy_s`，其中三個引數的其中兩個是的類型指標 `char`。由於引數是以傳址方式傳遞，而不是以傳值方式傳遞，因此，如果未宣告為，則函式可自由修改 `strDestination` 和 `strSource`。`strSource` 的宣告 `const` 會確保呼叫端 `strSource` 無法由被呼叫的函式變更。

#### NOTE

因為有從 `typename` 到 `typename` 的標準轉換，所以將 `* const typename*` 類型的引數傳遞 `char *` 給 `strcpy_s` 是合法的。不過，反向並不成立；從物件或指標移除屬性時，不會有隱含轉換 `const`。

`const` 指定類型的指標可以指派給相同類型的指標。不過，不 `const` 能將指標指派給 `const` 指標。下列程式碼顯示正確和不正確的指派：

```
// const_pointer.cpp
int *const cpObject = 0;
int *pObject;

int main() {
    pObject = cpObject;
    cpObject = pObject; // C3892
}
```

下列範例顯示在您有一個指標指向物件的指標時，如何將物件宣告為常數。

```
// 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;
}
```

另請參閱

[指標 原始指標](#)

# `new` 和 `delete` 運算子

2020/11/2 • [Edit Online](#)

C++ 支援使用和運算子來動態配置和解除配置物件 `new` `delete`。這些運算子會從稱為可用儲存區的集區配置物件的記憶體。`new` 運算子會呼叫特殊函式 `operator new`，而 `delete` 運算子會呼叫特殊函數 `operator delete`。

`new` C++ 標準程式庫中的函式支援 C++ 標準中指定的行為，`std::bad_alloc` 如果記憶體配置失敗，則會擲回例外狀況。如果您仍然想要的非擲回版本 `new`，請將您的程式與連結 `nothrownew.obj`。不過，當您使用連結時 `nothrownew.obj`，`operator new` C++ 標準程式庫中的預設值將不再有作用。

如需 C 執行時間程式庫和 C++ 標準程式庫中的程式庫檔案清單，請參閱 [CRT 程式庫功能](#)。

## `new` 運算子

編譯器會將這類語句轉譯為函式呼叫 `operator new`：

```
char *pch = new char[BUFFER_SIZE];
```

如果要求是針對零位元組的儲存體，則會傳回 `operator new` 不同物件的指標。也就是，重複呼叫以傳回 `operator new` 不同的指標。如果配置要求的記憶體不足，則會擲回 `operator new` `std::bad_alloc` 例外狀況。或者，`nullptr` 如果您已連結非擲回支援，它會傳回 `operator new`。

您可以撰寫嘗試釋放記憶體的常式，並重試配置。如需詳細資訊，請參閱 [\\_set\\_new\\_handler](#)。如需修復配置的詳細資訊，請參閱 [處理記憶體不足](#) 一節。

下表說明函式的兩個範圍 `operator new`。

### 函數的範圍 `operator new`

III	III
<code>::operator new</code>	全球
類別名稱*** <code>::operator new</code> *	類別

的第一個引數 `operator new` 必須是類型 `size_t` (定義于中 `<stddef.h>`)，而且傳回類型一律為 `void*`。

`operator new` 當 `new` 運算子用來配置內建類型的物件、不包含使用者定義函數之類別類型的物件，`operator new` 以及任何類型的陣列時，會呼叫全域函數。當 `new` 運算子用來配置已定義之類別類型的物件時 `operator new`，會呼叫該類別的 `operator new`。

`operator new` 對於類別定義的函式是靜態成員函式(不能是虛擬的)，它會隱藏 `operator new` 該類別類型之物件的全域函式。請考慮 `new` 用來配置記憶體並將其設定為指定值的案例：

```

#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;
}

```

括弧中提供的引數 `new` 會傳遞至 `Blanks::operator new` 做為 `chInit` 引數。不過，全域函式 `operator new` 是隱藏的，導致下列程式碼產生錯誤：

```
Blanks *SomeBlanks = new Blanks;
```

編譯器支援類別宣告中的成員陣列 `new` 和 `delete` 運算子。例如：

```

class MyClass
{
public:
    void * operator new[] (size_t)
    {
        return 0;
    }
    void operator delete[] (void*)
    {
    }
};

int main()
{
    MyClass *pMyClass = new MyClass[5];
    delete [] pMyClass;
}

```

## 處理記憶體不足

測試失敗的記憶體配置可以完成，如下所示：

```
#include <iostream>
using namespace std;
#define BIG_NUMBER 100000000
int main() {
    int *pI = new int[BIG_NUMBER];
    if( pI == 0x0 ) {
        cout << "Insufficient memory" << endl;
        return -1;
    }
}
```

還有另一種方式可以處理失敗的記憶體配置要求。撰寫自訂的修復常式來處理這類失敗，然後藉由呼叫執行時間函式來註冊您的函數 [\\_set\\_new\\_handler](#)。

## delete 運算子

使用運算子動態配置的記憶體 `new` 可以使用 `delete` 運算子釋放。Delete 運算子會呼叫 `operator delete` 函數，將記憶體釋放回可用的集區。使用 `delete` 運算子也會呼叫類別析構函式(如果有的話)。

有全域和類別範圍的函式 `operator delete`。指定的類別只能定義一個函式 `operator delete`，如果已定義，則會隱藏全域 `operator delete` 函數。`operator delete` 針對任何類型的陣列，一律會呼叫全域函數。

全域 `operator delete` 函數。全域 `operator delete` 和類別成員函式有兩種形式 `operator delete`：

```
void operator delete( void * );
void operator delete( void *, size_t );
```

針對指定的類別，只有上述兩種形式的其中一種可以存在。第一個表單採用類型的單一引數 `void *`，其中包含要解除配置的物件指標。第二個表單(調整大小的解除配置)採用兩個引數：第一個是要解除配置的記憶體區塊指標，而第二個是要解除配置的位元組數目。這兩種形式的傳回類型為 `void` (`operator delete` 無法傳回值)。

第二種形式的目的是要加速搜尋要刪除之物件的正確大小類別目錄。這種資訊通常不會儲存在配置本身附近，而且可能會進行快取。當來自基類的函式 `operator delete` 用來刪除衍生類別的物件時，第二種形式會很有用。

函式 `operator delete` 是靜態的，因此不能是虛擬的。函 `operator delete` 式會遵守存取控制，如[成員存取控制](#)中所述。

下列範例會顯示 `operator new` `operator delete` 設計用來記錄配置和記憶體取消配置的使用者定義和函數：

```

#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;
}

```

上述程式碼可用來偵測「記憶體流失」，也就是在免費存放區上配置但從未釋放的記憶體。若要偵測流失，全域 `new` 和 `delete` 運算子會重新定義，以計算記憶體的配置和解除配置。

編譯器支援類別宣告中的成員陣列 `new` 和 `delete` 運算子。例如：

```

// 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;
}

```

# 智慧型指標 (新式 C++)

2020/11/2 • [Edit Online](#)

在新式 C++ 程式設計中，標準程式庫包含 智慧型指標，可用來協助確保程式沒有記憶體和資源流失，而且是例外狀況安全的。

## 智慧型指標的用途

智慧型指標是在 `std` 標頭檔的命名空間中定義 `<memory>`。這些是對 RAI<sup>1</sup> 或資源取得的關鍵，就是初始化程式設計的用法。這個慣用語主要目標是確保在初始化物件的同時會進行擷取資源，使建立物件的所有資源在一行程式碼中建立並且準備就緒。實際上，RAI 的主要原則就是將任何堆積配置資源的擁有權（例如，動態配置記憶體或系統物件控制代碼）提供給含有刪除或釋放資源程式碼的解構函式，以及任何相關清除程式碼的堆疊配置物件。

大部分情況下，當您初始化原始指標或資源控制代碼以指向實際資源時，會立即將指標傳遞給智慧型指標。在現代 C++ 中，原始指標只能用於極重視效能且擁有權不會混淆之有限範圍的小型程式碼區塊、迴圈或 Helper 函式。

下列範例將原始指標宣告與智慧型指標宣告做比較。

```
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.
```

如範例所示，智慧型指標是在堆疊上宣告、並且以指向堆積配置物件的原始指標初始化的類別樣板。智慧型指標在初始化後，就會擁有原始指標。這表示智慧型指標會刪除原始指標指定的記憶體。智慧型指標解構函式包含對 `delete` 的呼叫，而且因為智慧型指標是在堆疊上宣告，因此當智慧型指標超出範圍時，便會叫用其解構函式，即使例外狀況是從比堆疊更上方的位置擲回亦然。

使用熟悉的指標運算子 `->` 和 `*`（智慧型指標類別會進行多載以傳回封裝的原始指標）存取封裝的指標。

C++ 智慧型指標慣用語與在 C# 等語言中建立物件的情形類似：您會建立物件，然後讓系統負責在適當時機將其刪除。差別在於背景中沒有執行是沒有個別記憶體回收行程；記憶體是透過標準 C++ 範圍設定規則來管理，因此執行階段環境會更快速且更有效率。

## IMPORTANT

請一律在不同的程式碼行上建立智慧型指標，絕不可在參數清單建立，如此可避免因某些參數清單配置規則，而發生無法預測的資源流失。

下列範例示範如何 `unique_ptr` 使用 c + + 標準程式庫中的智慧型指標類型，將指標封裝至大型物件。

```
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.
```

下列範例示範使用智慧型指標的基本步驟。

1. 將智慧型指標宣告為自動 (區域) 變數。(不使用 `new` `malloc` 智慧型指標本身的或運算式。)
2. 在類型參數中，指定封裝指標的指向類型。
3. 將原始指標傳遞至 `new` 智慧型指標函式中的-ed 物件。(某些公用程式函式或智慧型指標建構函式即可進行此步驟)。
4. 使用多載的 `->` 和 `*` 運算子來存取物件。
5. 讓智慧型指標刪除物件。

智慧型指標是針對在記憶體和效能兩方面都達到最大效率而設計。例如，封裝的指標 `unique_ptr` 的唯一資料成員。這表示 `unique_ptr` 與該指標有完全相同的大小，四個位元組或八個位元組。使用智慧型指標多載的 \* 和-> 運算子來存取封裝的指標，會比直接存取原始指標慢很多。

智慧型指標有自己的成員函式，可使用「點」標記法來存取。例如，某些 c + + 標準程式庫智慧型指標具有重設成員函式，可釋放指標的擁有權。當您要在智慧型指標超出範圍之前釋出智慧型指標所擁有的記憶體時，這將會很有用，如下列範例所示。

```

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...
}

```

智慧型指標通常會提供直接存取其原始指標的方法。C++ 標準程式庫智慧型指標具有 `get` 此用途的成員函式，且 `CComPtr` 具有公用 `p` 類別成員。智慧型指標可讓您直接存取基礎指標，您可以用來在自己的程式碼管理記憶體，而且仍然可以將原始指標傳遞至不支援智慧型指標的程式碼。

```

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());
}

```

## 智慧型指標的種類

下列章節中將摘要說明可在 Windows 程式設計環境中使用的各種不同智慧型指標，並描述其使用時機。

### C++ 標準程式庫智慧型指標

使用這些智慧型指標做為封裝純舊 C++ 物件 (POCO) 指標的首要選擇。

- `unique_ptr`

只允許一個基礎指標的擁有者。用做 POCO 的預設選項，除非您確信自己需要 `shared_ptr`。可以移至新擁有者，但不可複製或共用。替換已被取代的 `auto_ptr`。與 `boost::scoped_ptr` 比較。`unique_ptr` 很小且有效率；大小是一個指標，它支援從 C++ 標準程式庫集合快速插入和抓取的右值參考。標頭檔：`<memory>`。如需詳細資訊，請參閱 [如何：建立和使用 Unique\\_ptr 實例](#) 和 [unique\\_ptr 類別](#)。

- `shared_ptr`

參考計數的智慧型指標。在您想要將原始指標指派給多個擁有者時使用，例如，當您從容器傳回指標的複本，但是想要保留原來的指標時。原始指標只有在所有的 `shared_ptr` 擁有者都超出範圍或放棄擁有權之後，才會被刪除。大小是兩個指標；一個針對物件，另一個則針對含有參考計數的共用控制區塊。標頭檔：`<memory>`。如需詳細資訊，請參閱 [如何：建立和使用 Shared\\_ptr 實例](#) 和 [shared\\_ptr 類別](#)。

- `weak_ptr`

與 `shared_ptr` 一起使用的特殊案例智慧型指標。`weak_ptr` 可以讓您存取由一個或多個 `shared_ptr` 執行個體擁有的物件，但是不會參與參考計數。當您想要觀察物件，但不需要物件保持運作時，即可使用。在某些要中斷 `shared_ptr` 執行個體之間循環參考的情況下為必要項。標頭檔：`<memory>`。如需詳細資訊，請參閱 [如何：建立和使用 Weak\\_ptr 實例](#) 和 [weak\\_ptr 類別](#)。

## COM 物件的智慧型指標 (傳統的 Windows 程式設計)

當您使用 COM 物件時，請將介面指標包裝在適當的智慧型指標類型中。Active Template Library (ATL) 會定義數個各種用途的智慧型指標。您也可以使用 `_com_ptr_t` 智慧型指標類型，編譯器會在從 .tlb 檔案建立包裝函式類別時使用這個類型。當您不想要包含 ATL 標頭檔時，這是最佳選擇。

#### CComPtr 類別

除非無法使用 ATL，否則請使用這種方式。使用 `AddRef` 和 `Release` 方法執行參考計數。如需詳細資訊，請參閱 [如何：建立和使用 CComPtr 和 CComQIPtr 實例](#)。

#### CComQIPtr 類別

與 `CComPtr` 類似，但是有提供在 COM 物件上呼叫 `QueryInterface` 的簡化語法。如需詳細資訊，請參閱 [如何：建立和使用 CComPtr 和 CComQIPtr 實例](#)。

#### CComHeapPtr 類別

使用 `CoTaskMemFree` 釋放記憶體之物件的智慧型指標。

#### CComGIPtr 類別

由全域介面資料表 (GIT) 取得之介面的智慧型指標。

#### \_com\_ptr\_t 類別

在功能上與 `CComQIPtr` 類似，但是不相依於 ATL 標頭。

### POCO 物件的 ATL 智慧型指標

除了 COM 物件的智慧型指標之外，ATL 也會針對純舊的 c++ 物件定義智慧型指標和智慧型指標集合 (POCO)。在傳統的 Windows 程式設計中，這些類型是 c++ 標準程式庫集合的有用替代方案，特別是在不需要程式碼可攜性時，或當您不想要混用 c++ 標準程式庫和 ATL 的程式設計模型時。

#### CAutoPtr 類別

藉由傳送複本上的擁有權以強制唯一擁有權的智慧型指標。類似已被取代的 `std::auto_ptr` 類別。

#### CHheapPtr 類別

使用 C `malloc` 函數所配置之物件的智慧型指標。

#### CAutoVectorPtr 類別

用於以 `new[]` 所配置之陣列的智慧型指標。

#### CAutoPtrArray 類別

將 `CAutoPtr` 項目陣列封裝的類別。

#### CAutoPtrList 類別

將方法封裝以操作 `CAutoPtr` 節點清單的類別。

## 另請參閱

### 指標

[C++ 語言參考](#)

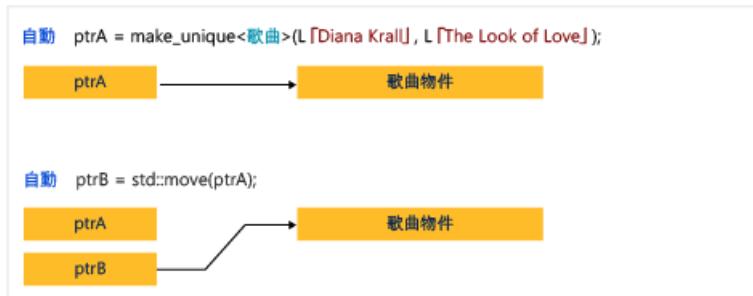
[C++ 標準程式庫](#)

# 如何：建立和使用 unique\_ptr 實例

2019/12/2 • [Edit Online](#)

`Unique_ptr`不會共用它的指標。它無法複製到另一個 `unique_ptr`，以傳值方式傳遞至函式，或用於C++任何需要進行複製的標準程式庫演算法。只能移動 `unique_ptr`。這表示記憶體資源的擁有權轉移到另一個 `unique_ptr`，原始 `unique_ptr` 不再擁有它。因為多重擁有權會增加程序邏輯的複雜度，建議您將物件限制為一個擁有者。因此，當您需要一般C++物件的智慧型指標時，請使用 `unique_ptr`，而當您建立 `unique_ptr` 時，請使用`make_unique` helper函數。

下圖說明兩個 `unique_ptr` 執行個體之間的擁有權轉移。



`unique_ptr` 是在C++標準程式庫的 `<memory>` 標頭中定義。它與原始指標一樣有效率，而且可以在標準程式庫容器C++中使用。將 `unique_ptr` 實例新增至C++標準程式庫容器的效率很高，因為 `unique_ptr` 的移動函式不需要複製作業。

## 範例 1

下列範例示範如何建立 `unique_ptr` 執行個體，並在函式之間傳遞這些執行個體。

```
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");
}
```

這些範例示範 `unique_ptr` 的這種基本特性：可加以移動，但無法複製。「移動」會將擁有權轉移到新的 `unique_ptr`，並重設舊的 `unique_ptr`。

## 範例 2

下列範例示範如何建立 `unique_ptr` 執行個體並在向量中使用這些執行個體。

```
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;
    }
}
```

在範圍 for 迴圈中，請注意 `unique_ptr` 是以傳址方式傳遞。如果您嘗試在這裡以傳值方式傳遞，編譯器就會擲回錯誤，因為 `unique_ptr` 複製建構函式已刪除。

## 範例 3

下列範例示範如何初始化本身為類別成員的 `unique_ptr`。

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

## 範例 4

您可以使用[make\\_unique](#)來建立陣列的 `unique_ptr`，但無法使用 `make_unique` 來初始化陣列元素。

```
// 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;
}
```

如需更多範例，請參閱[make\\_unique](#)。

## 另請參閱

[智慧型指標 \(現代 C++\)](#)

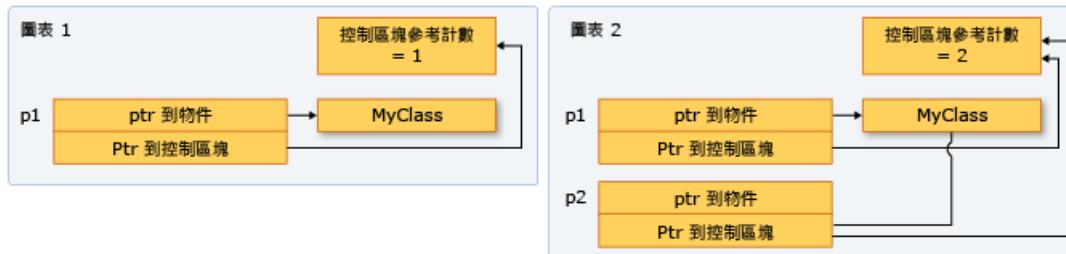
[make\\_unique](#)

# 如何：建立和使用 shared\_ptr 實例

2020/11/2 • [Edit Online](#)

`shared_ptr` 類型是 C++ 標準程式庫中的一種智慧型指標，是為有一個以上的擁有者可能必須管理物件在記憶體中的存留期之情節而設計。在您初始化 `shared_ptr` 之後，您可以函式引數中的值予以複製、傳送以及指派至其他 `shared_ptr` 執行個體。所有執行個體都會指向相同的物件，並共用對一個每當新的 `shared_ptr` 加入、超出範圍或重設時會遞增和遞減參考計數的「控制區塊」的存取。當參考計數達到零時，控制區塊會刪除記憶體資源和自己本身。

下圖顯示幾個指向一個記憶體位置的 `shared_ptr` 執行個體。



## 範例設定

以下範例假設您已包含必要的標頭，並宣告了必要類型，如此處所示：

```

// 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
}

```

## 範例 1

在任何可能的情況下，請在初次建立記憶體資源時使用 `make_shared` 函式來建立 `shared_ptr`。`make_shared` 是無例外狀況之虞。它會使用相同的呼叫來配置控制區塊的記憶體及資源，減少建構的額外負荷。如果您不使用 `make_shared`，則必須先使用明確 `new` 運算式來建立物件，然後再將它傳遞給此 `shared_ptr` 函數。下列範例顯示各種宣告和初始化 `shared_ptr` 及新物件的方式。

```

// Use make_shared function when possible.
auto sp1 = make_shared<Song>(L"The Beatles", L"I'm 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");

```

## 範例 2

下列範例顯示如何宣告和初始化 `shared_ptr` 執行個體，其具有已被另一個 `shared_ptr` 配置之物件的共用擁有權。假設 `sp2` 是已初始化的 `shared_ptr`。

```

//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);

```

## 範例 3

當您在 C++ 標準程式庫容器內使用會複製元素的演算法時，`shared_ptr` 也相當實用。您可以包裝 `shared_ptr` 中的項目，然後將它複製到能夠辨識只有需要時才有效（不再需要時則無效）之基礎記憶體的其他容器中。下列範例顯示如何在向量中的 `remove_copy_if` 執行個體上運用 `shared_ptr` 演算法。

```

vector<shared_ptr<Song>> v;

v.push_back(make_shared<Song>(L"Bob Dylan", L"The Times They Are A Changing"));
v.push_back(make_shared<Song>(L"Aretha Franklin", L"Bridge Over Troubled Water"));
v.push_back(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;
}

```

## 範例 4

您可以使用 `dynamic_pointer_cast`、`static_pointer_cast` 和 `const_pointer_cast` 轉換 `shared_ptr`。這些函數與 `dynamic_cast`、`static_cast` 和運算子類似 `const_cast`。下列範例顯示如何測試在基底類別的 `shared_ptr` 向量中每個項目的衍生類型，然後複製項目並顯示其相關資訊。

```
vector<shared_ptr<MediaAsset>> assets;

assets.push_back(shared_ptr<Song>(new Song(L"Himesh Reshammiya", L"Tera Surroor")));
assets.push_back(shared_ptr<Song>(new Song(L"Penaz Masani", L"Tu Dil De De")));
assets.push_back(shared_ptr<Photo>(new 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;
}
```

## 範例 5

您可以透過下列方式將 `shared_ptr` 傳遞至另一個函式：

- 以傳值方式傳遞 `shared_ptr`。這會叫用複製建構函式、遞增參考計數以及讓被呼叫端成為擁有者。此作業中有少量的額外負荷，也可能視您傳遞的 `shared_ptr` 物件多寡而變多。請在呼叫者和被呼叫者之間的隱含或明確程式碼合約要求被呼叫者成為擁有者時，使用此選項。
- 以傳址或 `const` 的傳址方式傳遞 `shared_ptr`。在這種情況下，參考計數不會遞增，並且只要呼叫者沒有離開範圍，被呼叫者都能存取該指標。或者，被呼叫者可以決定根據參考建立一個 `shared_ptr`，並成為共用擁有者。當呼叫端不了解被呼叫端或者您必須傳遞 `shared_ptr` 且要基於效能考量避免複製作業時，請使用這個選項。
- 將基底指標或參考傳遞至基礎物件。這可讓被呼叫者使用物件，但不會讓它共用擁有權或延長存留期。若被呼叫者從原始指標建立 `shared_ptr`，則新的 `shared_ptr` 會獨立於原始的 `shared_ptr`，並且不會控制基礎資源。當呼叫端和被呼叫端之間的協定明確指定呼叫端保留 `shared_ptr` 存留期的擁有權時，請使用這個選項。
- 在您決定如何傳遞 `shared_ptr` 時，請判斷被呼叫者是否必須共用基礎資源的擁有權。「擁有者」是一個只要需要時就讓基礎資源存活的物件或函式。如果呼叫端必須確保被呼叫端可以延長指標的壽命為超過其(函式的)存留期，請使用第一個選項。如果您不在乎被呼叫端是否延長存留期，則以傳址方式傳遞，並讓被呼叫端決定是否複製。
- 若您必須讓 helper 函式存取基礎指標，並且您知道 helper 函式只會在呼叫函式傳回之前使用指標並傳回，則該函式便不需要共用基礎指標的擁有權。它只需要存取呼叫端之 `shared_ptr` 的存留期內的指標。在這種情況下，以傳址方式傳遞 `shared_ptr`，或傳遞基礎物件的原始指標或參考是安全的。以這種方式傳遞有一小小的效能優點，並且也可以協助您表達程式設計的意圖。
- 在某些情況下，例如在 `std::vector<shared_ptr<T>>` 中，您可能必須將每個 `shared_ptr` 傳遞到 Lambda 運算式主體或具名函式物件。若 lambda 或函式並未儲存指標，請以傳址方式傳遞 `shared_ptr` 來避免叫用每個元素的複製建構函式。

## 範例 6

以下範例顯示 `shared_ptr` 如何多載各種比較運算子，以啟用 `shared_ptr` 執行個體所擁有之記憶體的指標比較。

```
// 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;
```

## 另請參閱

[智慧型指標\(現代 C++\)](#)

# 作法：建立和使用 `weak_ptr` 執行個體

2020/11/2 • [Edit Online](#)

有時候，物件必須儲存一種方法來存取`shared_ptr`的基礎物件，而不會導致參考計數遞增。一般而言，當您在實例之間有迴圈參考時，就會發生這種情況 `shared_ptr`。

最佳的設計是避免在每次可以時共用指標的擁有權。不過，如果您必須擁有實例的共用擁有權 `shared_ptr`，請避免其間的迴圈參考。當迴圈參考無法避免，甚至是基於某些原因而偏好時，請使用`weak_ptr`給一或多個擁有者提供另一個的弱式參考 `shared_ptr`。藉由使用 `weak_ptr`，您可以建立聯結 `shared_ptr` 至現有相關實例集的，但僅限於基礎記憶體資源仍然有效時。`weak_ptr` 本身不會參與參考計數，因此無法防止參考計數到達零。不過，您可以使用 `weak_ptr` 來嘗試取得 `shared_ptr` 其初始化所在的新複本。如果已刪除記憶體，則的 `weak_ptr` `bool` 運算子會傳回 `false`。如果記憶體仍然有效，新的共用指標會遞增參考計數，並保證只要 `shared_ptr` 變數留在範圍內，記憶體就會是有效的。

## 範例

下列程式碼範例示範 `weak_ptr` 用來確保適當刪除具有迴圈相依性之物件的案例。當您檢查範例時，假設它只是在考慮替代方案之後才建立的。`Controller` 物件代表電腦進程的某些層面，並獨立運作。每個控制站都必須能夠隨時查詢其他控制器的狀態，而且每個控制器都包含此用途的私用 `vector<weak_ptr<Controller>>`。每個向量都包含迴圈參考，因此 `weak_ptr` 會使用實例，而不是 `shared_ptr`。

```
#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)
        {
            try
            {
                auto p = wp.lock();
                wcout << L"Status of " << p->Num << " = " << p->Status << endl;
            }
        });
    }
}
```

```

        catch (bad_weak_ptr b)
        {
            wcout << L"Null object" << endl;
        }
    });
}

void RunTest()
{
    vector<shared_ptr<Controller>> v;

    v.push_back(shared_ptr<Controller>(new Controller(0)));
    v.push_back(shared_ptr<Controller>(new Controller(1)));
    v.push_back(shared_ptr<Controller>(new Controller(2)));
    v.push_back(shared_ptr<Controller>(new Controller(3)));
    v.push_back(shared_ptr<Controller>(new 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);
}

```

```
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
```

在實驗中，將向量修改 `others` 為，然後 `vector<shared_ptr<Controller>>` 在輸出中，請注意，當傳回時，不會叫用任何析構函數 `TestRun`。

## 另請參閱

[智慧型指標\(現代 c + +\)](#)

# 作法：建立和使用 CComPtr 與 CComQIPtr 執行個體

2020/11/2 • [Edit Online](#)

在傳統的 Windows 程式設計，程式庫通常是實作為 COM 物件（或更精確地說是 COM 伺服器）。許多 Windows 作業系統元件都會實作為 COM 伺服器，而且許多參與者提供這種形式的程式庫。如需 COM 基本概念的資訊，請參閱 [Component Object Model \(COM\)](#)。

當您具現化元件物件模型 (COM) 物件時，請將介面指標存放於 COM 智慧型指標，它在解構函式中使用 `AddRef` 和 `Release` 呼叫來執行參考計數。如果您使用 Active Template Library (ATL) 或 MFC 程式庫，則使用 `CComPtr` 智慧型指標。如果您不使用 ATL 或 MFC，則使用 `_com_ptr_t`。由於沒有 `std::unique_ptr` 的 COM 對等用法，為單一擁有者和多擁有者案例中使用這些智慧型指標。`CComPtr` 和 `ComQIPtr` 都支援有右值參考的移動作業。

## 範例：CComPtr

下列範例示範如何使用 `cComPtr` 以具現化 COM 物件和取得其介面的指標。請注意 `CComPtr::CoCreateInstance` 成員函式用以建立 COM 物件，而不是有相同名稱的 Win32 函式。

```

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` 和其親屬是 ATL 的一部分，而且是在中定義 `<atlcomcli.h>`。`_com_ptr_t` 在中宣告 `<comip.h>`。當編譯器產生類型程式庫的包裝函式類別時，編譯器會建立 `_com_ptr_t` 的特製化。

## 範例：CComQIPtr

ATL 也提供 `CComQIPtr`，查詢 COM 物件以擷取其他介面的語法更簡單。然而，建議使用 `CComPtr`，因為 `CComQIPtr` 可以執行的所有作業，它也可以執行，而且在語意上與原始 COM 介面指標更加一致。如果您使用 `CComPtr` 查詢介面，新介面指標是放在 `out` 參數中。如果呼叫失敗，傳回 `HRESULT`，這是一般的 COM 模式。使用 `CComQIPtr`，傳回值是指標本身，而且如果呼叫失敗，無法存取內部 `HRESULT` 傳回值。下列兩行顯示 `CComPtr` 和 `CComQIPtr` 的錯誤處理機制之間的差異。

```

// 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*/ }

```

## 範例：IDispatch

`CComPtr` 提供 `IDispatch` 特製化，讓它能夠儲存 COM Automation 元件指標並藉由使用晚期繫結叫用方法。

`CComDispatchDriver` 是 `CComQIPtr<IDispatch, &IID_IDispatch>` 的 typedef，它可以隱含地轉換為 `CComPtr<IDispatch>`。因此，當任何這三個名稱出現在程式碼中，它相當於 `CComPtr<IDispatch>`。下列範例顯示如何使用 `CComPtr<IDispatch>`，取得 Microsoft Word 物件模型的指標。

```

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();
}

```

## 另請參閱

[智慧型指標 \(新式 c + +\)](#)

# 編譯時期封裝的 Pimpl (現代 C++)

2019/12/2 • [Edit Online](#)

*Pimpl*的方法是用來C++隱藏執行的現代化技術，可將結合性降至最低，以及分隔介面。Pimpl 是「執行指標」的簡短。您可能已經熟悉此概念，但可透過其他名稱(例如 Cheshire Cat 或編譯器防火牆)來加以瞭解。

## 為何要使用 pimpl？

以下是 pimpl 的方法可以改善軟體發展生命週期的方式：

- 編譯相依性的小化。
- 區隔介面和實作為區隔。
- 相容。

## Pimpl 標頭

```
// my_class.h
class my_class {
    // ... all public and protected stuff goes here ...
private:
    class impl; unique_ptr<impl> pimpl; // opaque type here
};
```

Pimpl 的用法可避免重建級聯和脆弱物件版面配置。它非常適合(可轉移)的熱門類型。

## Pimpl 執行

定義 .cpp 檔案中的 `impl` 類別。

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

## 最佳作法

請考慮是否要新增對非擲回交換特製化的支援。

## 另請參閱

[歡迎回到C++](#)

[C++ 語言參考](#)

[C++ 標準程式庫](#)

# MSVC 中的例外處理

2020/4/15 • [Edit Online](#)

例外狀況是可能超出程式控制範圍的錯誤條件，它會使得程式無法繼續沿著其正常執行路徑進行。某些操作(包括物件創建、檔輸入/輸出和從其他模組進行的函數調用)都是異常的潛在來源,即使程式運行正確也是如此。穩定的程式碼會預測及處理例外狀況。要檢測邏輯錯誤,請使用斷言而不是異常(請參閱[使用斷言](#))。

## 例外類型

Microsoft C++ 編譯器 (MSVC) 支援三種異常處理:

- [C++異常處理](#)

對於大多數C++程式,應使用C++異常處理。它是類型安全的,並確保在堆疊展開期間調用物件析構函數。

- [結構化異常處理](#)

Windows 提供其自己的異常機制,稱為結構化異常處理 (SEH)。不建議用於C++或 MFC 程式設計。僅在非 MFC C 程式中使用 SEH。

- [MFC 例外狀況](#)

自版本 3.0 以來,MFC 一直使用C++異常。它仍然支援其較舊的異常處理宏,這些宏類似於窗體中C++異常。  
有關混合 MFC 宏和C++異常的建議,請參閱[例外:使用 MFC 宏和C++異常](#)。

使用[/EH](#)編譯器選項指定要在C++專案中使用的異常處理模型。標準C++異常處理 (/EHsc) 是 Visual Studio 中新C++專案中的預設值。

我們不建議您混合異常處理機制。例如,不要使用C++異常進行結構化異常處理。使用C++異常處理獨佔功能可使代碼更加便於使用,並且允許您處理任何類型的異常。有關結構化異常處理的缺點的詳細資訊,請參閱[結構化異常處理](#)。

## 本節內容

- [用於例外和錯誤處理的現代C++最佳實務](#)
- [如何設計異常安全](#)
- [如何連結例外狀況與非例外狀況代碼](#)
- [try、catch 和 throw 陳述式](#)
- [Catch 區塊的評估方式](#)
- [堆疊與堆疊](#)
- [例外規格](#)
- [noexcept](#)
- [未處理的 C++ 例外狀況](#)
- [混合 C \(結構化\) 和 C++ 例外狀況](#)
- [結構化例外狀況處理 \(SEH\) \(C/C++\)](#)

## 另請參閱

[C++語言參考](#)

[x64 例外狀況處理](#)

[例外處理\(C++/CLI 和C++/CX\)](#)

# 適用於例外狀況和錯誤處理的新式 C++ 最佳作法

2020/11/2 • [Edit Online](#)

在現代 C++ 中，在大部分情況下，報告和處理邏輯錯誤和執行階段錯誤的慣用方式是使用例外狀況。尤其是當堆疊可能在偵測到錯誤的函式之間包含數個函式呼叫，以及有內容可處理錯誤的函式時，更是如此。例外狀況會針對偵測錯誤的程式碼提供正式且妥善定義的方法，以將資訊傳遞至呼叫堆疊。

## 在例外狀況程式碼中使用例外狀況

程式錯誤通常分為兩種類別：程式設計錯誤所造成的邏輯錯誤，例如「索引超出範圍」錯誤。和程式設計人員的控制權以外的執行階段錯誤，例如「網路服務無法使用」錯誤。在 C 樣式程式設計和 COM 中，錯誤報表的管理方式是藉由傳回代表特定函式的錯誤碼或狀態碼的值，或是設定呼叫端在每次函式呼叫之後可選擇性地取出的全域變數，以查看是否已回報錯誤。例如，COM 程式設計會使用 HRESULT 傳回值，將錯誤傳達給呼叫者。和 WIN32 API 具有函式，`GetLastError` 可取得由呼叫堆疊回報的最後一個錯誤。在這兩種情況下，由呼叫端負責辨識程式碼並適當地回應。如果呼叫端未明確處理錯誤碼，則程式可能會損毀，而不會發出警告。或者，它可能會繼續使用不正確的資料執行，並產生不正確的結果。

現代 C++ 中偏好例外狀況的原因如下：

- 例外狀況會強制呼叫程式碼辨識錯誤狀況並加以處理。未處理的例外狀況會停止程式執行。
- 例外狀況會跳到可處理錯誤的呼叫堆疊中的點。中繼函數可讓例外狀況傳播。它們不需要與其他圖層協調。
- 例外狀況堆疊回溯機制會在擲回例外狀況之後，根據妥善定義的規則終結範圍中的所有物件。
- 例外狀況可讓偵測錯誤的程式碼和處理錯誤的程式碼之間有清楚的分隔。

下列簡化的範例顯示在 C++ 中擲回和捕捉例外狀況的必要語法。

```

#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++ 中的例外狀況類似于 C# 和 JAVA 等語言的例外狀況。在區塊中，如果擲回例外狀況，則會 `try` 由類型符合例外狀況的 `thrown` 第一個關聯區塊 `catch`。換句話說，執行會從 `throw` 語句跳至 `catch` 語句。如果找不到可使用的 `catch` 區塊，`std::terminate` 則會叫用並結束程式。在 C++ 中，可能會擲回任何類型；不過，我們建議您擲回直接或間接衍生的型別 `std::exception`。在上述範例中，例外狀況類型 `invalid_argument` 是在標頭檔的標準程式庫中定義 `<stdexcept>`。C++ 不會提供或要求 `finally` 區塊，以確保在擲回例外狀況時釋放所有資源。資源取得是 (RAII 的初始化) 使用智慧型指標的用法，提供資源清除所需的功能。如需詳細資訊，請參閱 [如何：設計例外狀況安全性](#)。如需 C++ 堆疊回溯機制的詳細資訊，請參閱 [例外狀況和堆疊回溯](#)。

## 基本指導方針

在任何程式設計語言中，健全的錯誤處理是一項挑戰。雖然例外狀況提供數個支援良好錯誤處理的功能，但無法為您執行所有工作。若要實現例外狀況機制的優點，請在設計程式碼時記住例外狀況。

- 使用判斷提示來檢查應該永遠不會發生的錯誤。使用例外狀況來檢查可能發生的錯誤，例如，在公用函式的參數上輸入驗證的錯誤。如需詳細資訊，請參閱 [例外狀況與判斷](#) 提示一節。
- 當處理錯誤的程式碼與透過一或多個中間函式呼叫偵測錯誤的程式碼分開時，請使用例外狀況。當處理錯誤的程式碼與偵測到錯誤的程式碼緊密結合時，請考慮是否要在效能關鍵迴圈中使用錯誤碼。
- 針對可能會擲回或傳播例外狀況的每個函式，提供三種例外狀況保證之一：強式保證、基本保證或 `nothrow` (`noexcept`) 保證。如需詳細資訊，請參閱 [如何：設計例外狀況安全性](#)。
- 依值擲回例外狀況，以傳址方式攔截例外狀況。請勿攔截您無法處理的內容。
- 請勿使用在 C++ 11 中已淘汰的例外狀況規格。如需詳細資訊，請參閱 [例外狀況 `noexcept` 規格](#) 和一節。
- 適用時，請使用標準程式庫例外狀況類型。從 `exception` 類別階層衍生自訂例外狀況類型。
- 不允許從析構函數或記憶體解除配置函式進行 escape 的例外狀況。

## 例外狀況和效能

如果沒有擲回例外狀況，則例外狀況機制會有最基本的效能成本。如果擲回例外狀況，堆疊遍歷和回溯的成本大致上相當於函式呼叫的成本。輸入區塊之後，需要額外的資料結構來追蹤呼叫堆疊 `try`，而如果擲回例外狀況，則需要其他指示來回溯堆疊。不過，在大部分情況下，效能和記憶體使用量的成本並不重要。例外狀況對效能的負面影響，可能只有在記憶體受限的系統上才有意義。或者，在效能關鍵的迴圈中，錯誤可能會定期發生，而且程式碼與處理它的程式碼之間有緊密的聯繫性。在任何情況下，不可能知道例外狀況的實際成本，而不需要進行分析和測量。即使在這種罕見的情況下，成本是很重要的，您也可以將它與設計完善的例外狀況原則所提供的更高正確性、更容易維護性以及其他優點進行權衡。

## 例外狀況與判斷提示

例外狀況和判斷提示是兩種不同的機制，用來偵測程式中的執行階段錯誤。`assert` 如果您的所有程式碼都是正確的，請使用語句在開發期間測試條件，而不應該為 `true`。由於錯誤指出程式碼中的某個內容必須固定，因此不會使用例外狀況來處理這類錯誤。它不代表程式必須在執行時間復原的條件。`assert` 會在語句中停止執行，如此您就可以在偵錯工具中檢查程式狀態。例外狀況會從第一個適當的 `catch` 處理常式繼續執行。使用例外狀況來檢查可能在執行時間發生的錯誤狀況，即使您的程式碼正確，例如「找不到檔案」或「記憶體不足」。例外狀況可以處理這些情況，即使復原只是將訊息輸出到記錄檔並結束程式。請一律使用例外狀況檢查公用函式的引數。即使您的函式是無錯誤的，您可能無法完全控制使用者可能傳遞給它的引數。

## C + + 例外狀況與 Windows SEH 例外狀況

C 和 c + + 程式都可以在 Windows 作業系統中使用結構化例外狀況處理 (SEH) 機制。SEH 中的概念與 c + + 例外狀況中的概念類似，不同之處在於 SEH 會使用 `_try`、`_except` 和 `_finally` 結構，而不是 `try` 和 `catch`。在 Microsoft c + + 編譯器 (MSVC) 中，c + + 例外狀況是針對 SEH 所執行。但是，當您撰寫 c + + 程式碼時，請使用 c + + 例外狀況語法。

如需 SEH 的詳細資訊，請參閱 [結構化例外狀況處理 \(C/c + +\)](#)。

## 例外狀況規格和 `noexcept`

例外狀況規格在 c + + 中是以指定函式可能擲回之例外狀況的方式來引進。不過，例外狀況規格在實務上已證明有問題，並已在 c + + 11 草稿標準中淘汰。我們建議您不要使用 `throw` 以外的例外狀況規格 `throw()`，這表示函式不允許任何例外狀況進行 escape。如果您必須使用已被取代之表單的例外狀況規格 `throw( type-name )`，MSVC 支援受限。如需詳細資訊，請參閱 [例外狀況規格 \(擲回\)](#)。`noexcept` 規範是在 c + + 11 中引進，做為的慣用替代方法 `throw()`。

## 請參閱

[如何：例外狀況和非例外狀況程式碼之間的介面](#)

[C + + 語言參考](#)

[C + + 標準程式庫](#)

# 如何：針對例外狀況安全性設計

2020/11/2 • [Edit Online](#)

其中一個例外狀況機制的優點是與例外狀況相關資料一起執行，直接從擲回例外狀況的陳述式跳躍到處理它的第一個 catch 陳述式。處理常式可能是在呼叫堆疊的任意層級。在 try 陳述式和 throw 陳述式之間呼叫的函式不必知道任何關於擲回例外狀況的事情。然而，它們必須經過設計，才能在可能從底下傳播例外狀況的任何點意外地超出範圍，而這麼做並不會留下部分建立的物件、流失的記憶體或在無法使用狀態的資料結構。

## 基本技巧

強大例外狀況處理原則需要仔細考量，且應該納入設計程序的一部分。一般而言，大部分例外狀況會在軟體模組的較低層被偵測到並擲回，不過，這些圖層通常沒有足夠的內容去處理錯誤，或公開訊息給終端使用者。在中介層，當必須檢查例外狀況物件時，函式可以攔截並重新擲回例外狀況，或有其他實用資訊提供給最後攔截例外狀況的最上層。只要可以完全復原，函式就應該攔截並「忍受」例外狀況。在大部分情況下，中介層的正確行為是讓例外狀況散佈到呼叫堆疊。在最高層，如果例外狀況讓程式無法保證正確性，最好讓未處理的例外狀況終止程式。

不論函式如何處理例外狀況，為了確保「例外狀況時仍然安全」，它必須根據下列基本規則設計。

### 讓資源類別保持簡單

當您在類別中封裝手動資源管理時，請使用沒有任何功能的類別，除了管理單一資源以外。藉由讓類別保持簡單，您可以降低引進資源流失的風險。盡可能使用智慧型指標，如下列範例所示。當使用 `shared_ptr` 時，這個範例會刻意人工化或簡單化以醒目提示差異之處。

```

// 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 RAI 用法來管理資源

若要讓例外狀況安全，函式必須確定已使用或所配置的物件已終結 `malloc` `new`，而且所有資源（例如檔案控制代碼）都已關閉或釋放，即使擲回例外狀況也是一樣。資源取得是初始化(RAI)將這類資源的管理系統至自動變數的存留期。當藉由正常傳回或因發生例外狀況造成函式超出範圍時，會叫用所有完整建構之自動變數的解構函式。RAI 包裝函式物件（例如智慧型指標）會呼叫其解構函式之適當刪除或終止函式。在例外狀況安全程式碼中，必須立即將每個資源擁有權傳遞到某種 RAI 物件。請注意 `vector`、`string`、`make_shared`、`fstream` 和類似的類別會為您處理資源的取得。不過，`unique_ptr` 和傳統 `shared_ptr` 的結構是特殊的，因為資源的取得是由使用者執行，而不是物件，因此，它們會計算為資源釋放的損毀，但也是可懷疑為 RAI 的問題。

# 這三個例外狀況保證

一般而言，例外狀況安全性會根據函式可提供的三個例外狀況來加以討論：不會失敗保證、強式保證和基本保證。

## 否-無法保證

無失誤（或「無擲回」）保證是函式可提供之最強保證。它表示函式不會擲回例外狀況，也不會允許散佈。不過，您無法可靠地提供這類保證，除非（a）您知道所有函式，且函式呼叫也是無失誤，或（b）您知道在到達這個函式之前擲回的例外狀況都被攔截，或（c）您知道如何攔截並正確處理可能會到達這個函式的所有例外狀況。

強烈保證與基本保證都假設解構函式是無誤的。標準程式庫中的所有容器和類型保證其解構函式不會擲回。也有相反的需求：標準程式庫會要求所提供的使用者定義類型（例如範本引數），必須具有非擲回解構函式。

## 強式保證

強力保證表示，如果因為發生例外狀況而使函式超出範圍，則不會流失記憶體，且程式狀態不會被修改。提供強力保證的函式基本上是有認可或復原語意的異動，不是完全成功，就是沒有作用。

## 基本保證

基本保證是三種保證中最弱的一個。不過，當強力保證在記憶體耗用量或在效能方面過於昂貴時，基本保證可能是最好的選擇。基本的保證表示，如果發生例外狀況，記憶體不會流失，而且物件仍處於可使用狀態，即使資料可能已經被修改。

# 例外狀況安全類別

此類別可確保其本身的例外狀況安全性，即使它是由不安全的函式使用，因為它可免於被部分建構或終結。如果類別建構函式在完成之前結束，則不會建立物件，且絕不會呼叫其解構函式。雖然在例外狀況之前初始化的自動變數都會叫用其解構函式，但會流失不是由智慧型指標或類似的自動變數管理的動態配置記憶體或資源。

內建類型都是無誤的，因此標準程式庫類型會支援最少的基本保證。遵循必須是例外狀況安全之使用者定義類型的方針：

- 使用智慧型指標或其他 RAII 類型的包裝函式來管理所有資源。避免類別解構函式的資源管理功能，因為如果建構函式擲回例外狀況，將不會叫用解構函式。不過，如果類別是只控制一個資源的專屬資源管理員，則使用解構函式管理資源是可接受的。
- 了解基底類別建構函式所擲回的例外狀況在衍生類別建構函式中是不可以被忍受的。如果您要轉譯和重新擲回在衍生類別建構函式的基底類別例外狀況，請使用函式 try 區塊。
- 考慮是否在智慧型指標包裝的資料成員中儲存所有類別狀態，特別是如果類別有「允許初始化失敗」的概念。雖然 C++ 允許未初始化的資料成員，但不支援未初始化或部分初始化的類別執行個體。建構函式必須不是成功就是失敗，因為如果建構函式沒有執行到完成，則不會建立物件。
- 不允許任何例外狀況從解構函式逸出。C++ 基本原則為解構函式不得允許例外狀況散佈到呼叫堆疊。如果解構函式必須執行可能會擲回例外狀況的作業，它必須在 try catch 區塊做這動作，且忍受例外狀況。標準程式庫會在其定義之所有解構函式提供這項保證。

## 另請參閱

[適用於例外狀況和錯誤處理的新式 C++ 最佳作法](#)

[如何：在例外狀況和非例外狀況程式碼之間進行介面](#)

# 如何：在例外狀況和非例外狀況程式碼之間進行介面

2020/11/2 • [Edit Online](#)

本文說明如何在 C++ 模組實作一致的例外狀況處理，以及如何將這些例外狀況與在例外狀況界限上的錯誤碼來回轉譯。

有時候 C++ 模組必須與不使用例外狀況的程式碼（非例外狀況程式碼）接合。這類介面就是所謂的例外狀況界限。例如，可以在 C++ 程式中呼叫 Win32 函式 `CreateFile`。`CreateFile` 不會擲回例外狀況，而是設定可由 `GetLastError` 函式擷取的錯誤碼。如果您的 C++ 程式為非一般的，您可能想要有一致的例外狀況架構錯誤處理原則。而且您可能不想要因為您正在使用非例外狀況程式碼而放棄例外狀況，也不想要在 C++ 模組中混合例外狀況架構和非例外狀況架構的錯誤處理原則。

## 從 C++ 呼叫非例外狀況函式

當您從 C++ 呼叫非例外狀況函式，這個概念是將函式包裝在可偵測所有錯誤然後擲回例外狀況的 C++ 函式中。當您設計這類包裝函式時，先決定要提供哪種例外狀況保證：不擲回、強式或基本。其次，設計函式，以致擲回例外狀況時正確地釋放所有資源，例如，檔案控制代碼。通常，這表示您使用智慧型指標或類似的資源管理員擁有資源。如需設計考慮的詳細資訊，請參閱[如何：針對例外狀況安全性進行設計](#)。

### 範例

下列範例顯示 C++ 函式內部使用 Win32 `CreateFile` 和 `ReadFile` 函式開啟和讀取兩個檔案。`File` 類別是檔案控制代碼的「資源擷取為初始設定」（Resource Acquisition Is Initialization, RAII）包裝函式。其建構函式偵測「找不到檔案」條件並擲回例外狀況，以將錯誤傳播至 C++ 模組的呼叫堆疊（在此範例中為 `main()` 函式）。如果於 `File` 物件完全建構之後擲回例外狀況，解構函式會被自動呼叫 `CloseHandle` 以釋放檔案控制代碼（如果您想要的話，可以使用 Active Template Library (ATL) `CHandle` 類別來進行相同的用途，或 `unique_ptr` 搭配自訂的刪除者。）呼叫 Win32 和 CRT API 的函式會偵測錯誤，然後使用本機定義的函式擲回 C++ 例外狀況，而此函式 `ThrowLastErrorIf` 接著會使用 `Win32Exception` 衍生自類別的類別 `runtime_error`。這個範例中的所有函式提供強式例外狀況保證；如果例外狀況在這些函式的任何時候擲回，資源不會流失，也不會修改程式狀態。

```
// 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 ThrowLastErrorHandlerIf(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);
        ThrowLastErrorHandlerIf(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);
    ThrowLastErrorHandlerIf(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);
    ThrowLastErrorHandlerIf(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
    }
}

```

## 從非例外狀況程式碼呼叫例外狀況程式碼

宣告為「extern C」的 C++ 函式可以由 C 程式呼叫。C++ COM 伺服器可由一些不同的語言撰寫的程式碼使用。當您實作由非例外狀況程式碼呼叫的 C++ 公用例外狀況感知函式，C++ 函式不應該允許任何例外狀況傳播回呼叫端。因此，C++ 函式必須明確攔截它知道如何處理的例外狀況，如果可行，將例外狀況轉換成錯誤碼以讓呼叫端了解例外狀況。如果不是所有可能的例外狀況都是已知，C++ 函式應該具有 `catch(...)` 區塊做為最後一個處理常式。在這種情況下，因為您的程式可能處於未知的狀態，最好向呼叫端報告嚴重錯誤。

下列範例會示範函式假設可能擲回的任何例外狀況是 `Win32Exception` 或衍生自 `std::exception` 的例外狀況類型。函式攔截這些類型的任何例外狀況，並將錯誤資訊做為 Win32 錯誤碼傳遞給呼叫端。

```

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;
}

```

當您將例外狀況轉換為錯誤碼時，一個可能發生的問題是錯誤碼通常不包含例外狀況可以儲存的豐富資訊。若要解決這個問題，您可以 `catch` 針對每個可能擲回的特定例外狀況類型提供區塊，並執行記錄以記錄例外狀況的詳細資料，然後再將其轉換為錯誤碼。如果多個函式都使用同一組區塊，這種方法可能會建立大量的程式碼重複 `catch`。避免程式碼重複的好方法，就是將這些區塊重構成一個私用公用程式函式，此函式會執行 `try` 和 `catch` 區塊，並接受在區塊中叫用的函式物件 `try`。在每個公用函式，將程式碼做為 Lambda 運算式傳遞至這個公用程式函式。

```

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;
}

```

下列範例顯示如何撰寫定義仿函式的 Lambda 運算式。當仿函式用 Lambda 運算式「內嵌」定義時，通常比寫入為具名函式物件更為容易讀取。

```
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;
    });
}
```

如需 lambda 運算式的詳細資訊，請參閱[Lambda 運算式](#)。

## 另請參閱

[適用于例外狀況和錯誤處理的新式 C++ 最佳作法](#)

[如何：針對例外狀況安全性設計](#)

# try、throw 和 catch 陳述式 (C++)

2020/11/2 • [Edit Online](#)

若要在 C++ 中執行例外狀況處理，您可以使用 `try`、`throw` 和 `catch` 運算式。

首先，使用 `try` 區塊來括住一個或多個可能會擲回例外狀況的語句。

`throw` 運算式表示區塊中發生例外狀況（通常是錯誤）`try`。您可以使用任何類型的物件做為運算式的運算元 `throw`。這個物件通常用來傳達與錯誤有關的資訊。在大部分情況下，我們建議您使用 `std::exception` 類別或在標準程式庫中定義的其中一個衍生類別。如果有任何不適用的類別，建議您從 `std::exception` 自行衍生例外狀況類別。

若要處理可能擲回的例外狀況，請 `catch` 在區塊之後立即執行一或多個區塊 `try`。每個 `catch` 區塊會指定它可以處理的例外狀況類型。

這個範例會顯示 `try` 區塊和其處理常式。假設 `GetNetworkResource()` 透過網路連線取得資料，且兩個例外狀況類型是衍生自 `std::exception` 的使用者定義類別。請注意，語句中的參考會攔截到例外狀況 `const` `catch`。建議您依照值擲回例外狀況並依 `const` 參考攔截這些例外狀況。

## 範例

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

## 備註

子句之後的 `try` 程式碼是程式碼的保護區段。`throw` 運算式會擲回（也就是引發）例外狀況。子句之後的程式碼區塊 `catch` 是例外狀況處理常式。如果和運算式中的型別相容，這就是攔截擲回之例外狀況的處理程式。

`throw` `catch`。如需在區塊中管理類型相符的規則清單 `catch`，請參閱[如何評估 Catch 區塊](#)。如果 `catch` 語句指定省略號(...)，而不是類型，則 `catch` 區塊會處理每種類型的例外狀況。當您使用`/eha`選項進行編譯時，這些可能包括 C 結構化例外狀況和系統產生或應用程式產生的非同步例外狀況，例如記憶體保護、零除和浮點違規。因為 `catch` 區塊會在程式順序中進行處理，以尋找相符的類型，所以省略號處理常式必須是相關聯區塊的最後一個處理常式 `try`。使用 `catch(...)` 時請小心；除非 `catch` 區塊知道如何處理所攔截到的特定例外狀況，否則不可允許程式繼續執行。通常，`catch(...)` 區塊的用途是在程式停止執行之前記錄錯誤和執行特殊清除。

`throw` 沒有運算元的運算式會重新擲回目前正在處理的例外狀況。重新擲回例外狀況時，建議使用此表單，因為其中保留了原始例外狀況的多型類型資訊。這類運算式只能用在處理常式中 `catch`，或是在從處理常式呼叫的函式中 `catch`。重新擲回的例外狀況物件是原始的例外狀況物件，而不是複本。

```
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;
}
```

## 另請參閱

[適用于例外狀況和錯誤處理的新式 C++ 最佳作法](#)

[關鍵字](#)

[未處理的 C++ 例外](#)

[\\_uncaught\\_exception](#)

# 評估 Catch 區塊的方式 (C++)

2020/11/2 • [Edit Online](#)

C++ 可讓您擲回任何類型的例外狀況，不過，一般建議擲回衍生自 std::exception 的類型。指定與所擲回 catch 例外狀況相同類型的處理常式，或可攔截任何例外狀況類型的處理常式，可能會捕捉到 C++ 例外狀況。

如果擲回的例外狀況類型是類別，而該類別同時擁有一個或多個基底類別，則可以使用接受例外狀況類型的基底類別，以及接受例外狀況類型之基底參考的處理常式攔截例外狀況。請注意，如果是以參考攔截例外狀況，它會繫結程序至實際擲回的例外狀況物件，否則它會是複本 (就如同函式的引數)。

當擲回例外狀況時，可能會由下列類型的 catch 處理常式攔截到：

- 可以接受任何類型的處理常式 (使用省略語法)。
- 接受與例外狀況物件相同類型的處理常式；由於它是複本，因此 const 會忽略和修飾詞 volatile 。
- 可以接受與例外狀況物件相同類型之參考的處理常式。
- 接受與 const volatile 例外狀況物件相同類型之或形式參考的處理常式。
- 接受與例外狀況物件相同類型之基底類別的處理常式；由於它是複本，const 因此會忽略和修飾詞 volatile 。  
catch 基底類別的處理常式不能在 catch 衍生類別的處理常式之前。
- 可以接受與例外狀況物件相同類型的基底類別之參考的處理常式。
- 處理常式，可接受與例外狀況 const volatile 物件相同類型之基底或形式的參考。
- 可以接受指標的處理常式，擲回的指標物件可以透過標準指標轉換規則轉換成該指標。

catch 處理常式出現的順序很重要，因為指定區塊的處理常式 try 會依其外觀的順序進行檢查。例如，將基底類別的處理常式放在衍生類別的處理常式前面就是錯誤的範例。找到相符的 catch 處理常式之後，就不會檢查後續的處理常式。因此，省略號 catch 處理常式必須是其區塊的最後一個處理常式 try 。例如：

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

在此範例中，省略號 catch 處理常式是唯一檢查的處理常式。

## 另請參閱

[適用於例外狀況和錯誤處理的新式 C++ 最佳作法](#)



# C++ 中的例外狀況和堆疊回溯

2020/11/2 • [Edit Online](#)

在 C++ 例外狀況機制中，控制項是從 throw 陳述式移動到可以處理擲回類型的第一個 catch 陳述式。當到達 catch 語句時，會在稱為堆疊回溯的進程中，終結 throw 和 catch 語句之間範圍內的所有自動變數。堆疊回溯中的執行方式如下：

1. Control 會依照 `try` 正常的循序執行來到達語句。會執行區塊中的保護區段 `try`。
2. 如果在執行保護區段期間未擲回任何例外狀況，則 `catch` 不會執行緊接在區塊後面的子句 `try`。執行會繼續在 `catch` 相關聯區塊後面的最後一個子句之後的語句中 `try`。
3. 如果在執行受保護區段期間，或在受保護區段直接或間接呼叫的任何常式中擲回例外狀況，則會從運算元所建立的物件建立例外狀況物件 `throw`。（這表示可能會牽涉到複製的構造函式）。此時，編譯器 `catch` 會在較高的執行內容中尋找子句，以處理所擲回之類型的例外狀況，或可 `catch` 處理任何例外狀況類型的處理常式。`catch` 處理常式會依其在區塊後面的外觀順序進行檢查 `try`。如果找不到適當的處理常式，則會檢查下一個動態封閉 `try` 區塊。此程式會繼續進行，直到檢查最外層的封閉 `try` 區塊為止。
4. 如果仍然找不到相符的處理常式，或是回溯過程中、處理常式取得控制之前發生例外狀況，則會呼叫預先定義的執行階段函式 `terminate`。如果在擲回例外狀況後但回溯開始之前發生例外狀況，則會呼叫 `terminate`。
5. 如果找到相符的 `catch` 處理常式，而且它是以傳值方式捕捉，其型式參數就會藉由複製例外狀況物件來初始化。如果它是以傳址方式攔截，則會初始化參數以參考例外狀況物件。在型式參數初始化之後，回溯堆疊的處理序才會開始。這牽涉到在 `try` 與處理常式相關聯的區塊開頭與例外狀況的擲回網站之間，終結所有已完全建立（但尚未解構）的自動物件 `catch`。解構會依照建構的反向順序進行。`catch` 處理常式會執行，而且程式會在最後一個處理常式之後繼續執行，也就是在不是處理常式的第一個語句或結構中 `catch`。控制項只能透過擲回的例外狀況來輸入 `catch` 處理常式，而不是透過語句 `goto` 或 `case` 語句中的標籤 `switch`。

## 堆疊回溯範例

下列範例將示範堆疊如何在擲回例外狀況時回溯。在執行緒上的執行會從 `c` 中的 `throw` 陳述式跳至 `main` 中的 `catch` 陳述式，然後一路回溯每個函式。請注意 `Dummy` 物件建立的順序，以及物件超出範圍後終結的順序。另請注意，除了包含 `catch` 陳述式的 `main` 之外，其他函式都不會完成。函式 `A` 絶不會從其對 `B()` 的呼叫傳回，而且 `B` 絶不會從其對 `C()` 的呼叫傳回。如果您取消 `Dummy` 指標定義和對應 `delete` 陳述式的註解，然後執行程式，請注意指標絕不會刪除。這就說明函式未提供例外狀況保證時，可能發生的狀況。如需詳細資訊，請參閱〈如何：例外狀況的設計〉。如果您註解 `catch` 陳述式，就可以觀察程式因為未處理的例外狀況結束時，會發生什麼情況。

```
#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.
*/

```

# 例外狀況規格 ( throw · noexcept ) ( C + + )

2020/11/2 • [Edit Online](#)

例外狀況規格是一種 C + + 語言功能，指出程式設計人員對於可透過函式傳播的例外狀況類型的意圖。您可以使用例外狀況規格，指定函式不一定是由例外狀況結束。編譯器可以使用這種資訊來優化對函式的呼叫，並在發生未預期的例外狀況時，終止程式。

在 C + + 17 之前，有兩種例外狀況規格。Noexcept 規格是 C + + 11 的新功能。它會指定是否可以對函式進行轉義的可能例外狀況集合是空的。動態例外狀況規格(或 `throw(optional_type_list)` 規格)在 C + + 11 中已被取代，並已在 C + + 17 中移除，但除外 `throw()`，這是的別名 `noexcept(true)`。這個例外狀況規格的設計，是為了提供有關可從函式擲出哪些例外狀況的摘要資訊，但實際上卻發現問題。有一項動態例外狀況規格證明，這是一種非常有用的方法，那就是無條件的 `throw()` 規格。例如，函式宣告：

```
void MyFunction(int i) throw();
```

通知編譯器，函式不會擲回任何例外狀況。不過，在 /std: c + + 14 模式中，如果函式擲回例外狀況，這可能會導致未定義的行為。因此，我們建議使用 `noexcept` 運算子，而不是上述各項：

```
void MyFunction(int i) noexcept;
```

下表摘要說明 Microsoft C + + 例外狀況規格的執行方式：

規格	行為
<code>noexcept</code> <code>noexcept(true)</code> <code>throw()</code>	函式不會擲回例外狀況。在 /std 中: C + + 14 模式(這是預設值)， <code>noexcept</code> 和相等 <code>noexcept(true)</code> 。當從宣告或的函式擲回例外狀況 <code>noexcept</code> 時 <code>noexcept(true)</code> ，會叫用 <code>std::terminate</code> 。當從宣告為 <code>throw()</code> /std: C + + 14 模式的函式擲回例外狀況時，結果會是未定義的行為。不會叫用特定的函式。這是與 C + + 14 標準的分歧，因此編譯器必須叫用 <code>std::非預期</code> 的。 Visual Studio 2017 15.5 版本: 在 /std: C + + 17 模式中， <code>noexcept</code> <code>noexcept(true)</code> 和 <code>throw()</code> 都是相等的。在 /std: C + + 17 模式中， <code>throw()</code> 是的別名 <code>noexcept(true)</code> 。在 /std: C + + 17 模式中，從使用任何這些規格所宣告的函式擲回例外狀況時，會叫用 C + + 17 標準所需的 <code>std::terminate</code> 。
<code>noexcept(false)</code> <code>throw(...)</code> 無規格	函式可能會擲回任何類型的例外狀況。
<code>throw(type)</code>	(C + + 14 版本)函式可能會擲回類型的例外狀況 <code>type</code> 。編譯器會接受語法，但會將它解釋為 <code>noexcept(false)</code> 。在 /std: C + + 17 模式中，編譯器會發出 warning C5040。

如果在應用程式中使用例外狀況處理，則呼叫堆疊中必須有一個函式，該函式會在結束標記為、或之函式的外部範圍之前，處理擲回的例外狀況 `noexcept` `noexcept(true)` `throw()`。如果在擲回例外狀況的函式和處理例外狀況的函數之間呼叫的任何函式指定為 `noexcept`、`noexcept(true)` (或 `throw()` 在 /std: C + + 17 模式中)，則程式會在 `noexcept` 函數傳播例外狀況時終止。

函式的例外狀況行為取決於下列因素：

- 設定的語言標準編譯模式。
- 在 C 或 C++ 中編譯函式。
- 您使用的/EH編譯器選項。
- 您是否明確指定例外狀況規格。

C 函式不允許明確例外狀況規格。假設 C 函數不會在 /ehsc 下擲回例外狀況，而且可能會在 /ehs、/eha 或 /EHAc 下擲回結構化例外狀況。

下表摘要說明 c++ 函式是否可能會在各種編譯器例外狀況處理選項下擲回：

if	/EHSC	/EHS	/EHA	/EHAC
沒有例外狀況規格的 C++ 函式	是	是	是	是
具有 <code>noexcept</code> 、 <code>noexcept(true)</code> 或 <code>throw()</code> 例外狀況規格的 C++ 函式	否	否	是	是
具有 <code>noexcept(false)</code> 、 <code>throw(...)</code> 或 <code>throw(type)</code> 例外狀況規格的 C++ 函式	是	是	是	是

## 範例

```

// exception_specification.cpp
// compile with: /EHs
#include <stdio.h>

void handler() {
    printf_s("in handler\n");
}

void f1(void) throw(int) {
    printf_s("About to throw 1\n");
    if (1)
        throw 1;
}

void f5(void) throw() {
    try {
        f1();
    }
    catch(...) {
        handler();
    }
}

// invalid, doesn't handle the int exception thrown from f1()
// void f3(void) throw() {
//     f1();
// }

void __declspec(nothrow) f2(void) {
    try {
        f1();
    }
    catch(int) {
        handler();
    }
}

// only valid if compiled without /EHc
// /EHc means assume extern "C" functions don't throw exceptions
extern "C" void f4(void);
void f4(void) {
    f1();
}

int main() {
    f2();

    try {
        f4();
    }
    catch(...) {
        printf_s("Caught exception from f4\n");
    }
    f5();
}

```

```

About to throw 1
in handler
About to throw 1
Caught exception from f4
About to throw 1
in handler

```

## 另請參閱

[try、throw 和 catch 陳述式 \(C++\)](#)

[適用於例外狀況和錯誤處理的新式 c++ 最佳作法](#)

# noexcept (C++)

2020/11/2 • [Edit Online](#)

C ++ 11：指定函式是否可能擲回例外狀況。

## 語法

*noexcept-expression:*

**noexcept**

**noexcept ( 常數運算式 )**

## 參數

### 常數運算式

類型的常數運算式 `bool`，表示可能的例外狀況類型集合是否是空的。無條件版本相當於 `noexcept(true)`。

## 備註

*Noexcept 運算式*是一種例外狀況規格，也就是函式宣告的後置詞，代表一組可能會因任何結束函式之例外狀況的例外狀況處理常式而比對的類型。一元條件運算子 `noexcept( constant_expression )` *constant\_expression*產生的位置 `true`，以及其無條件同義字，指定可以結束函式的 `noexcept` 可能例外狀況類型集合是空的。也就是說，函式永遠不會擲回例外狀況，而且永遠不允許將例外狀況傳播到其範圍外。*Constant\_expression* 產生的運算子 `noexcept( constant_expression )` *constant\_expression* `false`，或缺少例外狀況規格(除了針對析構函數或解除配置函式以外)，表示可以結束函數的一組可能的例外狀況是所有類型的集合。

`noexcept` 只有在直接或間接呼叫的所有函式都是或時，才將函式標示 `noexcept` 為 `const`。編譯器不一定會檢查每個程式碼路徑是否有可能會反升至函式的例外狀況 `noexcept`。如果例外狀況結束已標記之函式的外部範圍，則會立即叫用 `noexcept std:: terminate`，而且不保證會叫用任何範圍內物件的析構函式。使用 `noexcept`，而不是動態例外狀況規範 `throw()`，其現在已在標準中被取代。我們建議您套用 `noexcept` 到任何不允許例外狀況傳播呼叫堆疊的函式。宣告函式時 `noexcept`，它可讓編譯器在數個不同的內容中產生更有效率的程式碼。如需詳細資訊，請參閱[例外狀況規格](#)。

## 範例

複製其引數的樣板函式，可能會 `noexcept` 在要複製的物件為一般舊資料類型(POD)時宣告。這類函式可以宣告如下：

```
#include <type_traits>

template <typename T>
T copy_object(const T& obj) noexcept(std::is_pod<T>)
{
    // ...
}
```

## 另請參閱

[適用于例外狀況和錯誤處理的新式 C++ 最佳作法](#)

[例外狀況規格 \(throw, noexcept\)](#)

# 未處理的 C++ 例外狀況

2020/11/2 • [Edit Online](#)

如果找不到目前例外狀況的相符處理常式(或省略號 `catch` 處理常式)，則 `terminate` 會呼叫預先定義的執行時間函數。(您也可以 `terminate` 在任何處理程式中明確地呼叫)。的預設動作 `terminate` 是呼叫 `abort`。如果您希望 `terminate` 在結束應用程式之前呼叫程式中的其他函式，請使用做為單一引數呼叫的函式名稱來呼叫 `set_terminate` 函式。您可以隨時在程式中呼叫 `set_terminate`。`terminate` 常式一律會呼叫指定為之引數的最後一個函數 `set_terminate`。

## 範例

下列範例會擲回 `char *` 例外狀況，不過，其中並不包含任何指定攔截 `char *` 類型例外狀況的處理常式。對 `set_terminate` 的呼叫會指示 `terminate` 呼叫 `term_func`。

```
// exceptions_Unhandled_Exceptions.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;
void term_func() {
    cout << "term_func was called by terminate." << endl;
    exit( -1 );
}
int main() {
    try
    {
        set_terminate( term_func );
        throw "Out of memory!"; // No catch handler for this exception
    }
    catch( int )
    {
        cout << "Integer exception raised." << endl;
    }
    return 0;
}
```

## 輸出

```
term_func was called by terminate.
```

`term_func` 函式應該終止程式或目前的執行緒，最好是透過呼叫 `exit` 的方式進行。如果未如預期進行而傳回至呼叫端，則會呼叫 `abort`。

## 另請參閱

[適用於例外狀況和錯誤處理的新式 C++ 最佳作法](#)

# 混用 C (結構化) 和 c + + 例外狀況

2020/11/2 • [Edit Online](#)

如果您想要撰寫可移植的程式碼，則不建議在 c + + 程式中使用結構化例外狀況處理 (SEH)。不過，您有時可能會想要使用 `/EHs` 和混合結構化例外狀況和 c + + 原始程式碼，而且需要一些可處理這兩種例外狀況的功能。因為結構化例外狀況處理常式沒有物件或具類型例外狀況的概念，所以無法處理 c + + 程式碼所擲回的例外狀況。不過，c + + `catch` 處理常式可以處理結構化例外狀況。C + + 例外狀況處理語法 (`try`、`throw`、`catch`) 不會被 c 編譯器接受，但結構化例外狀況處理語法 (`_try`、`_except`、`_finally`) 是由 c + + 編譯器支援。

[\\_set\\_se\\_translator](#) 如需如何將結構化例外狀況處理為 c + + 例外狀況的資訊，請參閱。

如果您混用結構化和 c + + 例外狀況，請注意下列潛在問題：

- C + + 例外狀況和結構化例外狀況無法在相同的函式中混合。
- `_finally` 即使在擲回例外狀況之後回溯期間，一律會執行終止處理常式 (區塊)。
- C + + 例外狀況處理可以在所有以編譯器選項編譯的模組中攔截並保留回溯語義 `/EH`，以啟用回溯語義。
- 在某些情況下，不會針對所有物件呼叫函式函數。例如，嘗試透過未初始化的函式指標進行函式呼叫時，可能會發生結構化例外狀況。如果函式參數是在呼叫之前所建立的物件，就不會在堆疊回溯期間呼叫這些物件的析構函式。

## 後續步驟

- [setjmp](#) [longjmp](#) 在 c + + 程式中使用或

請參閱 `setjmp` `longjmp` c + + 程式中有關使用和的詳細資訊。

- [使用 C++ 處理結構化例外狀況](#)

請參閱您可以使用 c + + 處理結構化例外狀況的方法範例。

## 請參閱

[適用於例外狀況和錯誤處理的新式 c + + 最佳作法](#)

# 使用 setjmp 和 longjmp

2020/11/2 • [Edit Online](#)

當使用 `setjmp` 和 `longjmp` 時，它們會提供執行非本機的方式 `goto`。它們通常用於 C 程式碼，以將執行控制傳遞給先前呼叫之常式中的錯誤處理或復原程式碼，而不需要使用標準的呼叫或傳回慣例。

## Caution

因為 `setjmp` 和 `longjmp` 不支援正確地銷毀在 C++ 編譯器之間可能的堆疊框架物件，而且因為它可能會因為無法優化本機變數而降低效能，所以不建議在 C++ 程式中使用它們。我們建議您 `try` 改用和 `catch` 結構。

如果您決定 `setjmp` `longjmp` 在 C++ 程式中使用和，也請包含 `<setjmp.h>` 或，以確保函式 `<setjmpex.h>` 與結構化例外狀況處理(SEH)或 C++ 例外狀況處理之間的正確互動。

## Microsoft 特定的

如果您使用 `/EH` 選項來編譯 C++ 程式碼，則會在堆疊回溯期間呼叫本機物件的析構函數。不過，如果您使用 `/ehs` 或 `/ehsc` 進行編譯，而您的其中一個函式使用 `noexcept` 呼叫，則視優化工具的狀態而定，該 `longjmp` 函數的析構函數回溯可能不會發生。

在可移植的程式碼中，`longjmp` 執行呼叫時，標準會明確不保證以框架為基礎之物件的正確銷毀，而且其他編譯器可能不會支援。為了讓您知道，在警告層級4，對的呼叫 `setjmp` 會導致警告 C4611: '\_setjmp' 與 C++ 物件銷毀之間的互動是不可移植的。

## 結束 Microsoft 專有

## 另請參閱

[混合 C \(結構化\) 和 C++ 例外狀況](#)

# 使用 C++ 處理結構化例外狀況

2020/11/2 • [Edit Online](#)

C 結構化例外狀況處理(SEH)和 c + + 例外狀況處理的主要差異在於 c + + 例外狀況處理模型會處理類型，而 C 結構化例外狀況處理模型則會處理某種類型的例外狀況;具體而言，就是 `unsigned int`。也就是說，C 例外狀況是以不帶正負號的整數值來識別，而 C++ 例外狀況則是以資料類型來識別。當 C 中引發了結構化例外狀況時，每個可能的處理常式都會執行一個篩選準則，檢查 C 例外狀況內容並判斷是否接受例外狀況、將它傳遞給其他處理常式，或予以忽略。C++ 中擲回例外狀況時，它可能是任何類型。

第二個差異是 C 結構化例外狀況處理模型稱為非同步，因為例外狀況是在一般控制流程的次要處發生。C + + 例外狀況處理機制是完全同步的，這表示例外狀況只會在擲回時才會發生。

當您使用`/ehs`或`/ehsc`編譯器選項時，不會有任何 c + + 例外狀況處理常式處理結構化例外狀況。這些例外狀況只會由 `__except` 結構化例外狀況處理常式或 `__finally` 結構化終止處理常式處理。如需相關資訊，請參閱[結構化例外狀況處理\(C/c++\)](#)。

在`/eha`編譯器選項下，如果 c + + 程式中引發 c 例外狀況，它可以由結構化例外狀況處理常式與其相關聯的篩選器或 c + + `catch` 處理常式(以動態方式接近例外狀況內容)來處理。例如，此範例 c + + 程式會在 c + + 內容中引發 C 例外狀況 `try`：

## 範例-在 c + + catch 區塊中攔截 C 例外狀況

```
// exceptions_Exception_Handling_Differences.cpp
// compile with: /EHc
#include <iostream>

using namespace std;
void SEHFunc( void );

int main() {
    try {
        SEHFunc();
    }
    catch( ... ) {
        cout << "Caught a C exception." << endl;
    }
}

void SEHFunc() {
    __try {
        int x, y = 0;
        x = 5 / y;
    }
    __finally {
        cout << "In finally." << endl;
    }
}
```

```
In finally.
Caught a C exception.
```

## C 例外狀況包裝函式類別

在上述的簡單範例中，C 例外狀況只能由省略號(...) `catch` 捕捉。處理器. 與該例外狀況類型或特性相關的資訊，

都不會傳遞至處理常式。雖然這個方法可行，但在某些情況下，您可能會想要在兩個例外狀況處理模型之間定義轉換，讓每個 C 例外狀況都與特定的類別相關聯。若要轉換一個，您可以定義 C 例外狀況「包裝函式」類別，這可以用或衍生自，以便將特定類別類型的屬性設為 C 例外狀況。如此一來，每個 C 例外狀況都可以由特定 c + + `catch` 處理常式分別處理，而不是在單一處理程式中進行。

您的包裝函式類別包含的介面可能會包括一些決定例外狀況值的成員函式，而且這些成員函式會存取 C 例外狀況模型所提供的延伸例外狀況內容資訊。您可能也會想要定義預設的函式和接受引數的函式 `unsigned int`（以提供基礎 C 例外狀況標記法）和位複製的函數。以下是 C 例外狀況包裝函式類別的可能執行：

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

若要使用這個類別，請在每次擲回 C 例外狀況時，安裝內部例外狀況處理機制所呼叫的自訂 C 例外狀況轉譯函式。在您的轉譯函式內，您可以擲回任何具類型的例外狀況（可能是 `SE_Exception` 類型或衍生自的類別類型 `SE_Exception`），以由適當的相符 c + + `catch` 處理常式攔截。轉譯函數可以改為傳回，這表示它未處理例外狀況。如果轉譯函數本身引發 C 例外狀況，則會呼叫[terminate](#)。

若要指定自訂轉譯函式，請使用您的轉譯函式名稱做為其單一引數，呼叫[`\_set\_se\_translator`](#)函式。您所撰寫的轉譯函式會針對具有區塊之堆疊上的每個函式呼叫呼叫一次 `try`。沒有預設的轉譯函數；如果您未藉由呼叫 `_set_se_translator` 來指定它，則 C 例外狀況只能由省略號 `catch` 處理常式攔截。

## 範例-使用自訂翻譯函數

例如，下列程式碼會安裝自訂轉譯函式，然後引發由 `SE_Exception` 類別包裝的 C 例外狀況：

```

// exceptions_Exception_Handling_Differences3.cpp
// compile with: /EHa
#include <stdio.h>
#include <eh.h>
#include <windows.h>

class SE_Exception {
private:
    SE_Exception() {}
    unsigned int nSE;
public:
    SE_Exception( SE_Exception& e ) : nSE(e.nSE) {}
    SE_Exception(unsigned int n) : nSE(n) {}
    ~SE_Exception() {}
    unsigned int getSeNumber() { return nSE; }
};

void SEFunc() {
    __try {
        int x, y = 0;
        x = 5 / y;
    }
    __finally {
        printf_s( "In finally\n" );
    }
}

void trans_func( unsigned int u, _EXCEPTION_POINTERS* pExp ) {
    printf_s( "In trans_func.\n" );
    throw SE_Exception( u );
}

int main() {
    _set_se_translator( trans_func );
    try {
        SEFunc();
    }
    catch( SE_Exception e ) {
        printf_s( "Caught a __try exception with SE_Exception.\n" );
        printf_s( "nSE = 0x%x\n", e.getSeNumber() );
    }
}

```

```

In trans_func.
In finally
Caught a __try exception with SE_Exception.
nSE = 0xc0000094

```

## 另請參閱

[混合 C \(結構化\) 和 C++ 例外狀況](#)

# Structured Exception Handling (C/C++)

2020/11/2 • [Edit Online](#)

結構化例外狀況處理 (SEH) 是 Microsoft 的 C 延伸模組，可正常地處理特定的異常程式碼情況，例如硬體錯誤。雖然 Windows 和 Microsoft C++ 支援 SEH，但是建議您使用 ISO 標準 C++ 例外狀況處理。它讓您的程式碼更具可攜性和彈性。不過，若要維護現有的程式碼或特定種類的程式，您仍然必須使用 SEH。

Microsoft 專用：

## 文法

```
try-except-statement :  
    __try compound-statement __except ( expression ) compound-statement  
  
try-finally-statement :  
    __try compound-statement __finally compound-statement
```

## 備註

使用 SEH，您可以確保資源（例如記憶體區塊和檔案）會在執行意外終止時正確釋放。您也可以使用不依賴語句的簡潔結構化程式碼 `goto` 或更精細的傳回碼測試，來處理特定問題（例如，記憶體不足）。

本文 `try-except` `try-finally` 中參考的和語句是 C 語言的 Microsoft 擴充功能。它們支援 SEH，讓應用程式在終止執行的事件之後取得程式的控制權。雖然 SEH 與 C++ 原始程式檔搭配運作，但它不是專為 C++ 所設計。如果您在使用 `/EHs` 或 `/EHsc` 選項編譯的 C++ 程式中使用 SEH，則會呼叫本機物件的析構程式，但其他執行行為可能不是您預期的行為。如需圖例，請參閱本文稍後的範例。在大部分的情況下，我們建議您不要使用 Microsoft C++ 編譯器也支援的 ISO 標準 [C++ 例外狀況處理](#)，而不是 SEH。使用 C++ 例外狀況處理，確保您的程式碼更具可攜性，而且您可以處理任何類型的例外狀況。

如果您有使用 SEH 的 C 程式碼，您可以將它與使用 C++ 例外狀況處理的 C++ 程式碼混合使用。如需詳細資訊，請參閱 [在 C++ 處理結構化例外狀況](#)。

有兩種 SEH 機制：

- [例外狀況處理常式](#) 或 `__except` 區塊，可回應或關閉例外狀況。
- [終止處理常式](#) 或 `__finally` 一律會呼叫的區塊，不論例外狀況是否會導致終止。

這兩種處理常式不同，但會透過稱為回溯 [堆疊](#) 的進程密切相關。當結構化例外狀況發生時，Windows 會尋找目前作用中的最近安裝的例外狀況處理常式。處理常式可以執行下列三項事項的其中一項：

- 無法辨識例外狀況，並將控制權傳給其他處理常式。
- 辨識例外狀況，但關閉它。
- 辨識例外狀況，並處理它。

可辨識例外狀況的例外狀況處理常式可能不在例外狀況發生時正在執行的函式中。它可能在堆疊上的函式中更高。目前執行中函式和堆疊框架上的所有其他函式都會終止。在此程式期間，會將堆疊 [展開](#)。也就是說，已終止函式的區域非靜態變數會從堆疊中清除。

回溯堆疊時，作業系統會呼叫您為每個函式所撰寫的任何終止處理常式。藉由使用終止處理常式，您可以清除資源，否則會因為異常終止而維持開啟。如果您已輸入重要區段，您可以在終止處理常式中結束它。當程式即

將關閉時，您可以進行其他的維護工作，例如關閉和移除暫存檔案。

## 後續步驟

- 撰寫例外狀況處理常式
- 撰寫終止處理常式
- 使用 C++ 處理結構化例外狀況

## 範例

如先前所述，如果您在 c++ 程式中使用 SEH 並使用或選項進行編譯，則會呼叫本機物件的析構函數 `/EHsc`。但是，如果您也使用 c++ 例外狀況，執行期間的行為可能不是您預期的行為。此範例示範這些行為差異。

```
#include <stdio.h>
#include <Windows.h>
#include <exception>

class TestClass
{
public:
    ~TestClass()
    {
        printf("Destroying TestClass!\r\n");
    }
};

__declspec(noinline) void TestCPPEX()
{
#ifdef CPPEX
    printf("Throwing C++ exception\r\n");
    throw std::exception("");
#else
    printf("Triggering SEH exception\r\n");
    volatile int *pInt = 0x00000000;
    *pInt = 20;
#endif
}

__declspec(noinline) void TestExceptions()
{
    TestClass d;
    TestCPPEX();
}

int main()
{
    __try
    {
        TestExceptions();
    }
    __except(EXCEPTION_EXECUTE_HANDLER)
    {
        printf("Executing SEH __except block\r\n");
    }

    return 0;
}
```

如果您使用 `/EHsc` 來編譯此程式碼，但未定義本機測試控制程式宏 `CPPEX`，則 `TestClass` 不會執行此函式。輸出如下所示：

```
Triggering SEH exception  
Executing SEH __except block
```

如果您使用 `/EHsc` 來編譯器代碼，並 `CPPEX` 使用 (來定義，以便擲回 `/DCPPEX` C++ 例外狀況)，則會 `TestClass` 執行此函式，而且輸出看起來像這樣：

```
Throwing C++ exception  
Destroying TestClass!  
Executing SEH __except block
```

如果您使用 `/EHa` 來編譯器代碼，則會使用 `TestClass std::throw` 或使用 SEH 來觸發例外狀況，來執行此函式的擲回例外狀況。亦即，是否已 `CPPEX` 定義。輸出如下所示：

```
Throwing C++ exception  
Destroying TestClass!  
Executing SEH __except block
```

如需詳細資訊，請參閱 [/EH \(例外狀況處理模型\)](#)。

## 結束 Microsoft 專用

## 請參閱

[例外狀況處理](#)

[關鍵字](#)

[`<exception>`](#)

[錯誤和例外狀況處理](#)

[\(Windows\) 的結構化例外狀況處理](#)

# 撰寫例外狀況處理常式

2020/3/25 • [Edit Online](#)

例外狀況處理常式通常用來回應特定錯誤。除了知道如何處理的項目之外，您可以使用例外狀況處理語法篩選出所有例外狀況。其他例外狀況應該傳遞給為了尋找這些特定例外狀況所撰寫的其他處理常式（可能位於執行階段程式庫或作業系統中）。

例外狀況處理常式會使用 try-except 陳述式。

## 您還想知道關於哪些方面的詳細資訊？

- [Try-except 語句](#)
- [撰寫例外狀況篩選準則](#)
- [引發軟體例外狀況](#)
- [硬體例外狀況](#)
- [例外狀況處理常式的限制](#)

## 另請參閱

[結構化例外狀況處理 \(C/C++\)](#)

# try-except 陳述式

2020/11/2 • [Edit Online](#)

try-except 語句是 Microsoft 特定的擴充功能，可支援 C 和 c + + 語言的結構化例外狀況處理。

```
// . .
__try {
    // guarded code
}
__except ( /* filter expression */ ) {
    // termination code
}
// . . .
```

## 文法

```
try-except-statement :
    __try compound-statement __except ( expression ) compound-statement
```

## 備註

try-except 語句是 C 和 c + + 語言的 Microsoft 擴充功能。它可讓目標應用程式在通常會終止程式執行的事件發生時，掌握控制權。這類事件稱為 結構化例外狀況，或簡稱 例外 狀況。處理這些例外狀況的機制稱為 結構化例外狀況處理，(SEH)。

如需相關資訊，請參閱 [try finally 語句](#)。

例外狀況可能是以硬體為基礎或以軟體為基礎。即使應用程式無法從硬體或軟體例外狀況完全復原，結構化例外狀況處理仍很有用。SEH 可以顯示錯誤資訊，並將應用程式的內部狀態設為可協助診斷問題。它特別適用於不容易重現的間歇性問題。

### NOTE

結構化例外狀況處理可搭配 Win32 處理 C 和 C++ 原始程式檔。但是，它不是專為 c + + 所設計。使用 C++ 例外狀況處理可確保您的程式碼更具可移植性。此外，C++ 例外狀況處理更有彈性，因為它可以處理任何類型的例外狀況。針對 c + + 程式，建議您使用原生 c + + 例外狀況處理：[try、catch 和 throw 語句](#)。

子句後面的複合陳述式 `__try` 是 主體 或 受保護 的區段。`__except` 運算式也稱為篩選運算式。它的值會決定例外狀況的處理方式。子句後面的複合陳述式 `__except` 是例外狀況處理常式。如果在主體區段執行期間引發例外狀況，則處理常式會指定要採取的動作。執行程序如下所示：

1. 執行保護的區段。
2. 如果在執行受保護區段期間未發生任何例外狀況，則會在子句之後的語句繼續執行 `__except` 。
3. 如果在執行受保護區段期間發生例外狀況，或在受保護區段呼叫的任何常式中發生例外狀況，則 `__except` 會評估運算式。有三個可能的值：
  - `EXCEPTION_CONTINUE_EXECUTION` (-1) 例外狀況已關閉。在例外狀況發生的位置繼續執行。
  - `EXCEPTION_CONTINUE_SEARCH` 無法辨識 (0) 例外狀況。繼續搜尋堆疊中的處理常式，先針對包含

`try-except` 語句，然後針對具有下一個最高優先順序的處理常式。

- `EXCEPTION_EXECUTE_HANDLER` (1) 例外狀況被辨識。藉由執行複合陳述式來將控制權傳送至例外狀況處理常式 `_except`，然後在區塊之後繼續執行 `_except`。

`_except` 運算式會評估為 C 運算式。它受限於單一值、條件運算式運算子或逗號運算子。如果需要更廣泛的處理，運算式可以呼叫常式，傳回上面所列三個值的其中一個。

每個應用程式都有自己的例外狀況處理常式。

跳入語句是不正確 `_try`，但有效的方式就是跳出一個語句。如果進程在執行語句的中途結束，則不會呼叫例外狀況處理常式 `try-except`。

為了與舊版相容，`_try`、`_except` 和 `_leave` 是和的同義字 `_try`，`_except`，`_leave` 除非指定了編譯器選項 /za (停用語言延伸模組)。

#### `_leave` 關鍵字

`_leave` 關鍵字只有在語句的保護區段內才有效 `try-except`，而其效果是跳到受保護區段的結尾。然後在例外狀況處理常式之後的第一個陳述式繼續執行。

`goto` 語句也可以跳出受保護的區段，而不會降低效能，就像在 `try-catch` 語句中所做的一樣。這是因為堆疊回溯不會發生。不過，我們建議您使用 `_leave` 關鍵字而不是 `goto` 語句。原因是，如果受保護的區段很大或複雜，您較不可能犯程式設計錯誤。

#### 結構化例外狀況處理內建函式

結構化例外狀況處理提供兩個可搭配語句使用的內建函式 `try-except`：[GetExceptionCode](#) 和 [GetExceptionInformation](#)。

`GetExceptionCode` 傳回例外狀況的32位整數) 的程式碼 (。

內建函式會傳回 `GetExceptionInformation` `EXCEPTION_POINTERS` 結構的指標，其中包含有關例外狀況的其他資訊。透過這個指標就可以存取發生硬體例外狀況當時的電腦狀態。其結構如下所示：

```
typedef struct _EXCEPTION_POINTERS {
    PEXCEPTION_RECORD ExceptionRecord;
    PCONTEXT ContextRecord;
} EXCEPTION_POINTERS, *PEXCEPTION_POINTERS;
```

指標類型 `PEXCEPTION_RECORD` 和 `PCONTEXT` 定義于 include 檔中，而且 `<winnt.h>` `_EXCEPTION_RECORD` `_CONTEXT` 會定義于 include 檔中。`<excpt.h>`

您可以在 `GetExceptionCode` 例外狀況處理常式中使用。不過，您只能在 `GetExceptionInformation` 例外狀況篩選條件運算式內使用。它所指向的資訊通常是在堆疊上，而且當控制權轉移到例外狀況處理常式時，就無法再使用。

內建函式 `AbnormalTermination` 可在終止處理常式中使用。如果 `try-catch` 語句的主體依序終止，它會傳回0。在所有其他情況下，它會傳回 1。

`<excpt.h>` 定義這些內建函式的一些替代名稱：

`GetExceptionCode` 相當於 `_exception_code`

`GetExceptionInformation` 相當於 `_exception_info`

`AbnormalTermination` 相當於 `_abnormal_termination`

## 範例

```

// 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");
}

```

## 輸出

```

hello
in try
in try
in filter.
caught AV as expected.
in finally. termination:
    abnormal
in except
world

```

## 請參閱

[撰寫例外狀況處理常式](#)

[Structured Exception Handling \(C/C++\)](#)

[關鍵字](#)



# 撰寫例外狀況篩選條件

2020/4/15 • [Edit Online](#)

您可以藉由跳至例外狀況處理常式的層級或繼續執行的方式處理例外狀況。您可以使用篩選器來清理問題，然後返回 -1，在不清除堆疊的情況下恢復正常流，而不是使用異常處理程式代碼來處理異常並中斷。

## NOTE

某些例外狀況無法繼續執行。如果篩選器對於此類異常計算為 -1，則系統將引發新的異常。當您呼叫「[提升異常](#)」時，您可以確定異常是否將繼續。

例如，以下代碼在篩選器表示式中使用函數呼叫：此函數處理問題，然後返回 -1 以復原正常的控制流：

```
// exceptions_Writing_an_Exception_Filter.cpp
#include <windows.h>
int main() {
    int Eval_Exception( int );

    __try {}

    __except ( Eval_Exception( GetExceptionCode( )) ) {
        ;
    }

    void ResetVars( int ) {}
    int Eval_Exception ( int n_except ) {
        if ( n_except != STATUS_INTEGER_OVERFLOW &&
            n_except != STATUS_FLOAT_OVERFLOW ) // Pass on most exceptions
        return EXCEPTION_CONTINUE_SEARCH;

        // Execute some code to clean up problem
        ResetVars( 0 ); // initializes data to 0
        return EXCEPTION_CONTINUE_EXECUTION;
    }
}
```

最好在篩選器運算式中使用函數調用，每當篩選器需要執行任何複雜操作時。評估運算式會讓函式執行，在這個案例中是 `Eval_Exception`。

請注意使用 `GetExceptionCode` 來確定異常。您必須在 filter 本身內呼叫這個函式。`Eval_Exception` 無法呼叫 `GetExceptionCode`，但它必須將異常代碼傳遞給它。

除非例外狀況是整數或浮點溢位，否則這個處理常式會將控制項傳遞至另一個處理常式。如果是，處理常式會呼叫函式（`ResetVars` 只是範例，不是應用程式開發介面函式）重設部分全域變數。語句-塊-2（在此示例中為空）永遠無法執行，`Eval_Exception` 因為永遠不會返回 `EXCEPTION_EXECUTE_HANDLER` (1)。

使用函式呼叫是處理複雜的篩選條件運算式時理想的通用技術。另外兩項實用的 C 語言功能為：

- 條件運算子
- 逗號運算子

條件運算子通常很有用，因為它可以用來檢查特定傳回碼，然後傳回兩個不同值的其中一個。例如，只有在異常 `STATUS_INTEGER_OVERFLOW` 時，以下代碼中的篩選器才會識別異常：

```
__except( GetExceptionCode() == STATUS_INTEGER_OVERFLOW ? 1 : 0 ) {
```

這個案例中條件運算子的目的主要在於避免混淆，因為下列程式碼會產生相同的結果：

```
__except( GetExceptionCode() == STATUS_INTEGER_OVERFLOW ) {
```

條件運算子在您可能希望篩選器計算到 -1、EXCEPTION\_CONTINUE\_EXECUTION 的情況下更有用。

逗號運算子可讓您在單一運算式內執行多項獨立的運算。這個效果大致上與執行多個陳述式，然後傳回最後一個運算式的值相同。例如，下列程式碼會將例外狀況代碼儲存在變數中，然後進行測試：

```
__except( nCode = GetExceptionCode(), nCode == STATUS_INTEGER_OVERFLOW )
```

## 另請參閱

[編寫錯誤程式](#)

[Structured Exception Handling \(C/C++\)](#)

# 引發軟體例外狀況

2020/3/25 • • [Edit Online](#)

某些常見的程式錯誤來源是系統未標示為例外狀況。例如，如果您嘗試配置記憶體區塊，但是沒有足夠的記憶體，執行階段或 API 函式不會引發例外狀況，而是傳回錯誤碼。

不過，您可以藉由在程式碼中偵測該條件，然後藉由呼叫 `RaiseException` 函式來進行報告，來將任何條件視為例外狀況。以這種方式標幟錯誤，您就可以將結構化例外狀況處理的優點運用到任何類型的執行階段錯誤。

若要使用結構化例外狀況處理錯誤：

- 定義您自己的事件例外狀況代碼。
- 當您偵測到問題時，請呼叫 `RaiseException`。
- 使用例外狀況處理篩選條件測試您所定義的例外狀況代碼。

<winerror.h> 檔案顯示例外狀況代碼的格式。為確保您定義的代碼不會與現有的例外狀況代碼發生衝突，請將第三個最高有效位元設為 1。四個最高有效位元都應該依照下表所示進行設定。

BITS		
31-30	11	這兩個位元負責描述程式碼的基本狀態： 11 = 錯誤、00 = 成功、01 = 告知性、10 = 警告。
29	1	用戶端位元。設為 1，表示使用者定義的 程式碼。
28	0	保留位元 (保持設為 0)。

您可以將前兩個位元設定為 11 二進位以外的設定（如果您想要這樣做），不過「錯誤」設定適用於大部分例外狀況。重要的是，記得依照上表所示設定位元 29 和 28。

因此，產生的錯誤碼應該將最高四個位元設定為十六進位 E。例如，下列定義會定義不會與任何 Windows 例外狀況代碼發生衝突的例外狀況代碼。（不過，您可能需要查看協力廠商 DLL 使用哪些代碼）。

```
#define STATUS_INSUFFICIENT_MEM      0xE0000001
#define STATUS_FILE_BAD_FORMAT        0xE0000002
```

定義例外狀況代碼之後，就可以用它來引發例外狀況。例如，下列程式碼會引發 `STATUS_INSUFFICIENT_MEM` 例外狀況，以回應記憶體配置問題：

```
lpstr = _malloc( nBufferSize );
if (lpstr == NULL)
    RaiseException( STATUS_INSUFFICIENT_MEM, 0, 0, 0);
```

如果您只想要引發例外狀況，可以將最後三個參數設為 0。在傳遞額外資訊以及設定旗標防止處理常式繼續執行時，最後三個參數會很有用。如需詳細資訊，請參閱 Windows SDK 中的 `RaiseException` 函數。

然後就可以在您的例外狀況處理篩選條件中，測試您所定義的代碼。例如：

```
__try {
    ...
}
__except (GetExceptionCode() == STATUS_INSUFFICIENT_MEM ||
          GetExceptionCode() == STATUS_FILE_BAD_FORMAT )
```

## 另請參閱

[撰寫例外狀況處理常式](#)

[結構化例外狀況處理\(C++C/\)](#)

# 硬體例外狀況

2020/3/25 • [Edit Online](#)

大部分可由作業系統辨識的標準例外狀況是硬體定義的例外狀況。Windows 可辨識某些低階軟體例外狀況，不過，作業系統通常最適合處理這些例外狀況。

Windows 會將不同的處理器硬體錯誤對應到本節中的例外狀況代碼。在某些情況下，處理器可能只會產生這些例外狀況的子集。有關例外狀況以及發出適當例外狀況代碼的 Windows 前置處理資訊。

下表摘要說明 Windows 所辨識的硬體例外狀況：

例外狀況	說明
STATUS_ACCESS_VIOLATION	讀取或寫入無法存取的記憶體位置。
STATUS_BREAKPOINT	遇到硬體定義的中斷點，只能由偵錯工具使用。
STATUS_DATATYPE_MISALIGNMENT	在未正確對齊的位址讀取或寫入資料，例如，16 位元的實體必須在 2 個位元組的界限上對齊。（不適用於 Intel 80x86 處理器）。
STATUS_FLOAT_DIVIDE_BY_ZERO	除以 0.0 的浮點類型。
STATUS_FLOAT_OVERFLOW	超出浮點類型的最大正數。
STATUS_FLOAT_UNDERFLOW	超出浮點類型的最小負數範圍。
STATUS_FLOATING_RESEVERED_OPERAND	使用保留的浮點格式（使用無效的格式）。
STATUS_ILLEGAL_INSTRUCTION	嘗試執行處理器未定義的指令碼。
STATUS_PRIVILEGED_INSTRUCTION	執行目前電腦模式中不允許的指令。
STATUS_INTEGER_DIVIDE_BY_ZERO	除以 0 的整數類型。
STATUS_INTEGER_OVERFLOW	嘗試進行超過整數範圍的作業。
STATUS_SINGLE_STEP	在單一步驟模式中執行一個指令，只能由偵錯工具使用。

上表中列出的許多例外狀況主要是要由偵錯工具、作業系統，或其他低階程式碼處理。除了整數和浮點數的錯誤之外，您的程式碼不應該處理這些錯誤。因此，您通常應該使用例外狀況處理篩選條件來忽略例外狀況（運算結果為 0）。否則，您可以透過適當的回應來防止低階機制執行。不過，您可以藉由[撰寫終止處理常式](#)，針對這些低層級錯誤的潛在影響採取適當的預防措施。

## 另請參閱

[撰寫例外狀況處理常式](#)

[結構化例外狀況處理 \(C/C++\)](#)

# 例外狀況處理常式的限制

2020/11/2 • [Edit Online](#)

在程式碼中使用例外狀況處理常式的主要限制是，您不能使用 `goto` 語句跳入 `__try` 語句區塊。您必須改為透過一般控制流程進入陳述式區塊。您可以跳出 `__try` 語句區塊，也可以依您的選擇來嵌套例外狀況處理常式。

## 請參閱

[撰寫例外狀況處理常式](#)

[Structured Exception Handling \(C/C++\)](#)

# 撰寫終止處理常式

2020/3/25 • [Edit Online](#)

不同於例外狀況處理常式，無論程式碼保護區塊是否正常終止，都一定會執行終止處理常式。終止處理常式的唯一用途應該是確保資源（例如記憶體、控制代碼和檔案）適當地關閉，而不管程式碼區段的執行如何完成。

終止處理常式會使用 try-finally 陳述式。

## 您還想知道關於哪些方面的詳細資訊？

- [Try-finally 語句](#)
- [清除資源](#)
- [例外狀況處理中的動作時機](#)
- [終止處理常式的限制](#)

## 另請參閱

[結構化例外狀況處理 \(C/C++\)](#)

# try-finally 陳述式

2020/11/2 • [Edit Online](#)

try-finally 語句是 Microsoft 特定的擴充功能，可支援 C 和 c++ 語言的結構化例外狀況處理。

## 語法

下列語法描述 try-finally 語句：

```
// . .
__try {
    // guarded code
}
__finally {
    // termination code
}
// . . .
```

## 文法

try-finally-statement :

  \_\_try compound-statement \_\_finally compound-statement

try-finally 語句是 C 和 c++ 語言的 Microsoft 擴充功能，可讓目標應用程式在執行程式碼區塊中斷時，保證執行清除程式碼。清除包含如取消配置記憶體、關閉檔案和釋放檔案控制代碼等工作。try-finally 陳述式對於有多個地方要檢查可能會導致常式過早傳回的錯誤時會特別有用。

如需相關資訊和程式碼範例，請參閱 [try-except 語句](#)。如需一般結構化例外狀況處理的詳細資訊，請參閱 [結構化例外狀況處理](#)。如需使用 c++/CLI 處理 managed 應用程式例外狀況的詳細資訊，請參閱底下的[例外狀況處理 /clr](#)。

### NOTE

結構化例外狀況處理可搭配 Win32 處理 C 和 C++ 原始程式檔。不過，它不是專為 C++ 所設計。使用 C++ 例外狀況處理可確保您的程式碼更具可移植性。此外，C++ 例外狀況處理更有彈性，因為它可以處理任何類型的例外狀況。若是 c++ 程式，建議您使用 c++ 例外狀況處理機制，([try](#)、[catch](#) 和 [throw](#) 語句)。

子句後面的複合陳述式 [\\_\\_try](#) 是受保護的區段。子句後面的複合陳述式 [\\_\\_finally](#) 是終止處理常式。處理常式會指定一組當保護區段結束時執行的動作、是否因例外狀況 (異常終止) 而結束受保護的區段，或依標準 (正常終止)。

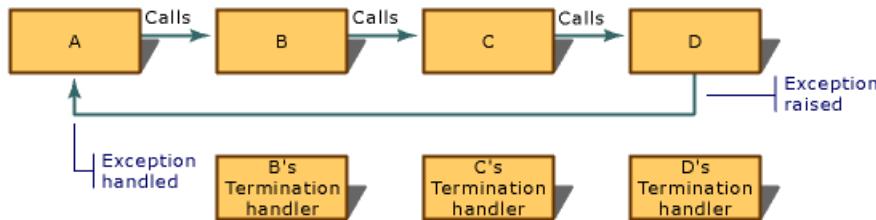
藉 [\\_\\_try](#) 由簡單的循序執行，將控制項傳遞至語句 (貫穿)。當控制項進入時 [\\_\\_try](#)，其相關聯的處理常式會變成作用中狀態。如果控制流程到達 try 區塊的結尾，執行程序如下所示：

1. 已叫用終止處理常式。
2. 終止處理常式完成時，會在語句之後繼續執行 [\\_\\_finally](#)。不過，受保護的區段會 (例如，透過 [goto](#) 超出防護主體或 [return](#) 語句) 來執行，終止處理常式會在控制流程移出保護區段 之前 執行。

[\\_\\_finally](#) 語句不會封鎖搜尋適當的例外狀況處理常式。

如果區塊中發生例外狀況 `_try`，作業系統必須找到例外狀況的處理常式，否則程式將會失敗。如果找到處理程式，則會執行任何和所有 `_finally` 區塊，並且在處理常式中繼續執行。

例如，假設有一系列的函式呼叫連結了函式 A 與函式 D，如下圖所示。每個函式都具有一個終止處理常式。如果例外狀況在函式 D 中引發，並在函式 A 中處理，則會在系統回溯堆疊時，依此順序呼叫終止處理常式：D、C、B。



終止處理常式的執行順序

#### NOTE

Try-catch 的行為不同于支援使用的其他語言 `finally`，例如 c #。單一 `_try` 可以有(但不是兩者)和兩者 `_try` `_finally` `_except`。如果要同時使用兩個，外層的 try-except 陳述式必須以引號括住內部 try-finally 陳述式。指定的規則在每個區塊執行時也不同。

為了與舊版相容，`_try` 和 `_finally` `_leave` 都是、和的同義字，`_try` `_finally` `_leave` 除非指定了編譯器選項 `/za` (停用語言延伸模組)。

## `_leave` 關鍵字

`_leave` 關鍵字只有在語句的保護區段內才有效 `try-finally`，而其效果是跳到受保護區段的結尾。然後從終止處理常式中的第一個陳述式繼續執行。

`goto` 語句也可以跳出受保護的區段，但它會降低效能，因為它會叫用堆疊回溯。`_leave` 語句會更有效率，因為它不會導致堆疊回溯。

## 異常終止

`try-finally` 使用 `longjmp` 執行時間函式結束語句會被視為異常終止。跳到陳述並不合法 `_try`，但要跳到其中一項是合法的。`_finally` 在起飛點之間作用的所有語句(區塊)的正常終止 `_try`，以及處理例外狀況的目的地(`_except`)必須執行。它稱為「本機回溯」。

如果 `_try` 區塊因為任何原因而提前終止，包括跳出區塊，系統會在回溯堆疊的過程中執行相關聯的 `_finally` 區塊。在這種情況下，`AbnormalTermination` `true` 如果從區塊內呼叫，函數會傳回，`_finally` 否則會傳回 `false`。

如果進程在執行語句的過程中終止，則不會呼叫終止處理常式 `try-finally`。

結束 Microsoft 專用

## 請參閱

[撰寫終止處理常式](#)

[Structured Exception Handling \(C/C++\)](#)

[關鍵字](#)

[終止-處理常式語法](#)

# 清除資源

2020/11/2 • • [Edit Online](#)

在終止處理常式執行期間，您可能不知道在呼叫終止處理常式之前，已取得了哪些資源。在 `__try` 取得所有資源之前，語句區塊可能會中斷，因此不會開啟所有資源。

為了安全起見，您應該先查看哪些資源已開啟，然後再繼續進行終止處理清除。建議的程序是：

1. 將控制代碼初始化為 NULL。
2. 在 `__try` 語句區塊中，取得資源。當取得資源時，控制碼會設定為正值。
3. 在 `__finally` 語句區塊中，釋放相對應的控制碼或旗標變數為非零或非 Null 的每個資源。

## 範例

例如，下列程式碼會使用終止處理常式來關閉三個檔案並釋放記憶體區塊。這些資源是在 `__try` 語句區塊中取得。清除資源之前，程式碼會先檢查是否已取得資源。

```
// 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();
}
```

## 請參閱

[撰寫終止處理常式](#)

## Structured Exception Handling (C/C++)

# 例外狀況處理的時機：摘要

2020/11/2 • [Edit Online](#)

無論語句區塊的結束方式為何，都會執行終止處理常式 `__try`。原因包括跳過 `__try` 區塊、將 `longjmp` 控制權移出區塊的語句，以及因例外狀況處理而回溯堆疊。

## NOTE

Microsoft C++ 編譯器支援兩種形式的 `setjmp` 和 `longjmp` 語句。快速版本會略過終止處理，但是會更有效率。若要使用此版本，請包含檔案 `<setjmp.h>`。另一個版本支援終止處理，如先前段落中所述。若要使用此版本，請包含檔案 `<setjmpex.h>`。快速版本的效能提升取決於硬體組態。

作業系統會先依適當的順序執行所有終止處理常式，再執行其他程式碼，包括例外狀況處理常式的主體。

當導致中斷的原因是來自例外狀況時，系統必須先執行一個或多個例外狀況處理常式的篩選條件部分，再決定要終止哪個部分。事件的順序為：

1. 引發例外狀況。
2. 系統會查看作用中例外狀況處理常式的階層，並執行具有最高優先順序的處理常式篩選。這就是最近安裝的例外狀況處理常式，以及最深層的嵌套（由區塊和函式呼叫）。
3. 如果此篩選準則通過控制（會傳回 0），進程會繼續執行，直到找到未通過控制權的篩選準則為止。
4. 如果此篩選傳回 -1，則會在引發例外狀況的情況下繼續執行，而且不會進行任何終止。
5. 如果篩選條件傳回 1，則會發生下列事件：
  - 系統回溯堆疊：它會清除引發例外狀況的所有堆疊框架，以及包含例外狀況處理常式的堆疊框架。
  - 回溯堆疊時，會執行堆疊上的每個終止處理常式。
  - 例外狀況處理常式本身會執行。
  - 控制權會傳遞給這個例外狀況處理常式結尾後方的程式碼。

## 請參閱

[撰寫終止處理常式](#)

[Structured Exception Handling \(C/C++\)](#)

# 終止處理常式的限制

2020/11/2 • • [Edit Online](#)

您無法使用 `goto` 語句跳到 `__try` 語句區塊或 `__finally` 語句區塊。您必須改為透過一般控制流程進入陳述式區塊。不過 (您可以跳出 `__try` 語句區塊)。此外，您也無法在區塊內將例外狀況處理常式或終止處理常式嵌入。

`__finally`

終止處理常式中允許的一些程式碼會產生可疑的結果，因此您應該謹慎使用它們。其中一個是 `goto` 跳出語句區塊的語句 `__finally`。如果在正常終止的過程中執行區塊，就不會發生任何異常狀況。但是，如果系統回溯堆疊，就會停止。然後，目前的函式會取得控制項，如同沒有異常終止。

`return` 語句區塊內的語句大致上有 `__finally` 相同的情況。控制權會返回包含終止處理常式之函式的立即呼叫端。如果系統回溯堆疊，就會停止此進程。然後，程式會繼續執行，就像未引發任何例外狀況一樣。

## 請參閱

[撰寫終止處理常式](#)

[Structured Exception Handling \(C/C++\)](#)

# 在執行緒之間傳輸例外狀況

2020/11/2 • [Edit Online](#)

Microsoft C++ 編譯器 (MSVC) 支援將例外狀況從某個執行緒 傳輸 到另一個執行緒。傳輸例外狀況可以讓您在某個執行緒攔截例外狀況，再使該例外狀況看似在另一個執行緒中擲回。舉例來說，您可以使用此功能撰寫多執行緒應用程式，其中由主執行緒處理次要執行緒所擲回的所有例外狀況。傳輸例外狀況對於建立平行程式設計程式庫或系統的開發人員最有用。為了執行傳輸例外狀況，MSVC 會提供 `exception_ptr` 型別和 `current_exception`、`rethrow_exception` 和 `make_exception_ptr` 函數。

## 語法

```
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;
}
```

### 參數

#### 未知

用於實作 `exception_ptr` 類型的未指定內部類別。

#### P

參考例外狀況的 `exception_ptr` 物件。

#### Pci-e

代表例外狀況的類別。

#### pci-e

參數 `E` 類別的執行個體。

## 傳回值

`current_exception` 函式會將 `exception_ptr` 物件傳回，此物件則參考目前進行中的例外狀況。如果沒有正在進行中的例外狀況，該函式會傳回與所有例外狀況皆不具關聯性的 `exception_ptr` 物件。

`make_exception_ptr` 函數 `exception_ptr` 會傳回物件，該物件會參考 `e` 參數所指定的例外狀況。

## 備註

### 狀況

假設您需要建立一個能縮放以處理不同工作量的應用程式，為達成此目標，您可以在一開始設計一個多執行緒的應用程式，而其主執行緒可以視工作執行所需建立任何數量的次要執行緒。次要執行緒能協助主執行緒管理資源、平衡負載和提升輸送量。透過分配工作的方式，多執行緒應用程式的運行效率可以優於單一執行緒應用程式。

不過，若次要執行緒擲回例外狀況，需由主要執行緒處理。這是因為無論次要執行緒的數量多寡，您的應用程式最好都能夠以一致、統一的方式處理例外狀況。

## 解決方案

為處理以上情況，C++ Standard 支援在執行緒之間傳輸例外狀況。如果次要執行緒擲回例外狀況，則該例外狀況會成為目前的例外狀況。藉由與真實世界相似，目前的例外狀況是在飛行中。從目前例外狀況被擲回那一刻起，一直到攔截到該例外狀況的例外狀況處理常式傳回為止，目前例外狀況都在執行中。

次要執行緒可以在區塊中攔截目前的例外狀況 `catch`，然後呼叫函 `current_exception` 式將例外狀況儲存在物件中 `exception_ptr`。`exception_ptr` 物件必須可供次要執行緒與主執行緒使用。舉例來說，`exception_ptr` 物件可以是由 Mutex 控制存取的全域變數。「傳輸例外狀況」一詞表示一個執行緒中的例外狀況可以轉換成另一個執行緒可以存取的表單。

接下來，主執行緒會呼叫 `rethrow_exception` 函式，而該函式再擷取並從 `exception_ptr` 物件擲回例外狀況。例外狀況被擲回時會變成主執行緒中目前的例外狀況。也就是說，該例外狀況看起來就像源自於主執行緒。

最後，主執行緒可以在區塊中攔截目前的例外狀況 `catch`，然後處理它，或將它擲回較高層級的例外狀況處理常式。或者，主執行緒可以忽略該例外狀況並允許處理序結束。

大部分的應用程式不需要在執行緒之間傳輸例外狀況。不過，由於系統可以區分次要執行緒、處理器或核心的工作，因此此功能對於平行計算系統非常有用。在平行計算環境下，一個專屬的執行緒可以處理次要執行緒擲回的所有例外狀況，而且可以對任何應用程式呈現一致的例外狀況處理模型。

如需關於 C++ Standard 委員會提案的詳細資訊，請上網搜尋文件編號 N2179，標題為「在執行緒之間傳輸例外狀況的語言支援」。

### 例外狀況處理模型和編譯器選項

應用程式的例外狀況處理模型決定該應用程式是否可以攔截和傳輸例外狀況。Visual C++ 支援三種可以處理 C++ 例外狀況、結構化例外處理 (SEH) 例外狀況，以及通用語言執行平台 (CLR) 例外狀況的模型。您可以使用 `/EH` 和 `/clr` 編譯器選項，來指定應用程式的例外狀況處理模型。

只有以下的編譯器選項和程式設計陳述式組合可以傳輸例外狀況，其他組合無法攔截例外狀況或可以攔截但無法傳輸例外狀況。

- `/EHa` 編譯器選項和 `catch` 語句可以傳輸 SEH 和 C++ 例外狀況。
- `/EHa`、`/ehs` 和 `/ehsc` 編譯器選項及 `catch` 語句可以傳輸 C++ 例外狀況。
- `/Clr` 編譯器選項和 `catch` 語句可以傳輸 C++ 例外狀況。`/Clr` 編譯器選項表示 `/eha` 選項的規格。請注意，編譯器不支援傳輸 Managed 例外狀況，這是因為從 `System.Exception` 類別衍生的 managed 例外狀況已經是物件，您可以使用 common language runtime 的功能線上程之間移動它們。

#### IMPORTANT

建議您指定 `/ehsc` 編譯器選項，並只攔截 C++ 例外狀況。如果您使用 `/eha` 或 `/clr` 編譯器選項，以及 `catch` 具有省略號 例外 狀況宣告的語句 `()`，您就會向自己公開安全性威脅 `catch(...)`。您可能想要使用 `catch` 語句來捕捉一些特定的例外狀況。不過，`catch(...)` 陳述式會擷取所有 C++ 和 SEH 例外狀況，其中包括嚴重的非預期例外狀況。如果您忽略非預期的例外狀況或處理不當，惡意程式碼可能會利用此機會破壞程式的安全性。

## 使用方式

下列各節描述如何使用類型、和函數來傳輸例外狀況 `exception_ptr`、`current_exception`、`rethrow_exception`、`make_exception_ptr`。

### exception\_ptr 類型

利用 `exception_ptr` 物件參考目前的例外狀況或使用者指定的例外狀況執行個體。在 Microsoft 實作中，例外狀況是以 `EXCEPTION_RECORD` 結構表示。每個 `exception_ptr` 物件均包含一個例外狀況參考欄位，指向代表該例外狀況的 `EXCEPTION_RECORD` 結構複本。

當您宣告 `exception_ptr` 變數時，此變數尚未關聯任何例外狀況。也就是說，其例外狀況參考欄位為 `NULL`。這樣的 `exception_ptr` 物件稱為 *null exception\_ptr*。

利用 `current_exception` 或 `make_exception_ptr` 函式將例外狀況指派給 `exception_ptr` 物件。當您將例外狀況指派給 `exception_ptr` 變數時，此變數的例外狀況參考欄位會指向該例外狀況的複本。如果記憶體空間不足，無法複製該例外狀況，則例外狀況參考欄位會指向 `std::bad_alloc` 例外狀況的複本。如果或函式 `current_exception` `make_exception_ptr` 因為任何其他原因而無法複製例外狀況，此函式會呼叫 `terminate` 函數來 [結束](#) 目前的進程。

`exception_ptr` 物件本身並不是指標（儘管其名稱如此）。它不會遵守指標語義，也不能搭配指標成員存取 (`->`) 或間接 (`*`) 運算子使用。`exception_ptr` 物件沒有公用資料成員或成員函式。

## 比較

您可以使用等於 (`==`) 和不等於 (`!=`) 運算子比較兩個 `exception_ptr` 物件。運算子不會比較代表例外狀況之 `EXCEPTION_RECORD` 結構的二進位值（位元模式）。相反地，運算子會比較 `exception_ptr` 物件的例外狀況參考欄位位址。因此，Null `exception_ptr` 和 Null 值的比較結果是相等。

## current\_exception 函式

`current_exception` 在區塊中呼叫函數 `catch`。如果發生例外狀況，而且 `catch` 區塊可以攔截例外狀況，則函式會傳回 `current_exception` 參考例外狀況的 `exception_ptr` 物件。否則，函式會傳回 Null `exception_ptr` 物件。

### 詳細資料

`current_exception` 無論 `catch` 語句是否指定 [例外](#) 狀況宣告語句，函式都會捕獲進行中的例外狀況。

如果您未重新擲回例外狀況，則會在區塊結尾呼叫目前例外狀況的函式 `catch`。不過，即使在解構函式中呼叫 `current_exception` 函式，該函式仍會傳回參考目前例外狀況的 `exception_ptr` 物件。

`current_exception` 函式的後續呼叫會傳回參考目前例外狀況不同複本的 `exception_ptr` 物件。因此，物件比較結果會是不相等，因為兩者參考不同的複本（即使複本的二進位值相同也一樣）。

### SEH 例外狀況

如果您使用 `/eha` 編譯器選項，就可以在 C++ 區塊中攔截 SEH 例外狀況 `catch`。`current_exception` 函式會傳回參考 SEH 例外狀況的 `exception_ptr` 物件。`rethrow_exception` 如果您使用 `thetransported` `exception_ptr` 物件做為其引數來呼叫，則函式會擲回 SEH 例外狀況。

`current_exception` `exception_ptr` 如果您在 SEH `__finally` 終止處理常式、`__except` 例外狀況處理常式或篩選運算式中呼叫它，此函數會傳回 null `__except`。

傳輸例外狀況不支援巢狀例外狀況。處理例外狀況時，如果再擲出另一個例外狀況，則會發生巢狀例外狀況。如果您攔截巢狀例外狀況，`EXCEPTION_RECORD.ExceptionRecord` 資料成員會指向描述關聯例外狀況的 `EXCEPTION_RECORD` 結構鏈結。`current_exception` 函式不支援巢狀例外狀況，因為該函式會傳回已清空 `exception_ptr` 資料成員的 `ExceptionRecord` 物件。

如果您攔截 SEH 例外狀況，則必須管理 `EXCEPTION_RECORD.ExceptionInformation` 資料成員陣列中任何指標所參考的記憶體。您必須保證對應 `exception_ptr` 物件存留期的記憶體是有效的，並在刪除 `exception_ptr` 物件時釋放其記憶體。

您可以同時使用結構化例外狀況 (SE) 轉譯器函式和傳輸例外狀況功能。如果 SEH 例外狀況轉譯為 C++ 例外狀況，`current_exception` 函式會傳回參考轉譯的例外狀況而非原始的 SEH 例外狀況的 `exception_ptr`。`rethrow_exception` 函式後續會擲回轉譯的例外狀況，而非原始的例外狀況。如需有關 SE translator 函式的詳細資訊，請參閱 [\\_set\\_se\\_translator](#)。

## rethrow\_exception 函式

將攔截到的例外狀況儲存在 `exception_ptr` 物件之後，主執行緒即可處理物件。在主執行緒中呼叫

`rethrow_exception` 函式，並使用 `exception_ptr` 物件做為其引數。`rethrow_exception` 函式會從 `exception_ptr` 物件擷取例外狀況，然後在主執行緒的內容中擲回該例外狀況。如果函式的 *p* 參數 `rethrow_exception` 為 null，則函式會擲回 `exception_ptr std::bad_exception`。

擷取到的例外狀況現在會變成主執行緒中的目前例外狀況，因此，您可以按照處理其他任何例外狀況的方式進行處理。如果您攔截例外狀況，可以立即處理它，或使用 `throw` 語句將它傳送至較高層級的例外狀況處理常式。否則，請勿執行任何動作，並讓預設系統例外狀況處理常式終止處理序。

## make\_exception\_ptr 函式

`make_exception_ptr` 函式會以類別的執行個體做為其引數，並傳回參考該執行個體的 `exception_ptr`。雖然任何類別物件都可以是 `make_exception_ptr` 函式的引數，但一般會指定 `exception` 類別物件作為其引數。

呼叫函式相當於擲 `make_exception_ptr` 回 C++ 例外狀況、在 `catch` 區塊中攔截它，然後呼叫函式 `current_exception` 式以傳回 `exception_ptr` 參考例外狀況的物件。Microsoft 實作 `make_exception_ptr` 函式比在擲回例外狀況之後攔截來得更有效率。

一般來說，應用程式通常不需要使用 `make_exception_ptr` 函式，所以我們不建議使用此功能。

## 範例

以下範例會在執行緒之間傳輸標準 C++ 例外狀況和自訂的 C++ 例外狀況。

```
// transport_exception.cpp
// compile with: /EHsc /MD
#include <windows.h>
#include <stdio.h>
#include <exception>
#include <stdexcept>

using namespace std;

// Define thread-specific information.
#define THREADCOUNT 2
exception_ptr aException[THREADCOUNT];
int aArg[THREADCOUNT];

DWORD WINAPI ThrowExceptions( LPVOID );
// Specify a user-defined, custom exception.
// As a best practice, derive your exception
// directly or indirectly from std::exception.
class myException : public std::exception {
};

int main()
{
    HANDLE aThread[THREADCOUNT];
    DWORD ThreadID;

    // Create secondary threads.
    for( int i=0; i < THREADCOUNT; i++ )
    {
        aArg[i] = i;
        aThread[i] = CreateThread(
            NULL,           // Default security attributes.
            0,              // Default stack size.
            (LPTHREAD_START_ROUTINE) ThrowExceptions,
            (LPVOID) &aArg[i], // Thread function argument.
            0,              // Default creation flags.
            &ThreadID); // Receives thread identifier.
        if( aThread[i] == NULL )
        {
            printf("CreateThread error: %d\n", GetLastError());
        }
    }
}
```

```

        return -1;
    }

// Wait for all threads to terminate.
WaitForMultipleObjects(THREADCOUNT, aThread, TRUE, INFINITE);
// Close thread handles.
for( int i=0; i < THREADCOUNT; i++ ) {
    CloseHandle(aThread[i]);
}

// Rethrow and catch the transported exceptions.
for ( int i = 0; i < THREADCOUNT; i++ ) {
    try {
        if (aException[i] == NULL) {
            printf("exception_ptr %d: No exception was transported.\n", i);
        }
        else {
            rethrow_exception( aException[i] );
        }
    }
    catch( const invalid_argument & ) {
        printf("exception_ptr %d: Caught an invalid_argument exception.\n", i);
    }
    catch( const myException & ) {
        printf("exception_ptr %d: Caught a myException exception.\n", i);
    }
}
}

// Each thread throws an exception depending on its thread
// function argument, and then ends.
DWORD WINAPI ThrowExceptions( LPVOID lpParam )
{
    int x = *((int*)lpParam);
    if (x == 0) {
        try {
            // Standard C++ exception.
            // This example explicitly throws invalid_argument exception.
            // In practice, your application performs an operation that
            // implicitly throws an exception.
            throw invalid_argument("A C++ exception.");
        }
        catch ( const invalid_argument & ) {
            aException[x] = current_exception();
        }
    }
    else {
        // User-defined exception.
        aException[x] = make_exception_ptr( myException() );
    }
    return TRUE;
}

```

```

exception_ptr 0: Caught an invalid_argument exception.
exception_ptr 1: Caught a myException exception.

```

## 規格需求

標頭: <exception>

## 另請參閱

### 例外狀況處理

/EH (例外狀況處理模型)

/clr (Common Language Runtime 編譯)

# 判斷提示和使用者提供的訊息 (C++)

2020/11/2 • [Edit Online](#)

C++ 語言支援三種錯誤處理機制，可協助您對應用程式進行調試：`#error`指示詞、`static_assert`關鍵字，以及`assert`宏、`_assert`、`_wassert`宏。這三種機制都會發出錯誤訊息，其中兩種還會測試軟體判斷提示。軟體判斷提示會指定您希望在程式中的某個特定點為 true 的條件。如果編譯時期判斷提示失敗，編譯器會發出診斷訊息和編譯錯誤。如果執行階段判斷提示失敗，作業系統會發出診斷訊息並關閉應用程式。

## 備註

應用程式的存留期包括前置處理、編譯和執行階段。每種錯誤處理機制均會存取其中任一階段可用的偵錯資訊。若要有效偵錯，請選取能夠針對該階段提供適當資訊的機制：

- `#Error`指示詞在前置處理時間生效。它會無條件發出使用者指定的訊息，並且會使編譯失敗並產生錯誤。訊息可以包含由前置處理器指示詞操作但不會評估任何結果運算式的文字。
  - `Static_assert`宣告會在編譯時期生效。它會測試由使用者指定的整數運算式代表的軟體判斷提示，此判斷提示可以轉換為布林值。如果運算式判斷值為零 (false)，編譯器會發出使用者指定的訊息，且編譯會失敗並產生錯誤。
- 宣告 `static_assert` 特別適合用於偵錯工具範本，因為樣板引數可以包含在使用者指定的運算式中。
- `Assert 宏 _assert, _wassert`宏會在執行時間生效。它會評估使用者指定的運算式，如果結果為零，系統會發出診斷訊息並關閉應用程式。許多其他宏(例如`_ASSERT`和`_ASSERTE`)與此宏類似，但會發出不同的系統定義或使用者定義的診斷訊息。

## 另請參閱

[#error 指示詞 \(C/c++\)](#)  
[assert 宏、\\_assert、\\_wassert](#)  
[\\_ASSERT、\\_ASSERTE、\\_ASSERT\\_EXPR 宏](#)  
[static\\_assert](#)  
[\\_STATIC\\_ASSERT 宏](#)  
[範本](#)

# static\_assert

2020/11/2 • [Edit Online](#)

在編譯時期測試軟體判斷提示。如果指定的常數運算式為 `false`，則編譯器會顯示指定的訊息(如果有提供的話)，而且編譯會失敗並出現錯誤 C2338；否則，宣告沒有任何作用。

## 語法

```
static_assert( constant-expression, string-literal );  
  
static_assert( constant-expression ); // C++17 (Visual Studio 2017 and later)
```

### 參數

#### 常數運算式

可以轉換為布林值的整數常數運算式。如果評估的運算式為零 (`false`)，則會顯示 字串常值參數，而且編譯會失敗並出現錯誤。如果運算式非零 (`true`)，則宣告 `static_assert` 沒有任何作用。

#### 字串常值

如果 常數運算式 參數為零，則會顯示訊息。訊息是編譯器 [基底字元集](#) 中的字元字串；也就是說，不是 [多位元組](#) 或 [寬字元](#)。

## 備註

宣告的 常數運算式 參數 `static_assert` 表示 軟體判斷提示。軟體判斷提示會指定您希望在程式中的某個特定點為 `true` 的條件。如果條件為 `true`，則宣告 `static_assert` 沒有任何作用。如果條件為 `false`，則判斷提示會失敗，編譯器會以 字串常值參數 顯示訊息，且編譯會失敗並出現錯誤。在 Visual Studio 2017 和更新版本中，字串常值參數是選擇性的。

`static_assert` 宣告會在編譯時期測試軟體判斷提示。相反地，判斷提示 [宏和 \\_assert 和 \\_wassert 函數](#) 會在執行時間測試軟體判斷提示，並在空間或時間產生執行時間成本。宣告 `static_assert` 特別適用於偵錯工具，因為樣板引數可以包含在 常數運算式 參數中。

當遇到宣告時，編譯器會檢查宣告是否 `static_assert` 有語法錯誤。如果運算式不相依於樣板參數，則編譯器會立即評估 常數運算式 參數。否則，編譯器會在範本具現化時評估 常數運算式 參數。因此，編譯器可能會在遇到宣告時發出診斷資訊一次，並且在樣板具現化時再次發出訊息。

您可以 `static_assert` 在命名空間、類別或區塊範圍中使用關鍵字。( `static_assert` 關鍵字在技術上是宣告，雖然它不會在您的程式中引入新的名稱，因為它可以在命名空間範圍中使用。)

### `static_assert` 具有命名空間範圍的描述

在下列範例中，宣告 `static_assert` 具有命名空間範圍。由於編譯器已知 `void *` 類型的大小，因此會立即計算運算式的值。

### 範例： `static_assert` 命名空間範圍

```
static_assert(sizeof(void *) == 4, "64-bit code generation is not supported.");
```

## static\_assert 具有類別範圍的描述

在下列範例中，宣告 `static_assert` 具有類別範圍。`static_assert` 會確認範本參數是純舊的資料(POD)類型。編譯器會檢查宣告的宣告 `static_assert`，但在中具現化類別樣板之前，不會評估 常數運算式參數 `basic_string` `main()`。

### 範例： static\_assert 使用類別範圍

```
#include <type_traits>
#include <iostream>
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;
}
```

## static\_assert 使用區塊範圍的描述

在下列範例中，宣告 `static_assert` 具有區塊範圍。`static_assert` 會驗證 VMPage 結構的大小是否等於系統的虛擬記憶體 pagesize。

### 範例： static\_assert 在區塊範圍

```
#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.");
        // ...
    }
    // ...
};
```

## 另請參閱

[判斷提示和使用者提供的訊息 \(C++\)](#)

[#error 指示詞 \(C/c++\)](#)

[assert 宏、\\_assert、\\_wassert](#)

[範本](#)

[ASCII 字元集](#)

## 宣告和定義

# C++ 中的模組概觀

2020/11/2 • [Edit Online](#)

C++ 20 引進了模組，這是元件化 C++ 程式庫和程式的現代化解決方案。模組是一組原始程式碼檔案，這些檔案會與匯入它們的轉譯單位分開編譯。模組會消除或大幅減少與使用標頭檔相關的許多問題，也可能會減少編譯時間。不會顯示在模組中宣告的宏、預處理器指示詞和非匯出的名稱，因此不會影響匯入模組之轉譯單元的編譯。您可以依照任何順序匯入模組，而不需要考慮宏重新定義。匯入轉譯單位中的宣告不會參與已匯入之模組中的多載解析或名稱查詢。在模組編譯一次之後，結果會儲存在二進位檔案中，以描述所有匯出的類型、函數和範本。該檔案的處理速度快於標頭檔，而且編譯器可以在每次將模組匯入專案的位置重複使用。

模組可以與標頭檔並存使用。C++ 原始檔可匯入模組，也可以 #include 標頭檔案。在某些情況下，您可以將標頭檔匯入為模組，而不是預處理器 #included 的以程式方式提供。我們建議新的專案盡可能使用模組，而不是標頭檔。對於開發中較大的現有專案，我們建議您嘗試將舊版標頭轉換成模組，以瞭解您在編譯時間是否有意義的縮減。

## 在 Microsoft C++ 編譯器中啟用模組

從 Visual Studio 2019 16.2 版，模組不會在 Microsoft C++ 編譯器中完全執行。您可以使用模組功能來建立單一資料分割模組，並匯入 Microsoft 所提供的標準程式庫模組。若要啟用對模組的支援，請使用 [experimental::module](#) 和 [std::c++\\_最新版本](#) 進行編譯。在 Visual Studio 專案中，以滑鼠右鍵按一下方案總管中的專案節點，然後選擇 [屬性]。將 [設定] 下拉式選為 [所有設定]，然後選擇 [設定] [屬性] [> c/c++ > 語言] [> 啟用 C++ 模組]

模組和使用它的程式碼必須使用相同的編譯器選項進行編譯。

## 使用 C++ 標準程式庫作為模組

雖然不是由 C++ 20 標準所指定，但 Microsoft 會啟用將 C++ 標準程式庫的實作為模組匯入。藉由匯入 C++ 標準程式庫做為模組，而不是透過標頭檔 #including 它，您可能會根據專案的大小來加速編譯時間。程式庫會元件化至下列模組：

- 標準 RegEx 提供標頭的內容 <regex>
- std::filesystem 提供標頭的內容 <filesystem>
- std 會提供標頭的內容 <memory>
- std::thread 提供標頭 <atomic>、<condition\_variable>、<future>、<mutex>、<shared\_mutex> 和的內容 <thread>
- std 提供 C++ 標準程式庫中的其他專案

若要使用這些模組，只要將匯入宣告新增至原始程式碼檔案的頂端即可。例如：

```
import std.core;
import std.regex;
```

若要使用 Microsoft 標準程式庫模組，請使用 [/ehsc](#) 和 [/md](#) 選項來編譯您的程式。

## 基本範例

下列範例顯示名為 Foo. ixx 之原始檔中的簡單模組定義。Visual Studio 中的模組介面檔案需要 ixx 副檔名。在此範例中，介面檔案包含函式定義和宣告。不過，定義也可以放在一或多個不同的檔案中（如稍後的範例所示）。

Export 模組 Foo 語句指出此檔案是稱為之模組的主要介面 `Foo`。`export` 上的修飾詞 `f()` 表示當 `Foo` 另一個

程式或模組匯入時，將會顯示此函式。請注意，模組會參考命名空間 `Bar`。

```
export module Foo;

#define ANSWER 42

namespace Bar
{
    int f_internal() {
        return ANSWER;
    }

    export int f() {
        return f_internal();
    }
}
```

File `myprogram.exe`會使用`import`宣告來存取所匯出的名稱 `Foo`。請注意，`Bar` 此處會顯示名稱，但不是所有的成員。另請注意，`ANSWER` 不會顯示宏。

```
import Foo;
import std.core;

using namespace std;

int main()
{
    cout << "The result of f() is " << Bar::f() << endl; // 42
    // int i = Bar::f_internal(); // C2039
    // int j = ANSWER; //C2065
}
```

匯入宣告只能出現在全域範圍中。

## 執行模組

您可以使用單一介面檔案(.ixx)來建立模組，該檔案會匯出名稱並包含所有函式和類型的實作為。您也可以將此部署放在一或多個不同的執行檔中，類似於使用 .h 和 .cpp 檔案的方式。`export` 關鍵字只在介面檔案中使用。執行檔案可以匯入另一個模組，但不能 `export` 有任何名稱。可以使用任何副檔名來命名執行檔。系統會將介面檔案和其後置的一組執行檔視為一種特殊的轉譯單位，稱為模組單位。在任何執行檔中宣告的名稱會自動顯示在相同模組單元中的所有其他檔案中。

對於較大的模組，您可以將模組分割為多個模組單元，稱為「分割區」。每個分割區都是由一個或多個執行檔支援的介面檔所組成。(從 Visual Studio 2019 版本16.2，尚未完全執行資料分割)。

## 模組、命名空間和引數相依的查閱

模組中的命名空間規則與任何其他程式碼中的相同。如果匯出命名空間內的宣告，則也會隱含地匯出封入的命名空間(不包括非匯出的成員)。如果明確匯出命名空間，則會匯出該命名空間定義內的所有宣告。

當匯入轉譯單位中的多載解析執行與引數相依的查閱時，編譯器會考慮在相同轉譯單位(包括模組介面)中宣告的函式，如同函數的引數類型的定義。

### 模組磁碟分割

#### NOTE

這一節是針對完整性而提供的。磁碟分割尚未在 Microsoft C++ 編譯器中執行。

模組可以元件化成分割區，每個都包含一個介面檔案和零或多個執行檔。模組磁碟分割類似于模組，不同之處在于它會共用整個模組中所有宣告的擁有權。分割區介面檔案所匯出的所有名稱都是由主要介面檔案匯入和重新匯出。資料分割的名稱必須以模組名稱開頭，後面接著冒號。在整個模組中，都可以看到任何分割區中的宣告。不需要採取任何特殊預防措施來避免發生單一定義規則(ODR)錯誤。您可以在一個資料分割中宣告名稱(函式、類別等)，並在另一個資料分割中定義它。分割區的執行檔案的開頭如下：

```
module Foo:part1
```

而分割區介面檔案的開頭就像這樣：

```
export module Foo:part1
```

若要存取另一個分割區中的宣告，分割區必須匯入，但它只能使用分割區名稱，而不是模組名稱：

```
module Foo:part2;
import :part1;
```

主要介面單位必須匯入並重新匯出所有模組的介面分割區檔案，如下所示：

```
export import :part1
export import :part2
...
```

主要介面單位可以匯入資料分割執行檔案，但無法匯出它們，因為這些檔案不允許匯出任何名稱。這可讓模組保有模組內部的執行詳細資料。

## 模組和標頭檔

您可以藉由將指示詞放在模組宣告之前，將標頭檔包含在模組來源檔案中 `#include`。這些檔案會被視為位於全域模組片段中。模組只能在全域模組片段中看到其明確包含的標頭中的名稱。全域模組片段只包含實際使用的符號。

```
// MyModuleA.cpp

#include "customlib.h"
#include "anotherlib.h"

import std.core;
import MyModuleB;

//... rest of file
```

您可以使用傳統的標頭檔來控制要匯入的模組：

```
// MyProgram.h
import std.core;
#ifndef DEBUG_LOGGING
import std.filesystem;
#endif
```

## 匯入的標頭檔

### NOTE

本節內容僅供參考。舊版匯入尚未在 Microsoft C++ 編譯器中執行。

有些標頭是完全獨立的，可以使用**import**關鍵字來帶入它們。匯入的標頭和匯入的模組之間的主要差異在於，緊接在**import**語句之後的匯入程式中，會顯示標頭中的任何預處理器定義。（不會顯示該標頭所包含之任何檔案中的預處理器定義）。

```
import <vector>
import "myheader.h"
```

## 另請參閱

[模組、匯入、匯出](#)

# 模組、匯入、匯出

2020/11/2 • [Edit Online](#)

模組、匯入和宣告 `export` 適用于 C++ 20，而且需要 `/experimental: module` 編譯器參數和 `/std: C++ 最新版本`。如需詳細資訊，請參閱 [C++ 中的模組總覽](#)。

## name

將模組宣告放在模組執行檔案的開頭，以指定檔案內容屬於命名模組。

```
module ModuleA;
```

## 匯出

針對模組的主要介面檔案使用匯出模組宣告，其副檔名必須是 `.ixx`：

```
export module ModuleA;
```

在介面檔案中，針對要做為 `export` 公用介面一部分的名稱使用修飾詞：

```
// ModuleA.ixx

export module ModuleA;

namespace Bar
{
    export int f();
    export double d();
    double internal_f(); // not exported
}
```

匯入模組的程式碼看不到非匯出的名稱：

```
//MyProgram.cpp

import module ModuleA;

int main() {
    Bar::f(); // OK
    Bar::d(); // OK
    Bar::internal_f(); // Ill-formed: error C2065: 'internal_f': undeclared identifier
}
```

`export` 關鍵字可能不會出現在模組執行檔案中。當套用 `export` 至命名空間名稱時，會匯出命名空間中的所有名稱。

## import

使用匯入宣告，讓模組的名稱可在您的程式中顯示。匯入宣告必須出現在模組宣告之後，以及任何 `#include` 指示詞之後，但在檔案中的任何宣告之前。

```
module ModuleA;

#include "custom-lib.h"
import std.core;
import std.regex;
import ModuleB;

// begin declarations here:
template <class T>
class Baz
{...};
```

## 備註

匯入和模組只有在邏輯行開頭出現時，才會被視為關鍵字：

```
// 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 特定的

在 Microsoft C++ 中，標記匯入和模組一律是識別碼，而絕不會使用關鍵字做為宏的引數。

### 範例

```
#define foo(...) __VA_ARGS__
foo(
import // Always an identifier, never a keyword
)
```

## 結束 Microsoft 專有

## 另請參閱

[C++ 中的模組概觀](#)

# 樣板 (C++)

2020/11/2 • [Edit Online](#)

範本是 C++ 中泛型程式設計的基礎。就強型別語言而言，C++ 要求所有變數都必須具有特定的型別，不論是由程式設計人員明確宣告或由編譯器推算。不過，許多資料結構和演算法的外觀都相同，不論它們是在哪種類型上運作。範本可讓您定義類別或函數的作業，並讓使用者指定這些作業應使用的具體類型。

## 定義和使用範本

範本是一種結構，它會在編譯時期根據使用者為範本參數所提供的引數來產生一般類型或函式。例如，您可以定義如下的函數樣板：

```
template <typename T>
T minimum(const T& lhs, const T& rhs)
{
    return lhs < rhs ? lhs : rhs;
}
```

上述程式碼描述具有單一類型參數 *T* 之泛型函式的範本，其傳回值和呼叫參數 (*lhs* 和 *rhs*) 都是此類型。您可以將類型參數命名為任何您喜歡的名稱，但依照慣例，最常使用單一大寫字母。*T* 是範本參數；`typename` 關鍵字指出此參數是類型的預留位置。呼叫函式時，編譯器會將每個實例取代 *T* 為使用者所指定或由編譯器推算的具象型別引數。編譯器從範本產生類別或函式的進程稱為樣板具現化；`minimum<int>` 是範本的具現化 `minimum<T>`。

在其他位置，使用者可以宣告專為 int 特殊化的範本實例。假設 `get_a()` 和 `get_b()` 是傳回 int 的函式：

```
int a = get_a();
int b = get_b();
int i = minimum<int>(a, b);
```

不過，因為這是函式樣板，而編譯器可以 *T* 從引數 *a* 和 *b* 推算的類型，所以您可以像一般函數一樣呼叫它：

```
int i = minimum(a, b);
```

當編譯器遇到最後一個語句時，它會產生新的函式，其中範本中每個出現的 *T* 都會取代為 `int`：

```
int minimum(const int& lhs, const int& rhs)
{
    return lhs < rhs ? lhs : rhs;
}
```

編譯器在函式樣板中執行類型推斷的規則是以一般函數的規則為基礎。如需詳細資訊，請參閱 [函數樣板呼叫的多載解析](#)。

## 型別參數

`minimum` 請注意，在上述的範本中，類型參數 *T* 不會以任何方式限定，除非在函式呼叫參數中使用它，其中會加入 `const` 和 `reference` 限定詞。

型別參數的數目沒有實際的限制。以逗號分隔多個參數：

```
template <typename T, typename U, typename V> class Foo{};
```

關鍵字 `class` `typename` 在此內容中相當於。您可以用下列方式表達先前的範例：

```
template <class T, class U, class V> class Foo{};
```

您可以使用省略號運算子 (...) 來定義接受任意數目零或多個型別參數的範本：

```
template<typename... Arguments> class vtclass;

vtclass< > vtinstance1;
vtclass<int> vtinstance2;
vtclass<float, bool> vtinstance3;
```

任何內建或使用者定義的類型都可以當做類型引數使用。例如，您可以使用標準程式庫中的 `std::vector` 來儲存類型為、`int` `double`、`std::string`、`MyClass`、`const MyClass*`、等 `MyClass&` 的變數。使用範本時的主要限制是，類型引數必須支援任何套用至類型參數的作業。例如，如果我們 `minimum` 使用來呼叫，`MyClass` 如下列範例所示：

```
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
}
```

將會產生編譯器錯誤，因為並未提供運算子的多載 `MyClass <`。

任何特定範本的類型引數都屬於相同的物件階層，並不會有任何固有的需求，雖然您可以定義可強制執行這類限制的範本。您可以結合物件導向技術與範本；例如，您可以將衍生的 \* 儲存在向量中 `<Base*>`。請注意，引數必須是指標

```
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` 和其他標準程式庫容器在的專案上強加的基本需求，`T` 是 `T` 可指派和複製可建構的元素。

## 非類型參數

不同于 C# 和 Java 等其他語言的泛型型別，C++ 範本支援非類型參數，也稱為值參數。例如，您可以提供常數整數值來指定陣列的長度，如同此範例，類似于標準程式庫中的 `std::array` 類別：

```
template<typename T, size_t L>
class MyArray
{
    T arr[L];
public:
    MyArray() { ... }
};
```

請注意範本宣告中的語法。`size_t` 值會在編譯時期當做樣板引數傳入，而且必須是 `const` 或 `constexpr` 運算式。您可以使用它，如下所示：

```
MyArray<MyClass*, 10> arr;
```

其他類型的值(包括指標和參考)可以當做非型別參數傳入。例如，您可以將指標傳入函數或函式物件，以自訂範本程式碼內的某些作業。

### 非類型樣板參數的類型推斷

在 Visual Studio 2017 和更新版本中，在 `/std: c++17` 模式中，編譯器會會推算使用所宣告之非類型樣板引數的類型 `auto`：

```
template <auto x> constexpr auto constant = x;

auto v1 = constant<5>;           // v1 == 5, decltype(v1) is int
auto v2 = constant<true>;         // v2 == true, decltype(v2) is bool
auto v3 = constant<'a'>;          // v3 == 'a', decltype(v3) is char
```

## 範本做為範本參數

範本可以是範本參數。在此範例中，`MyClass2` 有兩個範本參數：`typename` 參數 `T` 和樣板參數 `Arr`：

```
template<typename T, template<typename U, int I> class Arr>
class MyClass2
{
    T t; //OK
    Arr<T, 10> a;
    U u; //Error. U not in scope
};
```

因為 `Arr` 參數本身沒有主體，所以不需要其參數名稱。事實上，從的主體中參考 `Arr` 的 `typename` 或類別參數名稱是錯誤的 `MyClass2`。基於這個理由，可以省略 `Arr` 的類型參數名稱，如下列範例所示：

```
template<typename T, template<typename, int> class Arr>
class MyClass2
{
    T t; //OK
    Arr<T, 10> a;
};
```

## 預設範本引數

類別和函式範本可以有預設引數。當範本具有預設引數時，您可以在使用它時將它保留為未指定。例如，`std::vector` 範本具有配置器的預設引數：

```
template <class T, class Allocator = allocator<T>> class vector;
```

在大多數情況下，預設的 std:: 配置器類別是可接受的，因此您可以使用如下的向量：

```
vector<int> myInts;
```

但如有需要，您可以指定自訂配置器，如下所示：

```
vector<int, MyAllocator> ints;
```

若有多個樣板引數，則第一個預設引數之後的所有引數都必須有預設引數。

使用預設為所有參數的範本時，請使用空的角括弧：

```
template<typename A = int, typename B = double>
class Bar
{
    //...
};

...
int main()
{
    Bar<> bar; // use all default type arguments
}
```

## 範本特製化

在某些情況下，範本無法針對任何類型定義完全相同的程式碼，這是不可能或理想的做法。例如，您可能想要定義只有在類型引數是指標、std::wstring 或衍生自特定基類的類型時，才要執行的程式碼路徑。在這種情況下，您可以針對該特定類型定義範本的特製化。當使用者具現化具有該類型的範本時，編譯器會使用特製化來產生類別，而對於所有其他類型，編譯器會選擇較一般的範本。所有參數都特殊化的特製化就是完整特殊化。如果只有部分參數是特製化的，則稱為「部分特殊化」。

```
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
```

只要每個特殊化型別參數都是唯一的，範本可以有任意數目的特製化。只有類別樣板可以部分特製化。範本的所有完整和部分特製化都必須在與原始範本相同的命名空間中宣告。

如需詳細資訊，請參閱[範本特製化](#)。

# typename

2020/11/2 • [Edit Online](#)

在範本定義中，會向編譯器提供提示，指出未知的識別碼是一種類型。在範本參數清單中，是用來指定類型參數。

## 語法

```
typename identifier;
```

## 備註

如果範本定義中的名稱是相依于樣板引數的限定名稱，則必須使用此關鍵字；如果限定名稱不相依，這就是選擇性的。如需詳細資訊，請參閱[範本和名稱解析](#)。

`typename` 可供範本宣告或定義中任何位置的任何類型使用。除非是做為範本基底類別的樣板引數，否則不允許出現在基底類別清單中。

```
template <class T>
class C1 : typename T::InnerType // Error - typename not allowed.
{};
template <class T>
class C2 : A<typename T::InnerType> // typename OK.
{};


```

`typename` 關鍵字也可以用來取代 `class` 範本參數清單中的。例如，下列語句在語義上是相等的：

```
template<class T1, class T2>...
template<typename T1, typename T2>...
```

## 範例

```
// typename.cpp
template<class T> class X
{
    typename T::Y m_y;    // treat Y as a type
};

int main()
{
}
```

## 另請參閱

[範本](#)  
[關鍵字](#)

# 類別樣板

2020/3/25 • [Edit Online](#)

本主題描述C++類別範本特有的規則。

## 類別樣板的成員函式

成員函式可以在類別樣板內部或外部定義。如果是類別樣板外部定義，這些函式的定義方式就像函式樣板。

```
// 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()
{}
```

請注意，就像處理所有樣板類別成員函式一般，類別的建構函式成員函式定義會包含樣板引數清單兩次。

成員函式本身可以是函式樣板，用於指定其他參數，如下列範例所示。

```
// 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()
{}
```

## 嵌套類別樣板

樣板可以在類別或類別樣板內定義，在這種情況下，這些樣板稱為成員樣板。本身是類別的成員樣板稱為巢狀類別樣板。[成員函式樣板](#)中會討論屬於函式的成員範本。

巢狀類別樣板會在外部類別的範圍內宣告為類別樣板。這些樣板可以在封入類別的內部或外部定義。

下列程式碼將示範一般類別內的巢狀類別樣板。

```
// nested_class_template1.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

class X
{
    template <class T>
    struct Y
    {
        T m_t;
        Y(T t): m_t(t) { }
    };

    Y<int> yInt;
    Y<char> yChar;

public:
    X(int i, char c) : yInt(i), yChar(c) { }
    void print()
    {
        cout << yInt.m_t << " " << yChar.m_t << endl;
    }
};

int main()
{
    X x(1, 'a');
    x.print();
}
```

```
// nested_class_template2.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

template <class T>
class X
{
    template <class U> class Y
    {
        U* u;
    public:
        Y();
        U& Value();
        void print();
        ~Y();
    };

    Y<int> y;
public:
    X(T t) { y.Value() = t; }
    void print() { y.print(); }
};
```

```

template <class T>
template <class U>
X<T>::Y<U>::Y()
{
    cout << "X<T>::Y<U>::Y()" << endl;
    u = new U();
}

template <class T>
template <class U>
U& X<T>::Y<U>::Value()
{
    return *u;
}

template <class T>
template <class U>
void X<T>::Y<U>::print()
{
    cout << this->Value() << endl;
}

template <class T>
template <class U>
X<T>::Y<U>::~Y()
{
    cout << "X<T>::Y<U>::~Y()" << endl;
    delete u;
}

int main()
{
    X<int>* xi = new X<int>(10);
    X<char>* xc = new X<char>('c');
    xi->print();
    xc->print();
    delete xi;
    delete xc;
}

//Output:
X<T>::Y<U>::Y()
X<T>::Y<U>::Y()
10
99
X<T>::Y<U>::~Y()
X<T>::Y<U>::~Y()

```

區域類別不可以有成員樣板。

## 朋友的範本

類別樣板可以有[朋友](#)。類別或類別樣板、函式或函式樣板可以是樣板類別的 friend。friend 也可以是類別樣板或函式樣板的特製化，但不是部分特製化。

在下列範例中，friend 函式會定義為類別樣板中的函式樣板。此程式碼會為樣板的每個執行個體產生一個 friend 函式版本。如果您的 friend 函式取決於與類別相同的樣板參數，這個建構會很有用。

```

// template_friend1.cpp
// compile with: /EHsc

#include <iostream>
using namespace std;

template <class T> class Array {
    T* array;

```

```

    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

```

下一個範例內含具有樣板特製化的 friend。如果原始函式樣板為 friend，則函式樣板特製化會自動為 friend。

您也可以只將樣板的特製化版本宣告為 friend，如下列程式碼的 friend 告知之前的註解所示。如果要這麼做，您必

須將 friend 樣板特製化的定義置於樣板類別之外。

```
// 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

```

下一個範例會顯示在類別樣板中宣告的 friend 類別樣板。類別樣板接著會做為 friend 類別的樣板引數。friend 類別樣板必須定義在其宣告的類別樣板之外。friend 樣板的任何特製化或部分特製化也是原始類別樣板的 friend。

```

// 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

```

## 重複使用範本參數

範本參數可以在樣板參數清單中重複使用。例如，下列程式碼是可行的：

```
// 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()
{
}
```

[另請參閱](#)

[範本](#)

# 函式樣板

2020/11/2 • [Edit Online](#)

類別樣板會根據在具現化時傳遞至類別的型別引數，定義一系列相關的類別。函式樣板與類別樣板類似，但其定義一系列的函式。使用函式樣板時，您可以指定一組根據相同的程式碼，但在不同類型或類別上運作的函式。下列函式樣板會交換兩個項目：

```
// function_templates1.cpp
template< class T > void MySwap( T& a, T& b ) {
    T c(a);
    a = b;
    b = c;
}
int main() {
```

此程式碼會定義一系列交換引數值的函式。從這個範本，您可以產生將交換 `int` 和 `long` 類型以及使用者定義類型的函式。如果已正確定義類別的複製建構函式和指派運算子，則 `MySwap` 甚至會交換類別。

此外，函數樣板會防止您交換不同類型的物件，因為編譯器會在編譯時期知道 `a` 和 `b` 參數的類型。

雖然使用 `void` 指標可讓這個函式由非樣板化的函式執行，但樣板版本仍是 typesafe。請考慮下列呼叫：

```
int j = 10;
int k = 18;
CString Hello = "Hello, Windows!";
MySwap( j, k );           //OK
MySwap( j, Hello );      //error
```

因為編譯器無法產生具有不同類型之參數的 `MySwap` 函式，因此第二個 `MySwap` 呼叫會觸發編譯時間錯誤。如果使用了 `void` 指標，則會正確編譯兩個函式呼叫，不過，函式在執行階段將無法正常運作。

您可以明確指定函式樣板的樣板引數。例如：

```
// 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.
}
```

明確指定樣板引數時，一般會完成隱含轉換，以便將函式引數轉換為對應函式樣板參數的類型。在上述範例中，編譯器會將轉換 `j` 成類型 `char`。

## 另請參閱

[範本](#)

[函式樣板具現化](#)

[明確具現化](#)

[函式樣板的明確特製化](#)

# 函式樣板具現化

2020/3/25 • [Edit Online](#)

先呼叫每種類型的函式樣板時，編譯器會建立具現化。每次具現化都是一種為類型特製化樣板函式的版本。每次為類型使用函式時，都會呼叫此具現化。如果您有多個相同的具現化（即使分屬不同模組），可執行檔中只能有一個具現化複本。

任何引數和參數組合的函式樣板皆可進行函式引數轉換，前提是參數不可取決於該樣板引數。

以特定類型為引數宣告樣板，即可明確具現化該函式樣板。例如，下列程式碼是可行的：

```
// 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()
{
}
```

## 另請參閱

[函式樣板](#)

# 明確初始化

2020/11/2 • [Edit Online](#)

您可以使用明確具現化，建立樣板化類別或函式的具現化，而在程式碼中實際使用它。由於當您建立使用樣板散發的程式庫 (.lib) 檔案時，這樣做很有用的，未具現化的樣板定義不會進入目的檔 (.obj)。

此程式碼明確具現化 `MyStack<int, 6>` 變數和六個專案：

```
template class MyStack<int, 6>;
```

這個陳述式建立 `MyStack` 的具現化，沒有保留物件的任何儲存區。程式碼為所有成員產生。

下一行只明確具現化建構函式成員函式：

```
template MyStack<int, 6>::MyStack( void );
```

您可以使用特定的型別引數來重新宣告函式樣板，以明確地具現化函數樣板，如函數樣板具現化中的範例所示。

您可以使用 `extern` 關鍵字來防止成員的自動具現化。例如：

```
extern template class MyStack<int, 6>;
```

同樣地，您可以將特定成員標記為外部和未具現化：

```
extern template MyStack<int, 6>::MyStack( void );
```

您可以使用 `extern` 關鍵字，讓編譯器不會在一個以上的物件模組中產生相同的具現化程式碼。如果函式被呼叫，您必須在至少一個連結模組中使用指定的明確樣板參數，來具現化樣板函式，否則當程式建立時會發生連結器錯誤。

## NOTE

特製 `extern` 化中的關鍵字僅適用於在類別主體之外定義的成員函式。類別宣告內定義的函式被視為內嵌函式，永遠會具現化。

## 另請參閱

[函數樣板](#)

# 函式樣板的明確特製化

2020/11/2 • [Edit Online](#)

透過函式樣板，您可以提供該類型的明確特製化（覆寫）函式樣板，來定義該特定類型的特殊行為。例如：

```
template<> void MySwap(double a, double b);
```

這個宣告可讓您為變數定義不同的函式 `double`。就像非樣板函式，會套用標準類型轉換（例如將類型的變數升級 `float` 為 `double`）。

## 範例

```
// 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()
{
}
```

## 另請參閱

[函數樣板](#)

# 函式樣板的部分排序 (C++)

2019/12/2 • [Edit Online](#)

提供多個符合函式呼叫之引數清單的函式樣板。C++ 可定義函式樣板的部分排序，以指定應呼叫哪些函式。進行部分排序是因為某些樣板會被視為等同於特製化。

編譯器會從可能相符的項目中選取最特殊的範本函式。例如，如果函式樣板採用類型 `T`，而且有另一個可使用的函式範本 `T*`，`T*` 則會將版本視為更特殊化。當引數為指標類型 `T` 時，其偏好高於泛型版本，即使兩者都是允許的相符專案。

請使用下列程序來判斷某個函式樣板是否為特製化程度較高的候選項目：

1. 考量兩個函式樣板，`T1` 和 `T2`。
2. 使用一個假設的唯一類型 `X` 來取代 `T1` 中的參數。
3. 透過 `T1` 中的參數清單，查看 `T2` 對於該參數清單來說是否為有效的範本。忽略所有隱含轉換。
4. 以相反順序對 `T1` 和 `T2` 重複進行相同的程序。
5. 如果一個範本是其他範本的有效樣板引數清單，但反之不是 `true`，則該範本會被視為比其他範本更不特殊化。  
如果您使用上一個步驟，這兩個範本會形成彼此的有效引數，則會被視為相同的特製化，而當您嘗試使用它們時，會將其稱為不明確的呼叫結果。
6. 請使用下列規則：
  - a. 特定類型的樣板特製化比採用泛型類型引數的樣板特製化程度更高。
  - b. 只採取的範本 `T*` 只會比一個 `T` 使用更具特製化，因為假設 `x*` 的類型是樣板引數 `T` 的有效引數 `x`，但不是的有效引數 `T*` 樣板引數。
  - c. `const T` 比 `T` 更特殊化，因為 `const x` 是樣板引數的有效自 `T` 變數，`const T` 但 `x` 不是樣板引數的有效引數。
  - d. `const T*` 比 `T*` 更特殊化，因為 `const x*` 是樣板引數的有效自 `T*` 變數，`const T*` 但 `x*` 不是樣板引數的有效引數。

## 範例

下列範例的運作方式如標準中所指定：

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

template <class T> void f(T) {
    printf_s("Less specialized function called\n");
}

template <class T> void f(T*) {
    printf_s("More specialized function called\n");
}

template <class T> void f(const T*) {
    printf_s("Even more specialized function for const T*\n");
}

int main() {
    int i = 0;
    const int j = 0;
    int *pi = &i;
    const int *cpi = &j;

    f(i);    // Calls less specialized function.
    f(pi);   // Calls more specialized function.
    f(cpi); // Calls even more specialized function.
    // Without partial ordering, these calls would be ambiguous.
}
```

## Output

```
Less specialized function called
More specialized function called
Even more specialized function for const T*
```

## 另請參閱

[函式樣板](#)

# 成員函式樣板

2020/11/2 • [Edit Online](#)

成員樣板這個詞同時指成員函式樣板和巢狀類別樣板。成員函式樣板是樣板函式，為類別或類別樣板的成員。

成員函式可以是數種內容中的函式樣板。類別樣板的所有函式都是泛型，但不稱為成員樣板或成員函式樣板。如果這些成員函式採用自己的樣板引數，則會視為成員函式樣板。

## 範例：Declare 成員函式範本

非樣板或樣板類別的成員函式樣板會宣告為函式樣板，並且包含其樣板參數。

```
// member_function_templates.cpp
struct X
{
    template <class T> void mf(T* t) {}
};

int main()
{
    int i;
    X* x = new X();
    x->mf(&i);
}
```

## 範例：範本類別的成員函式範本

下列範例將示範樣板類別的成員函式樣板。

```
// member_function_templates2.cpp
template<typename T>
class X
{
public:
    template<typename U>
    void mf(const U &u)
    {
    }
};

int main()
```

## 範例：在類別外部定義成員範本

```

// 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()
{
}

```

## 範例：樣板化使用者定義轉換

區域類別不可以有成員樣板。

成員樣板函式不可以是虛擬函式，而且宣告時的名稱與基底類別虛擬函式相同時，不可覆寫來自基底類別的虛擬函式。

下列範例顯示樣板化使用者定義的轉換：

```

// 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>.
}

```

## 另請參閱

[函數範本](#)

# 樣板特製化 (C++)

2020/11/2 • [Edit Online](#)

類別樣板可以部分特製化，而產生的類別仍然是樣板。部分特製化可以視情況針對特定類型部分自訂樣板程式碼，例如：

- 樣板可以有多個類型，只需特製化其中一部分。結果是在其他類型參數化的樣板。
- 一個樣板只能有一個類型，不過，指標、參考、成員指標或函式指標類型都需要特製化。特製化本身仍然是指向類型或參考類型的樣板。

## 範例：類別樣板的部分特製化

```
// partial_specialization_of_class_templates.cpp
template <class T> struct PTS {
    enum {
        IsPointer = 0,
        IsPointerToDataMember = 0
    };
};

template <class T> struct PTS<T*> {
    enum {
        IsPointer = 1,
        IsPointerToDataMember = 0
    };
};

template <class T, class U> struct PTS<T U::*> {
    enum {
        IsPointer = 0,
        IsPointerToDataMember = 1
    };
};

struct S{};

extern "C" int printf_s(const char*,...);

int main() {
    printf_s("PTS<S>::IsPointer == %d PTS<S>::IsPointerToDataMember == %d\n",
            PTS<S>::IsPointer, PTS<S>:: IsPointerToDataMember);
    printf_s("PTS<S*>::IsPointer == %d PTS<S*>::IsPointerToDataMember ==%d\n"
            , PTS<S*>::IsPointer, PTS<S*>:: IsPointerToDataMember);
    printf_s("PTS<int S::*>::IsPointer == %d PTS<
            <int S::*>::IsPointerToDataMember == %d\n",
            PTS<int S::*>::IsPointer, PTS<int S::*>::
            IsPointerToDataMember);
}
```

```
PTS<S>::IsPointer == 0 PTS<S>::IsPointerToDataMember == 0
PTS<S*>::IsPointer == 1 PTS<S*>::IsPointerToDataMember ==0
PTS<int S::*>::IsPointer == 0 PTS<int S::*>::IsPointerToDataMember == 1
```

## 範例：指標類型的部分特製化

如果您有採用任何類型的樣板集合類別 `T`，您可以建立採用任何指標類型的部分特製化 `T*`。下列程式碼示範集合類別樣板 `Bag` 和指標類型的部分特製化，其中該集合會在將指標類型複製到陣列之前取值指標類型。然後，該集合會儲存指向的值。若使用原始樣板，只能將指標本身儲存在集合中，因此資料容易遭到刪除或修改。在這種特殊指標版本的集合中，會在 `add` 方法中新增檢查 null 指標的程式碼。

```
// 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;
    }
}
```

```

    elem[size++] = *t; // Dereference
}

void print() {
    for (int i = 0; i < size; i++)
        cout << elem[i] << " ";
    cout << endl;
}
};

int main() {
    Bag<int> xi;
    Bag<char> xc;
    Bag<int*> xp; // Uses partial specialization for pointer types.

    xi.add(10);
    xi.add(9);
    xi.add(8);
    xi.print();

    xc.add('a');
    xc.add('b');
    xc.add('c');
    xc.print();

    int i = 3, j = 87, *p = new int[2];
    *p = 8;
    *(p + 1) = 100;
    xp.add(&i);
    xp.add(&j);
    xp.add(p);
    xp.add(p + 1);
    p = NULL;
    xp.add(p);
    xp.print();
}
}

```

```

10 9 8
a b c
Null pointer!
3 87 8 100

```

## 範例：定義部分特製化，使其中一個類型為 `int`

下列範例會定義一個樣板類別，這個類別會採用兩種類型的配對，然後定義該樣板類別特製化的部分特製化，讓其中一個類型成為 `int`。特製化會額外定義排序方法，這個方法會根據整數實作簡單的反昇排序。

```

// partial_specialization_of_class_templates3.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

template <class Key, class Value> class Dictionary {
    Key* keys;
    Value* values;
    int size;
    int max_size;
public:
    Dictionary(int initial_size) : size(0) {
        max_size = 1;
        while (initial_size >= max_size)
            max_size *= 2;
        keys = new Key[max_size];
    }
}
```

```

        values = new Value[max_size];
    }

    void add(Key key, Value value) {
        Key* tmpKey;
        Value* tmpVal;
        if (size + 1 >= max_size) {
            max_size *= 2;
            tmpKey = new Key [max_size];
            tmpVal = new Value [max_size];
            for (int i = 0; i < size; i++) {
                tmpKey[i] = keys[i];
                tmpVal[i] = values[i];
            }
            tmpKey[size] = key;
            tmpVal[size] = value;
            delete[] keys;
            delete[] values;
            keys = tmpKey;
            values = tmpVal;
        }
        else {
            keys[size] = key;
            values[size] = value;
        }
        size++;
    }

    void print() {
        for (int i = 0; i < size; i++)
            cout << "{" << keys[i] << ", " << values[i] << "}" << endl;
    }
};

// Template partial specialization: Key is specified to be int.
template <class Value> class Dictionary<int, Value> {
    int* keys;
    Value* values;
    int size;
    int max_size;
public:
    Dictionary(int initial_size) : size(0) {
        max_size = 1;
        while (initial_size >= max_size)
            max_size *= 2;
        keys = new int[max_size];
        values = new Value[max_size];
    }
    void add(int key, Value value) {
        int* tmpKey;
        Value* tmpVal;
        if (size + 1 >= max_size) {
            max_size *= 2;
            tmpKey = new int [max_size];
            tmpVal = new Value [max_size];
            for (int i = 0; i < size; i++) {
                tmpKey[i] = keys[i];
                tmpVal[i] = values[i];
            }
            tmpKey[size] = key;
            tmpVal[size] = value;
            delete[] keys;
            delete[] values;
            keys = tmpKey;
            values = tmpVal;
        }
        else {
            keys[size] = key;
            values[size] = value;
        }
    }
};

```

```

        size++;
    }

void sort() {
    // Sort method is defined.
    int smallest = 0;
    for (int i = 0; i < size - 1; i++) {
        for (int j = i; j < size; j++) {
            if (keys[j] < keys[smallest])
                smallest = j;
        }
        swap(keys[i], keys[smallest]);
        swap(values[i], values[smallest]);
    }
}

void print() {
    for (int i = 0; i < size; i++)
        cout << "{" << keys[i] << ", " << values[i] << "}" << endl;
}
};

int main() {
    Dictionary<char*, char*>* dict = new Dictionary<char*, char*>(10);
    dict->print();
    dict->add("apple", "fruit");
    dict->add("banana", "fruit");
    dict->add("dog", "animal");
    dict->print();

    Dictionary<int, char*>* dict_specialized = new Dictionary<int, char*>(10);
    dict_specialized->print();
    dict_specialized->add(100, "apple");
    dict_specialized->add(101, "banana");
    dict_specialized->add(103, "dog");
    dict_specialized->add(89, "cat");
    dict_specialized->print();
    dict_specialized->sort();
    cout << endl << "Sorted list:" << endl;
    dict_specialized->print();
}

```

```

{apple, fruit}
{banana, fruit}
{dog, animal}
{100, apple}
{101, banana}
{103, dog}
{89, cat}

Sorted list:
{89, cat}
{100, apple}
{101, banana}
{103, dog}

```

# 樣板和名稱解析

2020/3/25 • [Edit Online](#)

在樣板定義中共有三種類型的名稱。

- 區域宣告名稱，包括樣板的名稱本身和樣板定義中宣告的任何名稱。
- 來自樣板定義封閉範圍之外的名稱。
- 以某種方式取決於樣板引數的名稱，也稱為相依名稱。

雖然前兩個名稱也屬於類別和函式範圍，不過在樣板定義中需要使用名稱解析的特殊規則來處理相依名稱增加的複雜度。這是因為編譯器在樣板具現化之前對這些名稱幾乎一無所知，它們可能是完全不同的類型，取決於所使用的樣板引數。非相依名稱會根據一般規則和樣板的定義點進行搜尋。這些名稱（與樣板引數無關）只會針對所有樣板特製化搜尋一次。相依名稱會在樣板具現化之後再進行搜尋，並且會分別對每個特製化進行搜尋。

類型取決於其相關的樣板引數。具體來說，類型是相依的，如果其符合下列各項：

- 樣板引數本身：

```
T
```

- 具有限定性的限定名稱，包含一個相依類型：

```
T::myType
```

- 一個限定名稱，但未限定部可識別一個相依類型：

```
N::T
```

- 基底類型為相依類型的 const 或 volatile 類型：

```
const T
```

- 以相依類型為基礎的指標、參考、陣列或函式指標類型：

```
T *, T &, T [10], T (*)()
```

- 一個陣列，其大小取決於樣板參數：

```
template <int arg> class X {  
    int x[arg] ; // dependent type  
}
```

- 從樣板參數建構的範本類型：

```
T<int>, MyTemplate<T>
```

## 類型相依和值相依

相依於樣板參數的名稱和運算式會分類為類型相依或值相依，取決於樣板參數是類型參數或值參數。此外，在具有類型相依的樣板中宣告的所有識別項在樣板引數上視為值相依，如使用值相依運算式初始化的整數或列舉類型。

類型相依和值相依運算式是一種包含類型相依或值相依變數的運算式。根據範本所使用的參數，這些運算式可能會有不同的語意。

另請參閱

[範本](#)

# 相依類型的名稱解析

2020/11/2 • [Edit Online](#)

`typename` 在範本定義中使用做為限定名稱，告知編譯器指定的限定名稱會識別型別。如需詳細資訊，請參閱[typename](#)。

```
// template_name_resolution1.cpp
#include <stdio.h>
template <class T> class X
{
public:
    void f(typename T::myType* mt) {}
};

class Yarg
{
public:
    struct myType { };
};

int main()
{
    X<Yarg> x;
    x.f(new Yarg::myType());
    printf("Name resolved by using typename keyword.");
}
```

Name resolved by using typename keyword.

相依名稱的名稱查閱會檢查範本定義內容中的名稱(在下列範例中，此內容會尋找 `myFunction(char)` )和範本具現化的內容。在下列範例中，範本會在 `main` 中具現化;因此，`MyNamespace::myFunction` 會從具現化的點看到，並挑選為較佳的相符項。如果 `MyNamespace::myFunction` 已重新命名，則會改為呼叫 `myFunction(char)` 。

所有名稱都會做為相依名稱解析。儘管如此，如果可能發生任何衝突，仍建議您使用完整限定名稱。

```
// 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);
}
```

## 輸出

```
Int MyNamespace::myFunction
```

## 樣板去除混淆

Visual Studio 2012 強制使用 C++ 98/03/11 標準規則來去除 "template" 關鍵字的混淆。在下列範例中，Visual Studio 2010 會接受不合格的行和符合的行。Visual Studio 2012 只接受符合規範的行。

```
#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;
}
```

遵循去除混淆規則是必要條件，因為根據預設，C++ 會假設 `AY::Rebind` 不是樣板，因此編譯器會將下列 "`<`" 解譯為小於。編譯器必須知道 `Rebind` 是樣板，才能將 "`<`" 正確剖析為角括號。

## 另請參閱

[名稱解析](#)

# 區域宣告名稱的名稱解析

2020/11/2 • [Edit Online](#)

樣本的名稱本身在參考時可以具有或不具有樣板引數。在類別樣板的範圍內，名稱本身會參考樣板。在樣板特製化或部分特製化的範圍內，個別本身可參考特製化或部分特製化。使用適當的樣板引數可以參考樣板的其他特製化或部分特製化。

## 範例：特製化與部分特製化

下列程式碼顯示在特製化或部分特製化的範圍內，類別樣板的名稱 A 的不同解譯。

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

## 範例：範本參數與物件之間的名稱衝突

在樣板參數和其他物件之間發生名稱衝突的情況下，樣板參數可以或無法隱藏。下列規則可協助判斷優先順序。

樣板參數的範圍是從第一次出現的位置，一直到類別或函式樣板的結尾。如果名稱再次出現在樣板引數清單或基底類別清單中，其會參考相同的類型。在 Standard C++ 中，在相同範圍內不可以宣告與樣板參數相同的名稱。

Microsoft 擴充功能允許在樣板的範圍內重新定義樣板參數。下列範例顯示在類別樣板的基底規格中使用樣板參數。

```
// template_name_resolution4.cpp
// compile with: /EHsc
template <class T>
class Base1 {};

template <class T>
class Derived1 : Base1<T> {};

int main() {
    // Derived1<int> d;
}
```

## 範例：在類別樣板之外定義成員函式

在類別樣板的外部定義樣板的成員函式時，可以使用不同的樣板參數名稱。如果樣板成員函式定義使用與宣告不同的樣板參數名稱，而且在定義中使用的名稱與宣告的其他成員發生衝突，則會優先使用樣板宣告中的成員。

```

// template_name_resolution5.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

template <class T> class C {
public:
    struct Z {
        Z() { cout << "Z::Z()" << endl; }
    };
    void f();
};

template <class Z>
void C<Z>::f() {
    // Z refers to the struct Z, not to the template arg;
    // Therefore, the constructor for struct Z will be called.
    Z z;
}

int main() {
    C<int> c;
    c.f();
}

```

Z::Z()

## 範例：定義命名空間外部的範本或成員函式

在宣告樣板的命名空間外部定義樣板函式或成員函式時，樣板引數會優先於命名空間的其他成員名稱。

```

// template_name_resolution6.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

namespace NS {
    void g() { cout << "NS::g" << endl; }

    template <class T> struct C {
        void f();
        void g() { cout << "C<T>::g" << endl; }
    };
}

template <class T>
void NS::C<T>::f() {
    g(); // C<T>::g, not NS::g
}

int main() {
    NS::C<int> c;
    c.f();
}

```

C<T>::g

## 範例：基底類別或成員名稱隱藏範本引數

在樣板類別宣告外部的定義中，如果樣板類別具有未相依於樣板引數的基底類別，而且，如果基底類別或其中一個成員的名稱與樣板引數相同，則基底類別或成員名稱會隱藏樣板引數。

```
// template_name_resolution7.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

struct B {
    int i;
    void print() { cout << "Base" << endl; }
};

template <class T, int i> struct C : public B {
    void f();
};

template <class B, int i>
void C<B, i>::f() {
    B b; // Base class b, not template argument.
    b.print();
    i = 1; // Set base class's i to 1.
}

int main() {
    C<int, 1> c;
    c.f();
    cout << c.i << endl;
}
```

```
Base
1
```

## 另請參閱

[名稱解析](#)

# 函式樣板呼叫的多載解析

2020/11/2 • [Edit Online](#)

函式樣板可以多載相同名稱的非樣板函式。在這種情節中，函式呼叫會先使用樣板引數推算解析，以具現化具有唯一特製化的函式樣板。如果樣板引數推算失敗，才會考慮使用其他函式多載解析呼叫。這些其他多載也稱為候選集合，包括非樣板函式和其他具現化的函式樣板。如果樣板引數推算成功，則會依照多載解析的規則將產生的函式與其他函式進行比較，以判斷最符合的項目。如需詳細資訊，[請參閱函式多載](#)。

## 範例：選擇非範本函數

如果非樣板函式是與樣板函式一樣理想的相符項目，則會選擇非樣板函式（除非明確指定樣板引數），如同下列範例中的呼叫 `f(1, 1)`。

```
// 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 template function.
    f<int, int>(2, 2); // Template arguments explicitly specified.
}
```

```
f(int, int)
void f(T1, T2)
void f(T1, T2)
```

## 範例：慣用完全相符的範本函式

下一個範例將說明，如果非樣板函式需要轉換，則建議您使用完全相符的樣板函式。

```
// template_name_resolution10.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

void f(int, int) { cout << "f(int, int)" << endl; }

template <class T1, class T2>
void f(T1, T2)
{
    cout << "void f(T1, T2)" << endl;
};

int main()
{
    long l = 0;
    int i = 0;
    // Call the template function f(long, int) because f(int, int)
    // would require a conversion from long to int.
    f(l, i);
}
```

```
void f(T1, T2)
```

## 另請參閱

[名稱解析](#)

[typename](#)

# 原始程式碼組織 (C++ 範本)

2020/11/2 • [Edit Online](#)

定義類別範本時，您必須以這種方式組織原始程式碼，在需要時將成員定義顯示給編譯器。您可以選擇使用「包含模型」\*\* 或「明確具現化」\*\* 模型。在包含模型中，您可以在使用範本的每個檔案中包含成員定義。此方法最簡單，且在可以使用範本的實體類別方面提供最大彈性。其缺點是可能會增加編譯時間。如果專案和/或包含的檔案本身很大，則影響可能會很大。使用明確具現化方法時，範本本身會具現化特定類型的實體類別或類別成員。這種方法可以加快編譯時間，但僅限於範本實作者事先啟用之類別的使用方式。除非編譯時間會造成問題，否則在一般情況下我們建議您使用包含模型。

## 背景

以編譯器不為範本或其任何成員產生物件程式碼的意義來說，範本與一般類別不同。在使用實體類別將範本具現化之前，不會產生任何內容。當編譯器遇到範本具現化（例如 `MyClass<int> mc;`）且不存在具有該簽章的類別時，將產生新的類別。也會嘗試為所使用的任何成員函式產生程式碼。如果這些定義位於不直接或間接包含於正在編譯的 .cpp 檔案中，則編譯器將無法看到這些定義。從編譯器的觀點來看這不一定是錯誤，因為函式可能會在另一個轉譯單位中定義，在此情況下連結器會找到這些函式。如果連結器找不到該程式碼，則會引發無法解析的外部錯誤。

## 包含模型

在整個轉譯單位中顯示範本定義的最簡單和最常用方法是將定義放在標頭檔案中。任何使用範本的任何 .cpp 檔案都必須包含標頭。這是標準程式庫中使用的方法。

```
#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
```

使用此方法時，編譯器可以存取完整的範本定義，並可依需求將任何類型的範本具現化。此方法簡單且相對較容易維護。不過，包含模型具有編譯時間方面的成本。在大型程式中，這項成本可能很大，特別是當範本標頭本身包含其他標頭時。`#Includes` 標頭的每個檔案都會取得自己的函式樣板 `.cpp` 和所有定義的複本。連結器通常能夠妥善處理，使您不至於面對函式具有多個定義的情況，但完成此工作需要時間。在較小型的程式中，可能只需較短的額外編譯時間。

## 明確具現化模型

如果您的專案無法使用包含模型，而且您知道將用來具現化範本的類型集合，您可以將範本程式碼分隔成 `.h` 和檔案 `.cpp`，並在檔案中 `.cpp` 明確地具現化範本。這會導致產生當編譯器遇到使用者具現化將會看到的物件程式碼。

您可以使用關鍵字範本，後面接著要具現化之實體的簽章，來建立明確具現化。可以是類型或成員。如果您將類型明確具現化，則所有成員會具現都化。

```
template MyArray<double, 5>;
```

```
//MyArray.h
#ifndef MYARRAY
#define MYARRAY

template<typename T, size_t N>
class MyArray
{
    T arr[N];
public:
    MyArray();
    void Print();
    T& operator[](int i);
};

#endif
```

```
//MyArray.cpp
#include <iostream>
#include "MyArray.h"

using namespace std;

template<typename T, size_t N>
MyArray<T,N>::MyArray(){}

template<typename T, size_t N>
void MyArray<T,N>::Print()
{
    for (const auto v : arr)
    {
        cout << v << " ";
    }
    cout << endl;
}

template MyArray<double, 5>;template MyArray<string, 5>;
```

在上述範例中，明確具現化是在 `.cpp` 檔案的底部。只能用於 `MyArray`、`double` 或 `String` 類型。

### NOTE

在 C++ 11 中，`export` 關鍵字在範本定義的內容中已被取代。實際上這不會有太大的影響，因為大多數的編譯器都未對其提供支援。

# 事件處理

2020/12/10 • [Edit Online](#)

(c + + 類別(通常會使用 ATL 類別或 `coclass` 屬性) )來處理 com 類別，則主要支援事件處理。如需詳細資訊，請參閱 [COM 中的事件處理](#)。

原生 c + + 類別也支援事件處理(不會)執行 COM 物件的 c + + 類別。原生 c + + 事件處理支援已被取代，並將在未來的版本中移除。如需詳細資訊，請參閱 [原生 c + + 中的事件處理](#)。

## NOTE

原生 c + + 中的事件屬性與 Standard c + + 不相容。當您指定一致性模式時，不會編譯它們 `/permissive-`。

事件處理支援單一和多執行緒的使用。它可保護資料不受同時多執行緒存取。您可以從事件來源或接收者類別衍生子類別。這些子類別支援擴充的事件來源和接收。

Microsoft c + + 編譯器包含用來宣告事件和事件處理常式的屬性和關鍵字。事件屬性和關鍵字可以在 CLR 程式與原生 C++ 程式中使用。

III	II
<code>event_source</code>	建立事件來源。
<code>event_receiver</code>	建立事件接收器(接收)。
<code>_event</code>	宣告事件。
<code>_raise</code>	強調事件的呼叫位置。
<code>_hook</code>	建立處理常式方法與事件的關聯。
<code>_unhook</code>	解除處理常式方法與事件之間的關係。

## 另請參閱

[C + + 語言參考](#)  
[關鍵字](#)

# `_event` 關鍵字

2020/12/10 • [Edit Online](#)

宣告事件。

## NOTE

原生 C++ 中的事件屬性與 Standard C++ 不相容。當您指定一致性模式時，不會編譯它們 `/permissive-`。

## 語法

```
_event member-function-declarator ;
/_event _interface interface-specifier ;
/_event data-member-declarator ;
```

## 備註

Microsoft 專有的關鍵字 `_event` 可以套用至成員函式宣告、介面宣告或資料成員宣告。不過，您無法使用 `_event` 關鍵字來限定嵌套類別的成員。

根據您的事件來源和接收器是原生 C++, COM 或 Managed (.NET Framework)，您可以將下列建構當做事件使用：

NATIVE C++	COM	MANAGED (.NET FRAMEWORK)
成員函式	-	method
-	interface	-
-	-	資料成員

`_hook` 在事件接收器中使用，將處理常式成員函式與事件成員函式產生關聯。當您使用關鍵字建立事件之後 `_event`，會在呼叫該事件時呼叫該事件的所有事件處理常式。

`_event` 成員函式宣告不能有定義，而是以隱含方式產生定義，所以可以呼叫事件成員函式，就像是任何一般成員函式一樣。

## NOTE

樣板化類別或結構不能包含事件。

## 原生事件

原生事件是成員函式。傳回型別通常是 `HRESULT` 或 `void`，但可以是任何整數類型，包括 `enum`。當事件使用整數傳回型別時，當事件處理常式傳回非零值時，就會定義錯誤條件。在此情況下，引發的事件會呼叫其他委派。

```
// Examples of native C++ events:
_event void OnDblClick();
_event HRESULT OnClick(int* b, char* s);
```

請參閱 [原生 c++ 中的事件處理](#) 以取得範例程式碼。

## COM 事件

COM 事件是介面。事件來源介面中成員函式的參數應該是在參數 `out` 中，但未嚴格強制執行。這是因為當多播時，`out` 參數並沒有用。如果您使用 `out` 參數，則會發出層級 1 警告。

傳回型別通常是 `HRESULT` 或 `void`，但可以是任何整數類型，包括 `enum`。當事件使用整數傳回型別，而事件處理常式傳回非零值時，就是錯誤的狀況。引發的事件會中止對其他委派的呼叫。編譯器會自動將事件來源介面標示為 `source` 產生的 IDL 中的。

在 `_interface` COM 事件來源之後，一律需要關鍵字 `_event`。

```
// Example of a COM event:  
_event _interface IEvent1;
```

如需範例程式碼，請參閱 [COM 中的事件處理](#)。

## 受控事件

如需新語法中程式碼撰寫事件的詳細資訊，請參閱 [事件](#)。

Managed 事件是資料成員或成員函式。搭配事件使用時，委派的傳回型別必須符合 [Common Language Specification](#) 的規範。事件處理常式的傳回類型必須符合委派的傳回類型。如需委派的詳細資訊，請參閱 [委派和事件](#)。如果 Managed 事件是資料成員，其類型必須是委派的指標。

在 .NET Framework 中，您可以將資料成員視同方法本身（也就是，其對應委派的 `Invoke` 方法）。若要這樣做，請預先定義宣告 managed 事件資料成員的委派類型。相反地，managed 事件方法會隱含定義對應的 managed 委派（如果尚未定義的話）。例如，您可以將事件的值（例如 `onClick`）宣告為事件，如下所示：

```
// Examples of managed events:  
_event ClickEventHandler* OnClick; // data member as event  
_event void OnClick(String* s); // method as event
```

當隱含宣告 managed 事件時，您可以指定 `add` `remove` 在新增或移除事件處理常式時呼叫的和存取子。您也可以定義成員函式，以呼叫（從類別外部引發）事件。

## 範例：原生事件

```
// EventHandling_Native_Event.cpp  
// compile with: /c  
[event_source(native)]  
class CSource {  
public:  
    _event void MyEvent(int nValue);  
};
```

## 範例：COM 事件

```
// EventHandling_COM_Event.cpp
// compile with: /c
#define _ATL_ATTRIBUTES 1
#include <atbase.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;
    }
};


```

## 另請參閱

關鍵字

事件處理

[event\\_source](#)

[event\\_receiver](#)

[\\_hook](#)

[\\_unhook](#)

[\\_raise](#)

\_hook

# 關鍵字

2020/12/10 • [Edit Online](#)

建立處理常式方法與事件的關聯。

## NOTE

原生 C++ 中的事件屬性與 Standard C++ 不相容。當您指定一致性模式時，不會編譯它們 `/permissive-`。

## 語法

```
long __hook(
    &SourceClass::EventMethod,
    source,
    &ReceiverClass::HandlerMethod
    [, receiver = this]
);

long __hook(
    interface,
    source
);
```

## 參數

`&SourceClass::EventMethod`

您用於連結事件處理常式方法的事件方法指標：

- 原生 C++ 事件：`SourceClass` 是事件來源類別，而 `EventMethod` 是事件。
- COM 事件：`SourceClass` 是事件來源介面，而且 `EventMethod` 是它的其中一個方法。
- Managed 事件：`SourceClass` 是事件來源類別，而 `EventMethod` 是事件。

`interface`

要連結的介面名稱 `receiver`，僅適用於屬性的參數為的 COM 事件接收器 `Layout_dependent` `event_receiver` `true`。

`source`

事件來源執行個體的指標。根據 `type` 中指定的程式碼 `event_receiver`，`source` 可以是下列其中一種類型：

- 原生事件來源物件指標。
- `IUnknown` COM 來源) 的指標。
- Managed 物件指標 (用於 Managed 事件)。

`&ReceiverClass::HandlerMethod`

要連結至事件的事件處理常式方法的指標。此處理程式會指定為類別的方法或相同的參考。如果您未指定類別名稱，則 `_hook` 會假設類別是從中呼叫它的類別。

- 原生 C++ 事件：`ReceiverClass` 是事件接收器類別，而 `HandlerMethod` 是處理常式。
- COM 事件：`ReceiverClass` 是事件接收器介面，而且 `HandlerMethod` 是它的其中一個處理常式。

- Managed 事件：`ReceiverClass` 是事件接收器類別，而 `HandlerMethod` 是處理常式。

#### receiver

(選擇性) 事件接收器類別實例的指標。如果您未指定接收者，預設值就是呼叫的接收者類別或結構 `_hook`。

## 使用方式

可以在事件接收器類別之外的任何函式範圍中使用，包括 main。

## 備註

`_hook` 在事件接收器中使用內建函數，將處理常式方法與事件方法產生關聯或連結。然後，當來源引發指定事件時，就會呼叫指定的處理常式。您可以將數個處理常式連結至單一事件，也可以將數個事件連結至單一處理常式。

有兩種形式 `_hook`。在大部分的情況下，您可以使用第一個 (四個引數) 的表單，特別是針對屬性的 *layout\_dependent* 參數為的 COM 事件接收器 `event_receiver` `false`。

在這些情況下，您不需要在其中一個方法上引發事件之前，先攔截介面中的所有方法。您只需要攔截處理事件的方法。您可以使用第二個 (雙引數) 形式的，`_hook` 僅適用於中的 COM 事件接收器 `Layout_dependent` = true。

`_hook` 傳回 long 值。非零傳回值表示已發生錯誤 (Managed 事件擲回例外狀況)。

編譯器會檢查事件是否存在，以及事件簽章與委派簽章是否一致。

`_hook` `_unhook` 除了 COM 事件之外，您可以在事件接收器之外呼叫。

使用的替代方法 `_hook` 是使用 += 運算子。

如需以新語法撰寫 managed 事件程式碼的詳細資訊，請參閱 [event](#)。

#### NOTE

樣板類別或結構不能包含事件。

## 範例

如需範例，請參閱 [原生 C++ 中的事件處理](#) 和 [COM 中的事件處理](#)。

## 另請參閱

[關鍵字](#)

[事件處理](#)

`event_source`

`event_receiver`

`_event`

`_unhook`

`_raise`

# `_raise` 關鍵字

2020/12/10 • [Edit Online](#)

強調事件的呼叫位置。

## NOTE

原生 C++ 中的事件屬性與 Standard C++ 不相容。當您指定一致性模式時，不會編譯它們 `/permissive-`。

## 語法

```
| _raise | method-declarator | ;
```

## 備註

從 managed 程式碼中，只能從其定義所在的類別內引發事件。如需詳細資訊，請參閱 [event](#) (英文)。

`_raise` 如果您呼叫非事件，關鍵字會導致發出錯誤。

## NOTE

樣板類別或結構不能包含事件。

## 範例

```
// 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
}
```

## 另請參閱

[關鍵字](#)

[事件處理](#)

`__event`

`__hook`

`__unhook`

適用於.NET 與 UWP 的元件延伸模組

# `__unhook` 關鍵字

2020/12/10 • [Edit Online](#)

解除處理常式方法與事件的之間的解除。

## NOTE

原生 C++ 中的事件屬性與 Standard C++ 不相容。當您指定一致性模式時，不會編譯它們 `/permissive-`。

## 語法

```
long __unhook(
    &SourceClass::EventMethod,
    source,
    &ReceiverClass::HandlerMethod
    [, receiver = this]
);

long __unhook(
    interface,
    source
);

long __unhook(
    source
);
```

## 參數

`&SourceClass::EventMethod`

事件方法的指標，您會從中解除事件處理常式方法的連結：

- 原生 C++ 事件：`SourceClass` 是事件來源類別，而 `EventMethod` 是事件。
- COM 事件：`SourceClass` 是事件來源介面，而且 `EventMethod` 是它的其中一個方法。
- Managed 事件：`SourceClass` 是事件來源類別，而 `EventMethod` 是事件。

`interface`

要從 接收者 解除掛鉤的介面名稱，僅適用於屬性的 `layout_dependent` 參數為的 COM 事件接收器

`event_receiver` `true`。

`source`

事件來源執行個體的指標。根據 `type` 中指定的程式碼 `event_receiver`，`source` 可以是下列其中一種類型：

- 原生事件來源物件指標。
- `IUnknown` COM 來源) (的指標。
- Managed 物件指標 (用於 Managed 事件)。

`&ReceiverClass::HandlerMethod` 要從事件解除掛鉤的事件處理常式方法指標。此處理程式會指定為類別的方法或相同的參考；如果您未指定類別名稱，則 `__unhook` 會假設類別是呼叫它的類別。

- 原生 C++ 事件：`ReceiverClass` 是事件接收器類別，而 `HandlerMethod` 是處理常式。

- COM 事件: `ReceiverClass` 是事件接收器介面，而且 `HandlerMethod` 是它的其中一個處理常式。
- Managed 事件: `ReceiverClass` 是事件接收器類別，而 `HandlerMethod` 是處理常式。

`receiver` (選擇性) 事件接收器類別實例的指標。如果您未指定接收者，預設值就是呼叫的接收者類別或結構  
`_unhook`。

## 使用方式

可以在任何函式範圍中使用，包括 `main` 事件接收器類別以外的範圍。

## 備註

`_unhook` 在事件接收器中使用內建函數，將處理常式方法與事件方法解除關聯或「解除接收器」。

有三種形式 `_unhook`。大部分情況下可以使用第一種 (四個引數) 形式。您可以針對 COM 事件接收器使用第二個 (兩個引數) 形式的，它會解除 `_unhook` 整個事件介面的掛鉤。您可以使用第三種 (單一引數) 形式從指定的來源解除所有委派的連結。

非零的傳回值表示發生錯誤 (Managed 事件將會擲回例外狀況)。

如果您 `_unhook` 在尚未連結的事件和事件處理常式上呼叫，則不會有任何作用。

在編譯時期，編譯器會驗證事件是否存在，並且對指定的處理常式進行參數類型檢查使。

`_hook` `_unhook` 除了 COM 事件之外，您可以在事件接收器之外呼叫。

使用 `_unhook` 的替代方式是使用`-=` 運算子。

如需以新語法撰寫 managed 事件程式碼的詳細資訊，請參閱 [事件](#)。

### NOTE

樣板類別或結構不能包含事件。

## 範例

如需範例，請參閱 [原生 C++ 中的事件處理](#) 和 [COM 中的事件處理](#)。

## 另請參閱

### 關鍵字

`event_source`  
`event_receiver`  
`_event`  
`_hook`  
`_raise`

# 原生 C++ 中的事件處理

2020/12/10 • [Edit Online](#)

在原生 C++ 事件處理中，您可以分別使用 `event_source` 和 `event_receiver` 屬性來設定事件來源和事件接收器，並指定 `type = native`。這些屬性可讓它們所套用的類別引發事件，並在原生的非 COM 內容中處理事件。

## NOTE

原生 C++ 中的事件屬性與 Standard C++ 不相容。當您指定一致性模式時，不會編譯它們 `/permissive-`。

## 宣告事件

在事件來源類別中，請在 `_event` 方法宣告上使用關鍵字，將此方法宣告為事件。請務必宣告方法，但不要定義方法。如果您這麼做，它會產生編譯器錯誤，因為編譯器會在事件發生時，隱含地定義方法。原生事件可以是包含零個或多個參數的方法。傳回型別可以是 `void` 或任何整數型別。

## 定義事件處理常式

在事件接收器類別中，您會定義事件處理常式。事件處理常式是具有簽章（傳回類型、呼叫慣例和引數的方法，）符合它們將處理的事件。

## 將事件處理常式連結至事件

此外，在事件接收器類別中，您可以使用內建函數，`_hook` 將事件與事件處理常式產生關聯，以及將事件與事件 `_unhook` 處理常式解除關聯。您可以在事件處理常式中攔截多個事件，或在事件中攔截多個事件處理常式。

## 引發事件

若要引發事件，請呼叫在事件來源類別中宣告為事件的方法。如果在事件中攔截到處理常式，則會呼叫處理常式。

## 原生 C++ 事件程式碼

下列範例示範如何在原生 C++ 中引發事件。若要編譯和執行這個範例，請參閱程式碼中的註解。若要在 Visual Studio IDE 中建立程式碼，請確認 `/permissive-` 已關閉該選項。

## 範例

### 程式碼

```

// 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);
}

```

## 輸出

```

MyHandler2 was called with value 123.
MyHandler1 was called with value 123.

```

## 另請參閱

[事件處理](#)

# COM 中的事件處理

2020/12/10 • [Edit Online](#)

在 COM 事件處理中，您可以分別使用和屬性來設定事件來源和事件接收器 `event_source` `event_receiver`，並指定 `type = com`。這些屬性會將適當的程式碼插入自訂、分派和雙重介面。插入的程式碼可讓屬性化類別引發事件，並透過 COM 連接點處理事件。

## NOTE

原生 C++ 中的事件屬性與 Standard C++ 不相容。當您指定一致性模式時，不會編譯它們 `/permissive-`。

## 宣告事件

在事件來源類別中，`_event` 對應介面宣告使用關鍵字，將該介面的方法宣告為事件。當您以介面方法呼叫它們時，會引發該介面的事件。事件介面上的方法可以有零或多個參數（這些參數都應該在）的參數 中。傳回型別可為 `void` 或任何整數類型。

## 定義事件處理常式

您可以在事件接收器類別中定義事件處理常式。事件處理常式是具有簽章（傳回類型、呼叫慣例和引數的方法，）符合它們將處理的事件。針對 COM 事件，呼叫慣例不需要相符。如需詳細資訊，請參閱以下的 [版面配置相依 COM 事件](#)。

## 將事件處理常式連結至事件

此外，在事件接收器類別中，您可以使用內建函數，`_hook` 將事件與事件處理常式產生關聯，以及將事件與事件 `_unhook` 處理常式解除關聯。您可以在事件處理常式中攔截多個事件，或在事件中攔截多個事件處理常式。

## NOTE

通常，有兩種方法可以讓 COM 事件接收器存取事件來源介面定義。第一種是共用常見的標頭檔，如下所示。第二種方式是使用 `#import` 搭配匯 `embedded_idl` 入限定詞，以便將事件來源類型程式庫寫入 .tlh 檔案，並保留屬性產生的程式碼。

## 引發事件

若要引發事件，請呼叫在 `_event` 事件來源類別中使用關鍵字宣告的介面中的方法。如果在事件中攔截到處理常式，則會呼叫處理常式。

### COM 事件碼

下列範例示範如何在 COM 中引發事件。若要編譯和執行這個範例，請參閱程式碼中的註解。

```
// 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;
```

伺服器端的程式碼如下：

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

用戶端的程式碼如下：

```

// 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;
}

```

## 輸出

```

MyHandler1 was called with value 123.
MyHandler2 was called with value 123.

```

## 版面配置相依的 COM 事件

配置相依性是只有 COM 程式設計才有的問題。在原生和 managed 事件處理中，簽章 (傳回型別、呼叫慣例和引

數) 處理常式必須符合它們的事件，但處理常式名稱不一定要與它們的事件相符。

不過，在 COM 事件處理中，當您將的 `Layout_dependent` 參數設定 `event_receiver` 為時 `true`，就會強制執行名稱和簽章比對。事件接收器和已連結事件中處理常式的名稱和簽章必須完全相符。

當 `Layout_dependent` 設定為時 `false`，可以在引發事件方法和攔截方法之間混合和比對呼叫慣例和儲存類別(虛擬、靜態等等，(其委派)。更有效率的作法 `Layout_dependent = true`。

例如，假設已定義的 `IEventSource` 中內含下列方法：

```
[id(1)] HRESULT MyEvent1([in] int value);
[id(2)] HRESULT MyEvent2([in] int value);
```

假設事件來源的格式如下：

```
[coclass, event_source(com)]
class CSource : public IEventSource {
public:
    __event __interface IEvents;

    HRESULT FireEvent() {
        MyEvent1(123);
        MyEvent2(123);
        return S_OK;
    }
};
```

然後，在事件接收器中，所有在 `IEventSource` 的方法中攔截到的處理常式必須符合其名稱和簽章，如下所示：

```
[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
    }
};
```

## 另請參閱

[事件處理](#)

# Microsoft 特定修飾詞

2020/11/2 • [Edit Online](#)

本節將描述下列各層面 Microsoft 專有的 C++ 擴充功能：

- [根據定址](#)，使用指標作為基底的做法，可從中位移其他指標
- [函式呼叫慣例](#)
- 使用 [\\_\\_declspec](#) 關鍵字宣告的擴充儲存類別屬性
- [\\_\\_W64](#)關鍵字

## Microsoft 專有的關鍵字

許多 Microsoft 專有關鍵字可用來將宣告子修改為衍生類型。如需宣告子的詳細資訊，請參閱[宣告子](#)。

宣告子	說明	是否適用？
<a href="#">__based</a>	後面的名稱會將 32 位元位移宣告為宣告中包含的 32 位元基底。	是
<a href="#">__cdecl</a>	後面的名稱會使用 C 命名和呼叫慣例。	是
<a href="#">__declspec</a>	後面的名稱會指定 Microsoft 專有的儲存類別屬性。	否
<a href="#">__fastcall</a>	後面的名稱會將函式宣告為使用暫存器 (如果有的話)，而不使用可進行引數傳遞的堆疊。	是
<a href="#">__restrict</a>	類似于 <a href="#">__declspec</a> ( <a href="#">限制</a> )，但用於變數。	否
<a href="#">__stdcall</a>	後面的名稱會指定採用標準呼叫慣例的函式。	是
<a href="#">__w64</a>	在 64 位元編譯器上將資料類型標示為較大。	否
<a href="#">__unaligned</a>	指出某個類型或其他資料的指標未對齊。	否
<a href="#">__vectorcall</a>	後面的名稱會將函式宣告為只要有暫存器可用即使用暫存器 (包括 SSE 暫存器)，而不使用可進行引數傳遞的堆疊。	是

## 另請參閱

[C++ 語言參考](#)

# 基底定址

2020/3/25 • [Edit Online](#)

本節包含下列主題：

- [\\_based 文法](#)
- [Based 指標](#)

## 另請參閱

[Microsoft 專有的修飾詞](#)

# `_based` 文法

2020/11/2 • [Edit Online](#)

## Microsoft 特定的

基底位址在您需要精確控制配置物件的區段時很實用 (靜態和動態架構資料)。

32位和64位編譯中可接受的唯一定址形式是「以指標為基礎」，其定義一個類型，其中包含對32位或64位基底或以為基礎的32位或64位位置換 `void`。

## 文法

以範圍為基礎的修飾詞: `_based` (基底運算式)

以運算式為基礎的 *variablebased-abstract-declaratorsegment-namesegment-cast*

以變數為基礎: 識別碼

型-抽象-宣告子: 抽象-宣告子

基底類型: 類型-名稱

結束 Microsoft 專有

## 另請參閱

[以指標為基礎](#)

# Based 指標 (C++)

2020/11/2 • [Edit Online](#)

`__based` 關鍵字可讓您根據指標(從現有指標位移的指標)來宣告指標。`__based` 關鍵字是 Microsoft 專有的。

## 語法

```
type __based( base ) declarator
```

## 備註

以指標位址為基礎的指標是 `__based` 關鍵字在32位或64位編譯中有效的唯一形式。對於 Microsoft 32 位元 C/C++ 編譯器，基底指標是來自 32 位元指標基底的 32 位元位移。在 64 位元環境中保留了一個類似的限制，其中基底指標是來自 64 位元基底的 64 位元位移。

以指標為基礎之指標的其中一項用途，就是包含指標的持續性識別項。以指標為基礎的指標所組成的連結清單可以儲存至磁碟，然後以仍然有效的指標重新載入至記憶體中的另一個位置。例如：

```
// based_pointers1.cpp
// compile with: /c
void *vpBuffer;
struct llist_t {
    void __based( vpBuffer ) *vpData;
    struct llist_t __based( vpBuffer ) *llNext;
};
```

為指標 `vpBuffer` 所指派的記憶體位址，會在程式稍後的位置中進行配置。連結清單會重新配置相對於 `vpBuffer` 的值。

### NOTE

保存包含指標的識別碼也可以透過使用記憶體對應檔案來完成。

對基底指標取值時，該基底必須透過宣告明確指定或以隱含方式得知。

為了與舊版相容，`_based` `__based` 除非指定了編譯器選項/z/a 停用 (語言擴充功能，否則 `_based` 是同義字。

## 範例

下列程式碼示範如何變更其基底以變更基底指標。

```
// 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;
}
```

```
1
2
10
11
```

## 另請參閱

[關鍵字](#)

[alloc\\_text](#)

# 呼叫慣例

2020/11/2 • [Edit Online](#)

Visual C/C++ 編譯器提供數個用於呼叫內部和外部函式的不同慣例。了解這些不同的方法可協助您偵錯程式，並將您的程式碼與組合語言常式連結。

與此主旨有關的主題將說明呼叫慣例之間的差異、引數傳遞方式，以及函式傳回值的方式。並且也會探討 naked 函式呼叫，這是一個讓您自行撰寫初構和終解程式碼的進階功能。

如需 x64 處理器呼叫慣例的詳細資訊，請參閱[呼叫慣例](#)。

## 本節主題：

- [引數傳遞和命名慣例](#) (`__cdecl`、`__stdcall`、`__fastcall` 和其他)
- [呼叫範例:函式原型和呼叫](#)
- [使用 naked 函式呼叫撰寫自訂初構/終解程式碼](#)
- [浮點副處理器和呼叫慣例](#)
- [淘汰呼叫慣例](#)

## 另請參閱

[Microsoft 專有的修飾詞](#)

# 引數傳遞和命名慣例

2020/12/10 • [Edit Online](#)

## Microsoft 特定的

Microsoft C++ 編譯器可讓您指定在函數和呼叫端之間傳遞引數和傳回值的慣例。並非所有慣例都適用於所有支援的平台，部分慣例會使用平台特定實作。在大部分情況下，會忽略在特定平台上指定不支援慣例的關鍵字或編譯器參數，並且使用平台預設慣例。

在 x86 平臺上，當傳遞引數時，所有引數都會擴展至 32 位。傳回值也會被擴大為 32 位元並在 EAX 暫存器中傳回，但 8 位元組結構例外，它是在 EDX:EAX 暫存器組中傳回。較大的結構會在 EAX 暫存器中以隱藏傳回結構的指標傳回。參數會從右至左推送至堆疊。非 POD 的結構不會在暫存器中傳回。

如果在函式中使用 ESI、EDI、EBX 和 EBP 暫存器，編譯器會產生初構和終解程式碼來儲存和還原這些暫存器。

### NOTE

從函式以傳值方式傳回結構、等位或類別時，類型的所有定義必須相同，否則程式可能會在執行階段失敗。

如需如何定義自己的函式初構和終解程式碼的詳細資訊，請參閱 [Naked 函式呼叫](#)。

如需以 x64 平臺為目標之程式碼中預設呼叫慣例的詳細資訊，請參閱 [X64 呼叫慣例](#)。如需以 ARM 平臺為目標之程式碼中呼叫慣例問題的詳細資訊，請參閱 [常見的 VISUAL C++ Arm 遷移問題](#)。

Visual C/C++ 編譯器支援下列呼叫慣例。

名稱	說明	備註
<a href="#">_cdecl</a>	呼叫者	以相反順序 (從右至左) 將參數推送至堆疊
<a href="#">_clrcall</a>	n/a	依照順序 (從左至右) 將參數載入至 CLR 運算式堆疊。
<a href="#">_stdcall</a>	被呼叫端	以相反順序 (從右至左) 將參數推送至堆疊
<a href="#">_fastcall</a>	被呼叫端	儲存在暫存器中，然後推送至堆疊
<a href="#">_thiscall</a>	被呼叫端	已推送至堆疊； <code>this</code> 在 ECX 中儲存的指標
<a href="#">_vectorcall</a>	被呼叫端	儲存在暫存器中，然後以相反順序 (從右至左) 推送至堆疊

如需相關資訊，請參閱 [過時呼叫慣例](#)。

結束 Microsoft 專有

另請參閱

[呼叫慣例](#)

# `_cdecl`

2020/11/2 • [Edit Online](#)

`_cdecl` 是 C 和 c++ 程式的預設呼叫慣例。因為堆疊是由呼叫端所清除，所以它可以執行 `vararg` 函數。`_cdecl` 呼叫慣例會建立比 `_stdcall` 更大的可執行檔，因為它會要求每個函式呼叫都包含堆疊清除程式碼。下列清單會顯示這個呼叫慣例的實作。`_cdecl` 修飾詞是 Microsoft 專有的。

II	II
引數傳遞順序	由右至左。
堆疊維護責任	呼叫函式會從堆疊取出引數。
名稱裝飾慣例	底線字元(_)前面會加上名稱，但在 _ 汇出使用 C 連結的 <i>cdecl</i> 函式時除外。
大小寫轉譯慣例	未執行大小寫轉譯。

## NOTE

如需相關資訊，請參閱[裝飾名稱](#)。

將修飾詞放在 `_cdecl` 變數或函數名稱之前。由於 C 命名和呼叫慣例都是預設值，因此，您在 x86 程式碼中唯一必須使用的時間 `_cdecl` 是指定 `/Gv` (vectorcall)、`/Gz` (stdcall) 或 `/Gr` (fastcall) 編譯器選項。`/Gd` 編譯器選項會強制執行 `_cdecl` 呼叫慣例。

在 ARM 和 x64 處理器上，`_cdecl` 會接受但編譯器通常會忽略。ARM 和 x64 的慣例會盡可能在暫存器中傳遞引數，並在堆疊上傳遞後續的引數。在 x64 程式碼中，請使用覆 `_cdecl` 寫 `/Gv` 編譯器選項，並使用預設的 x64 呼叫慣例。

對於非靜態類別函式，如果函式是以非正規的方式定義，則不需要在非正規定義上指定呼叫慣例修飾詞。也就是說，對於類別非靜態成員方法而言，宣告時所指定的呼叫慣例是在定義時假設。如果已指定此類別定義：

```
struct CMyClass {  
    void __cdecl mymethod();  
};
```

這個：

```
void CMyClass::mymethod() { return; }
```

相當於這個：

```
void __cdecl CMyClass::mymethod() { return; }
```

為了與舊版相容，除非指定了 `_cdecl` `cdecl` 編譯器選項/`za` (停用語言擴充功能)，否則 `cdecl` 和 `_cdecl` 是同義字。

## 範例

在下列範例中，編譯器收到的指示是針對 `system` 函式使用 C 命名及呼叫慣例。

```
// 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, ...);
```

## 另請參閱

[引數傳遞和命名慣例](#)

[關鍵字](#)

# `_clrcall`

2020/11/2 • • [Edit Online](#)

指定函式只可以從 Managed 程式碼呼叫。將 `_clrcall` 用於所有只從 managed 程式碼呼叫的虛擬函式。不過，這個呼叫慣例不能用於將會從機器碼呼叫的函式。`__Clrcall`修飾詞是 Microsoft 特定的。

使用 `_clrcall` 可改善從 managed 函式呼叫虛擬 managed 函式時的效能，或從 managed 函式透過指標從 managed 函式至 managed 函數。

進入點是編譯器產生的不同函式。如果函式同時具有原生和 Managed 進入點，則其中一個會是具有函式實作的實際函式。另一個函式將會是呼叫實際函式內部的另一個函式 (Thunk)，並且會讓 Common Language Runtime 執行 `PIvoke`。將函式標示為 `_clrcall` 時，表示函式的執行必須是 MSIL，且不會產生原生進入點函數。

如果未指定 `_clrcall`，取得原生函式的位址時，編譯器會使用原生進入點。`_clrcall` 表示函式是受管理的，而且不需要經過 managed 至原生的轉換。在這種情況下，編譯器會使用 Managed 進入點。

`/clr` 使用 (not `/clr:pure` 或 `/clr:safe`) 時，若未使用 `_clrcall`，則取得函式的位址一律會傳回原生進入點函式的位址。使用 `_clrcall` 時，不會建立原生進入點函式，因此您會取得 managed 函式的位址，而不是進入點 Thunk 函式的位址。如需詳細資訊，請參閱 [雙重 Thunking](#)。`/Clr: pure`和`/clr: safe`編譯器選項在 Visual Studio 2015 中已被取代，Visual Studio 2017 中不支援。

`/clr` (Common Language Runtime 編譯) 暗示所有函式和函式指標都是 `_clrcall` 的，而且編譯器不允許在編譯單位內將函式標記為 `_clrcall`以外的任何功能。當使用 `/clr: pure` 時，`_clrcall` 只能在函式指標和外部宣告上指定。

只要該函式具有 MSIL 執行，您就可以直接從使用 `/clr` 編譯的現有 C++ 程式碼中呼叫 `_clrcall`函式。`_clrcall` 函式不能直接從具有內嵌 asm 的函式呼叫，以及呼叫 CPU 特定 intrinsics，例如，即使這些函式是以編譯的 `/clr`。

`_clrcall` 函式指標只適用於其建立所在的應用程式域中。請使用，而不是在應用程式域之間傳遞 `_clrcall` 函式指標 `CrossAppDomainDelegate`。如需詳細資訊，請參閱 [應用程式域](#)和 [Visual C++](#)。

## 範例

請注意，當使用 `_clrcall`宣告函式時，會在需要時產生程式碼;例如，呼叫函式時。

```

// 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();
}

```

```

in Func1
&Func1 != pf, comparison fails
in Func1
in Func1

```

下列範例說明您可以定義函式指標，這樣就表示您宣告函式指標只能從 Managed 程式碼叫用。如此編譯器就能夠直接呼叫 Managed 函式，並且避開原生進入點(雙 Thunk 問題)。

```

// clrcall3.cpp
// compile with: /clr
void Test() {
    System::Console::WriteLine("in Test");
}

int main() {
    void (*pTest)() = &Test;
    (*pTest)();

    void (__clrcall *pTest2)() = &Test;
    (*pTest2)();
}

```

## 另請參閱

[引數傳遞和命名慣例](#)

[關鍵字](#)

# `_stdcall`

2020/11/2 • [Edit Online](#)

呼叫 `_stdcall` 慣例用於呼叫 WIN32 API 函式。被呼叫端會清除堆疊，因此編譯器會建立函式 `vararg` / `_cdecl`。使用這個呼叫慣例的函式需要函式原型。`_stdcall` 修飾詞是 Microsoft 專有的。

## 語法

傳回類型 \* `_stdcall` \*\*\*函數名稱 ( 引數清單 ) ]

## 備註

下列清單會顯示這個呼叫慣例的實作。

引數傳遞順序	由右至左。
引數傳遞慣例	以傳值方式，除非傳遞指標或參考類型。
堆疊維護責任	被呼叫函式會從堆疊快顯其本身的引數。
名稱裝飾慣例	底線( <code>_</code> )的前面會加上名稱。名稱後面會接著 @ 符號()，@ 後面接著引數清單中的位元組數目(十進位)。因此，宣告為 <code>int func( int a, double b )</code> 的函式會裝飾為如下： <code>_func@12</code>
大小寫轉譯慣例	無

/Gz 編譯器選項 `_stdcall` 會為所有未以不同呼叫慣例明確宣告的函式指定。

為了與舊版相容，`_stdcall` / `_stdcall` 除非指定了編譯器選項 [停用 `/za` ( 語言擴充功能 )]，否則會是同義字。

使用修飾詞所宣告的函式傳回 `_stdcall` 值的方式，與使用宣告的函式相同 `_cdecl`。

在 ARM 和 x64 處理器上，`_stdcall` 編譯器會接受並忽略；在 ARM 和 x64 架構上，依照慣例，引數會盡可能在暫存器中傳遞，而後續引數會在堆疊上傳遞。

對於非靜態類別函式，如果函式是以非正規的方式定義，則不需要在非正規定義上指定呼叫慣例修飾詞。也就是說，對於類別非靜態成員方法而言，宣告時所指定的呼叫慣例是在定義時假設。如果已指定此類別定義，

```
struct CMyClass {  
    void __stdcall mymethod();  
};
```

this

```
void CMyClass::mymethod() { return; }
```

相當於這個

```
void __stdcall CMyClass::mymethod() { return; }
```

## 範例

在下列範例中，會將 `__stdcall` 所有函式類型中的結果當做 `WINAPI` 標準呼叫來處理：

```
// 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, ...);
```

## 另請參閱

[引數傳遞和命名慣例](#)

[關鍵字](#)

# `_fastcall`

2020/11/2 • [Edit Online](#)

## Microsoft 特定的

`_fastcall` 呼叫慣例會指定函式的引數會盡可能在暫存器中傳遞。這個呼叫慣例僅適用於 x86 架構。下列清單會顯示這個呼叫慣例的實作。

引數傳遞順序	引數清單中的前兩個 DWORD 引數或較小引數會由左至右傳入 ECX 和 EDX 暫存器，所有其他引數則由右至左傳遞至堆疊上。
堆疊維護責任	所呼叫的函式會從堆疊取出引數。
名稱裝飾慣例	@ Sign ( @ )的前面會加上名稱; 參數清單中的正負號後面接著位元組數(以十進位表示)，其後綴為名稱。
大小寫轉譯慣例	未執行大小寫轉譯。

### NOTE

未來的編譯器版本可能會使用不同的暫存器來儲存參數。

使用 /Gr 編譯器選項會導致模組中的每個函式編譯為 `_fastcall` 除非使用衝突的屬性來宣告函式，或函數的名稱為 `main`。

`_fastcall` 目標為 ARM 和 x64 架構的編譯器會接受並忽略關鍵字；在 x64 晶片上，依照慣例，前四個引數會盡可能在暫存器中傳遞，並在堆疊上傳遞其他引數。如需詳細資訊，請參閱 [X64 呼叫慣例](#)。在 ARM 晶片上，最多可以在暫存器中傳遞四個整數引數和八個浮點引數，而其他引數則是在堆疊上傳遞。

對於非靜態類別函式，如果函式是以非正規的方式定義，則不需要在非正規定義上指定呼叫慣例修飾詞。也就是說，對於類別非靜態成員方法而言，宣告時所指定的呼叫慣例是在定義時假設。如果已指定此類別定義：

```
struct CMyClass {  
    void __fastcall mymethod();  
};
```

這個：

```
void CMyClass::mymethod() { return; }
```

相當於這個：

```
void __fastcall CMyClass::mymethod() { return; }
```

為了與舊版相容，`_fastcall` `__fastcall` 除非指定了編譯器選項 /za 停用 (語言擴充) 功能，否則 `_fastcall` 是同義字。

## 範例

下列範例會對暫存器中的 `DeleteAggrWrapper` 函式傳遞引數：

```
// 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, ...);
```

結束 Microsoft 專有

## 另請參閱

[引數傳遞和命名慣例](#)

[關鍵字](#)

# `_thiscall`

2020/12/10 • [Edit Online](#)

Microsoft 特定的 `_thiscall` 呼叫慣例用於 x86 架構上的 C++ 類別成員函式。這是不使用變數引數 (函式) 的成員函式所使用的預設呼叫慣例 `vararg`。

在下 `_thiscall`，被呼叫端會清除堆疊，而不是函式的可能 `vararg`。引數會從右至左推入堆疊。`this` 指標是透過 REGISTER ECX 傳遞，而不是在堆疊上。

在 ARM、ARM64 和 x64 電腦上，`_thiscall` 編譯器會接受並忽略。這是因為它們預設會使用以暫存器為基礎的呼叫慣例。

使用的其中一個原因 `_thiscall` 是預設使用成員函式的類別 `_clrcall`。在這種情況下，您可以使用 `_thiscall` 來讓個別成員函式可從機器碼呼叫。

使用進行編譯時，所有函式和函式 `/clr:pure` 指標都是，`_clrcall` 除非另有指定。`/clr:pure` 和 `/clr:safe` 編譯器選項在 Visual Studio 2015 中已被取代，Visual Studio 2017 中不支援。

`vararg` 成員函式會使用 `_cdecl` 呼叫慣例。所有函式引數都會推送到堆疊上，`this` 最後會將指標放在堆疊上。

因為此呼叫慣例僅適用於 C++，所以不會有 C 名稱裝飾配置。

當您定義非靜態類別成員函式的行時，請只在宣告中指定呼叫慣例修飾詞。您不需要在非正規定義上再次指定它。編譯器會在定義時使用宣告期間所指定的呼叫慣例。

## 請參閱

[引數傳遞和命名慣例](#)

# `_vectorcall`

2020/11/2 • [Edit Online](#)

## Microsoft 特定的

`_vectorcall` 呼叫慣例會指定函式的引數會盡可能在暫存器中傳遞。`_vectorcall` 使用引數的多個暫存器，而不是`_fastcall` 使用預設的x64 呼叫慣例。`_vectorcall` 只有在包含 Streaming SIMD Extensions 2 (SSE2) 和更新版本的 x86 和 x64 處理器上，機器碼才支援呼叫慣例。使用 `_vectorcall` 來加速傳遞數個浮點或 SIMD 向量引數的函式，並執行可利用暫存器中載入之引數的作業。下列清單顯示 x86 和 x64 執行的通用功能 `_vectorcall`。這些差異稍後會在本文中加以說明。

II	II
C 名稱裝飾慣例	函式名稱前面會加上兩個 "at" 符號 ( @ @ )，後面接著參數清單中的位元組數目(十進位)。
大小寫轉譯慣例	不會執行大小寫轉譯。

使用 `/Gv` 編譯器選項會導致模組中的每個函式編譯為 `_vectorcall`，除非該函式為成員函式、以衝突的呼叫慣例屬性宣告、使用 `vararg` 變數引數清單，或具有名稱 `main`。

您可以藉由在函式中註冊來傳遞三種引數 `_vectorcall`：整數類型值、向量類型值和同質向量匯總(HVA)值。

整數類型符合兩個需求：它符合處理器的原生暫存器大小 (例如，在 x86 電腦上為 4 位元組，在 x64 電腦上為 8 位元組)，且可轉換為各種暫存器長度的整數，而不會變更它的位元表示。例如，可以 `int` 在 x86 (在 x64 上) 升級為任何類型 (例如 `long long` `char` 或 `short`)，或者可以轉換成 `int` (`long long` 在 x64 上) 並回到其原始類型，而不需要變更為整數類型。整數類型包括指標、參考，以及 `struct` `union` 4 個位元組 (x64 上為 8 個位元組) 或更少的類型。在 x64 平臺上，較大的 `struct` 和 `union` 類型會以傳址方式傳遞至呼叫端所配置的記憶體；在 x86 平臺上，它們會在堆疊上以傳值方式傳遞。

向量類型可以是浮點數類型 (例如 `float` 或 `double`) 或 SIMD 向量類型 (例如 `_m128` 或) `_m256`。

HVA 類型是一種複合類型，具有四個相同向量類型的資料成員。HVA 類型與其成員的向量類型具有相同的對齊需求。這是 HVA `struct` 定義的範例，其中包含三個相同的向量類型，且具有 32 位元組對齊：

```
typedef struct {
    _m256 x;
    _m256 y;
    _m256 z;
} hva3; // 3 element HVA type on _m256
```

`_vectorcall` 在標頭檔中使用關鍵字明確宣告函式，以允許另行編譯的程式碼連結而不會發生錯誤。函式必須是原型才能使用 `_vectorcall`，而且不能使用 `vararg` 可變長度的引數清單。

成員函式可以使用規範來宣告 `_vectorcall`。隱藏的 `this` 指標會由 `register` 當做第一個整數類型引數傳遞。

在 ARM 機器上，`_vectorcall` 編譯器會接受並忽略。

對於非靜態類別成員函式，如果函式是以非正規的方式定義，則不需要在非正規定義上指定呼叫慣例修飾詞。也就是說，對於類別非靜態成員而言，宣告時所指定的呼叫慣例是在定義時假設。如果已指定此類別定義：

```
struct MyClass {  
    void __vectorcall mymethod();  
};
```

這個：

```
void MyClass::mymethod() { return; }
```

相當於這個：

```
void __vectorcall MyClass::mymethod() { return; }
```

建立函式 `__vectorcall` 的指標時，必須指定呼叫慣例修飾詞 `__vectorcall`。下一個範例會建立函式指標的，該函式 `typedef __vectorcall` 會接受四個 `double` 引數並傳回 `_m256` 值：

```
typedef _m256 (__vectorcall * vcfnptr)(double, double, double, double);
```

為了與舊版相容，`__vectorcall` `__vectorcall` 除非指定了編譯器選項 [停用 [/za](#) (語言擴充功能)]，否則會是同義字。

## 在 x64 上的 `__vectorcall` 慣例

`__vectorcall` X64 上的呼叫慣例會擴充標準 x64 呼叫慣例，以利用額外的暫存器。整數型別引數和向量型別引數都是根據引數清單中的位置對應到暫存器。HVA 引數會配置給未使用的向量暫存器。

當前四個引數（由左至右的順序）的任何一個為整數型別引數時，會在對應於該位置的暫存器（RCX、RDX、R8 或 R9）中傳遞。隱藏的 `this` 指標會被視為第一個整數類型引數。當 HVA 引數（前四個引數的其中一個）無法在可用暫存器中傳遞時，呼叫端配置記憶體的參考會在對應的整數類型暫存器中傳遞。第四個參數位置之後的整數型別引數會在堆疊上傳遞。

當前六個引數（由左至右的順序）的任何一個為向量類型引數時，會根據引數位置，在 SSE 向量暫存器 0 到 5 中以傳值方式傳遞。浮點和 `_m128` 類型會在 XMM 暫存器中傳遞，而 `_m256` 類型則會在 YMM 暫存器中傳遞。這不同於標準的 x64 呼叫慣例，因為向量類型是以傳值方式傳遞而不是以傳址方式傳遞，因此會使用其他的暫存器。為向量類型引數配置的陰影堆疊空間固定為 8 個位元組，而 [/homeparams](#) 選項則不適用。在第七個和後面的參數位置上的向量類型引數，是在堆疊上以傳址方式傳遞到呼叫端所配置的記憶體。

針對向量引數配置暫存器之後，只要有足夠的暫存器可供整個 YMM5 使用，HVA 引數的資料成員就會以遞增的順序配置給未使用的向量暫存器 XMM0 至 XMM5（或 YMM0 至 HVA，適用於 `_m256` 類型）。如果沒有足夠的暫存器可供使用，HVA 引數會以傳址方式傳遞至呼叫端所配置的記憶體。HVA 引數的堆疊陰影空間會固定為 8 個位元組，且具有未定義的內容。HVA 引數是以參數清單中由左至右的順序指派給暫存器，而且可以位於任何位置。位於未指派給向量暫存器的前四個引數位置其中之一的 HVA 引數，會在對應至該位置的整數暫存器中以傳址方式傳遞。在第四個參數位置之後以傳址方式傳遞的 HVA 引數會推送至堆疊上。

函式的結果 `__vectorcall` 會盡可能在暫存器中以傳值方式傳回。整數類型（包括 8 位元組或較少的結構或等位）的結果會在 RAX 中以傳值方式傳回。視大小而定，向量類型的結果是以傳值方式在 XMM0 或 YMM0 中傳回。HVA 結果的每個資料項目會視大小而定，以傳值方式從暫存器 XMM0:XMM3 或 YMM0:YMM3 中傳回。對應暫存器無法容納的結果類型會以傳址方式傳回至呼叫端配置的記憶體。

堆疊是由的 x64 執行中的呼叫者維護 `__vectorcall`。呼叫端初構和終解程式碼會配置並清除所呼叫函式的堆疊。堆疊上的引數會從右至左推入，並且為暫存器中傳遞的引數配置陰影堆疊空間。

範例：

```

// 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 <xmmmintrin.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 慣例

`__vectorcall` 呼叫慣例 `__fastcall` 會遵循32位整數類型引數的慣例，並利用 SSE 向量暫存器來進行向量類型和 HVA 引數。

存在於參數清單中的前兩個整數類型引數，會由左至右分別放置在 ECX 和 EDX 中。隱藏的 `this` 指標會被視為第一個整數型別引數，並在 ECX 中傳遞。前六個向量型別引數會透過 SSE 向量暫存器 0 到 5，或者在 YMM 或 XMM 暫存器中（視引數大小而定），以傳值方式傳遞。

前六個向量型別引數依序由左至右，在 SSE 向量暫存器 0 到 5 中，以傳值方式傳遞。浮點和 `__m128` 類型會在 XMM 暫存器中傳遞，而 `__m256` 類型則會在 YMM 暫存器中傳遞。陰影堆疊空間不會配置給暫存器所傳遞的向量型別引數。第七個和後續向量類型引數，會在堆疊上以傳址方式傳遞至呼叫端配置的記憶體。編譯器錯誤 C2719 的限制不適用於這些引數。

針對向量引數配置暫存器之後，只要有足夠的暫存器可供整個 YMM5 使用，HVA 引數的資料成員就會以遞增順序配置給未使用的向量暫存器 XMM0，以 XMM5（或 YMM0 至 HVA，適用於 `__m256` 類型）。如果沒有足夠的暫存器可供使用，HVA 引數會在堆疊上以傳址方式傳遞至呼叫端所配置的記憶體。不會配置 HVA 引數的陰影堆疊空間。HVA 引數是以參數清單中由左至右的順序指派給暫存器，而且可以位於任何位置。

函式的結果 `__vectorcall` 會盡可能在暫存器中以傳值方式傳回。整數類型（包括 4 位元組或較少的結構或等位）的結果會在 EAX 中以傳值方式傳回。8 位元組或較少的整數類型結構或等位會在 EDX:EAX 中，以傳值方式傳回。視大小而定，向量類型的結果是以傳值方式在 XMM0 或 YMM0 中傳回。HVA 結果的每個資料項目會視大小而定，以傳值方式從暫存器 XMM0:XMM3 或 YMM0:YMM3 中傳回。其他結果類型是以傳址方式，由呼叫端所配置的記憶體傳回。

的 x86 執行 `__vectorcall` 遵循由呼叫端從右至左推送至堆疊的引數慣例，而被呼叫的函式會在傳回之前清除堆疊。只有未放置在暫存器上的引數會推入至堆疊。

範例：

```

// 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 <xmmmintrin.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);
}
```

結束 Microsoft 專有

## 另請參閱

[引數傳遞和命名慣例](#)

[關鍵字](#)

# 呼叫範例：函式原型和呼叫

2020/3/25 • [Edit Online](#)

## Microsoft 專屬

下列範例示範使用各種呼叫慣例執行函式呼叫的結果。

這個範例是以下列函式架構為基礎。以適當的呼叫慣例取代 `calltype`。

```
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);
```

如需詳細資訊，請參閱[呼叫範例的結果](#)。

END Microsoft 特定的

## 另請參閱

[呼叫慣例](#)

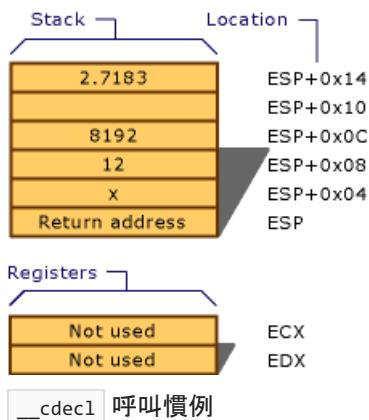
# 呼叫範例的結果

2020/11/2 • [Edit Online](#)

Microsoft 特定的

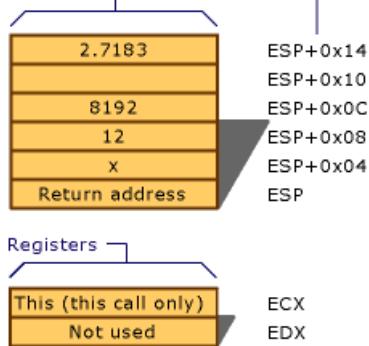
## \_cdecl

C 裝飾函數名稱是 `_MyFunc`。



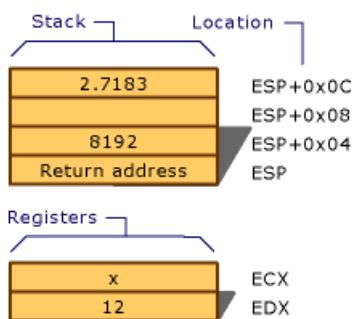
## \_stdcall 和 thiscall

C 裝飾名稱(`_stdcall`)為 `_MyFunc@20`。C++ 裝飾名稱是實作為特定的。



## \_fastcall

C 裝飾名稱(`_fastcall`)為 `@MyFunc@20`。C++ 裝飾名稱是實作為特定的。



## \_fastcall 呼叫慣例

結束 Microsoft 專有

**另請參閱**

[呼叫範例:函式原型和呼叫](#)

# Naked 函式呼叫

2020/11/2 • [Edit Online](#)

## Microsoft 特定的

以屬性宣告的函式會在沒有初構或終解程式 `naked` 代碼的情況下發出，讓您使用 [內嵌](#) 組合語言撰寫自己的自訂初構/終解序列。Naked 函式是做為進階功能提供。這些函式可讓您宣告從 C/C++ 以外的內容呼叫的函式，因此對於參數位置以及要保留哪些暫存器會做出不同的假設。範例包括像是中斷處理常式這類常式。這項功能對於虛擬裝置驅動程式 (VxD) 的撰寫者特別實用。

## 您還想知道關於哪些方面的詳細資訊？

- [naked](#)
- [Naked 函式的規則和限制](#)
- [撰寫初構/終解程式碼的考慮](#)

結束 Microsoft 專有

## 另請參閱

[呼叫慣例](#)

# Naked 函式的規則和限制

2020/11/2 • [Edit Online](#)

## Microsoft 特定的

下列規則和限制適用於 naked 函式：

- `return` 不允許使用語句。
- 不允許結構化例外狀況處理和 C++ 例外狀況處理建構，因為它們必須跨堆疊框架回溯。
- 基於相同理由，亦不得使用任何形式的 `setjmp`。
- 禁止使用 `_alloca` 函式。
- 為確保初構序列之前不會出現區域變數的初始化程式碼，函式範圍不可使用初始化的區域變數。尤其不允許在函式範圍宣告 C++ 物件。不過，巢狀範圍中可以有初始化資料。
- 不建議使用框架指標最佳化 (/Oy 編譯器選項)，但 naked 函式會自動隱藏此最佳化。
- 您不能在函式語彙範圍宣告 C++ 類別物件。不過，您可以在巢狀區塊中宣告物件。
- `naked` 使用 /clr 進行編譯時，會忽略關鍵字。
- 對於 `_fastcall` naked 函式，只要 C/C++ 程式碼中的其中一個暫存器引數有參考，初構程式碼就應該將該暫存器的值儲存在該變數的堆疊位置中。例如：

```
// 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
    }
}
```

結束 Microsoft 專有

另請參閱

[Naked 函式呼叫](#)

# 撰寫初構/終解程式碼的考量

2020/4/15 • • [Edit Online](#)

## Microsoft 特定的

在編寫自己的 prolog 和 epilog 代碼序列之前，瞭解堆疊幀的佈局非常重要。瞭解如何使用 `__LOCAL_SIZE` 符號也很有用。

## 堆疊幀佈局

這個範例將示範可能出現在 32 位元函式中的標準初構程式碼：

```
push    ebp          ; Save ebp
mov     ebp, esp      ; Set stack frame pointer
sub     esp, localbytes ; Allocate space for locals
push    <registers>   ; Save registers
```

`localbytes` 變數代表區域變數堆疊上所需的位元組數目，而 `<registers>` 變數是預留位置，代表要儲存在堆疊上的暫存器清單。推出暫存器之後，您就可以在堆疊上放置任何適當的資料。以下是對應的終解程式碼：

```
pop    <registers>   ; Restore registers
mov     esp, ebp      ; Restore stack pointer
pop    ebp           ; Restore ebp
ret               ; Return from function
```

堆疊一律從高到低排列（從高到低記憶體位址）。基底指標（`ebp`）會指向 `ebp` 的推送值。區域是從 `ebp-4` 開始。若要存取區域變數，請從 `ebp` 減去適當的值計算 `ebp` 的位移。

## `__LOCAL_SIZE`

編譯器提供一個符號，`__LOCAL_SIZE` 用於函數 prolog 代碼的內聯彙編器塊。這個符號是用來在自訂初構程式碼中堆疊框架上為區域變數配置空間。

編譯器確定的 `__LOCAL_SIZE` 值。其值是所有使用者定義之區域變數和編譯器所產生之暫存變數的位元組總數。

`__LOCAL_SIZE` 只能用作即時操作；它不能在表達式中使用。您不得變更或重新定義這個符號的值。例如：

```
mov    eax, __LOCAL_SIZE        ; Immediate operand--Okay
mov    eax, [ebp - __LOCAL_SIZE] ; Error
```

以下包含自訂 prolog 和 epilog 序列的裸函 `__LOCAL_SIZE` 式範例使用 prolog 序列中的符號：

```
// 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
    }
}
```

結束微軟的

另請參閱

[Naked 函式呼叫](#)

# 浮點常數副處理器和呼叫慣例

2020/11/2 • • [Edit Online](#)

如果您要撰寫浮點副處理器的元件常式，必須保留浮點控制字組並清除副處理器堆疊，除非您要傳回 `float` 或 `double` 值（您的函式應該會在 ST (0) 中傳回）。

## 另請參閱

[呼叫慣例](#)

# 過時呼叫慣例

2020/3/25 • [Edit Online](#)

## Microsoft 專屬

不再支援 \_\_pascal、\_\_fortran 和 \_\_syscall 呼叫慣例。您可以使用其中一種支援的呼叫慣例和適當的連結器選項模擬其功能。

<的 windows.h > 現在支援 WINAPI 宏，它會轉譯為適合目標的呼叫慣例。使用您先前使用 PASCAL 或 \_\_far \_\_pascal 的 WINAPI。

END Microsoft 特定的

## 另請參閱

[引數傳遞和命名慣例](#)

# restrict (C++ AMP)

2020/11/2 • • [Edit Online](#)

限制規範可以套用到函式和 Lambda 声明。它会在函式中的程式碼上強制执行限制，以及在使用 C++ Accelerated Massive Parallelism (C++ AMP) 的應用程式中函式的行為上強制执行限制。

## NOTE

如需 `restrict` 屬於儲存類別屬性一部分之關鍵字的詳細資訊 `__declspec`，請參閱[restrict](#)。

`restrict` 子句會採用下列形式：

如果	如果
<code>restrict(cpu)</code>	函式可以使用完整的 C++ 語言。只有使用 <code>restrict(cpu)</code> 函式宣告的其他函式可以呼叫函式。
<code>restrict(amp)</code>	函式只能使用 C++ AMP 可以加速之 C++ 語言的子集。
<code>restrict(cpu)</code> 和 <code>restrict(amp)</code> 的序列。	函式必須同時符合 <code>restrict(cpu)</code> 和 <code>restrict(amp)</code> 的限制。函式可由使用 <code>restrict(cpu)</code> 、 <code>restrict(amp)</code> 、 <code>restrict(cpu, amp)</code> 或 <code>restrict(amp, cpu)</code> 告知的函式呼叫。  <code>restrict(A) restrict(B)</code> 格式可以撰寫為 <code>restrict(A,B)</code> 。 。

## 備註

`restrict` 關鍵字是內容關鍵字。限制規範、`cpu` 和 `amp` 不是保留字。規範的清單無法擴充。沒有子句的函數與具有子句的函式 `restrict` 相同 `restrict(cpu)`。

內含 `restrict(amp)` 子句的函式具有下列限制：

- 函式只會呼叫內含 `restrict(amp)` 子句的函式。
- 函式必須為可內嵌。
- 函式只能宣告 `int`、`unsigned int`、`float` 和 `double` 變數，以及只包含這些類型的類別和結構。`bool` 也允許，但如果您在複合類型中使用，它必須是4位元組對齊。
- Lambda 函式無法透過參考方式擷取，也無法擷取指標。
- 參考和單一間接取值指標只支援區域變數、函式引數和傳回型別。
- 不允許使用下列各項：
  - 遷迴。
  - 以 `volatile` 關鍵字宣告的變數。
  - 虛擬函式。
  - 函式的指標。

- 成員函式的指標。
- 結構中的指標。
- 指標的指標。
- `goto` 報表。
- 標記陳述式。
- `try`、`catch` 或 `throw` 語句。
- 全域變數。
- 靜態變數。請改用[`Tile\_static` 關鍵字](#)。
- `dynamic_cast` 廣播。
- `typeid` 運算子。
- `asm` 告白。
- `Varargs`。

如需函數限制的討論，請參閱[限制\(amp\)限制](#)。

## 範例

下列範例顯示如何使用 `restrict(amp)` 子句。

```
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();
}
```

## 另請參閱

[C++ AMP \(C++ Accelerated Massive Parallelism\)](#)

# tile\_static 關鍵字

2020/3/25 • [Edit Online](#)

Tile\_static 關鍵字是用來宣告可供執行緒磚中的所有線程存取的變數。變數的存留期從執行達到宣告點時開始，並在核心函式傳回時結束。如需使用磚的詳細資訊，請參閱[使用磚](#)。

Tile\_static 關鍵字具有下列限制：

- 只能用於具有 `restrict(amp)` 修飾詞之函式中的變數。
- 不可用於屬於指標或參考類型的變數。
- Tile\_static 變數不能有初始化運算式。不會自動叫用預設建構函式和解構函式。
- 未初始化的 tile\_static 變數的值未定義。
- 如果 tile\_static 變數宣告于呼叫圖形中，而該圖表是以 `parallel_for_each` 的非磚式呼叫為基礎，則會產生警告，而且不會定義變數的行為。

## 範例

下列範例顯示如何使用 tile\_static 變數，在磚中的多個執行緒之間累積資料。

```
// 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,
    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
```

## 另請參閱

[Microsoft 專有的修飾詞](#)

[C++ AMP 概觀](#)

[parallel\\_for\\_each 函式C++ \(AMP\)](#)

[逐步解說:矩陣乘法](#)

## `_declspec`

2020/11/2 • [Edit Online](#)

### Microsoft 特定的

用於指定儲存類別資訊的擴充屬性語法會使用 `_declspec` 關鍵字，這會指定特定類型的實例要與下列 Microsoft 特有的儲存類別屬性一起儲存。其他儲存類別修飾詞的範例包括 `static` 和 `extern` 關鍵字。不過，這些關鍵字是 C 和 C++ 語言的 ANSI 規格的一部分，因此擴充屬性語法未涵蓋它們。擴充屬性語法可簡化並標準化 Microsoft 專有的 C 和 C++ 語言擴充功能。

## 文法

```
decl-specifier :  
    _declspec ( extended-decl-modifier-seq )  
  
extended-decl-modifier-seq :  
    extended-decl-modifier 選擇性  
    extended-decl-modifier extended-decl-modifier-seq  
  
extended-decl-modifier :  
    align( 數位 )  
    allocate(" segname ")  
    allocator  
    appdomain  
    code_seg(" segname ")  
    deprecated  
    dllimport  
    dllexport  
    jit intrinsic  
    naked  
    noalias  
    noinline  
    noreturn  
    noexcept  
    novtable  
    process  
    property( { get= get-func-name | ,put= put-name } )  
    restrict  
    safebuffers  
    selectany  
    spectre(nomitigation)  
    thread  
    uuid(" ComObjectGUID ")
```

空白字元會分隔宣告修飾詞序列。範例會在後面的章節中顯示。

擴充屬性文法支援下列 Microsoft 特有的儲存類別屬性：`align`、`allocate`、`allocator`、`appdomain`、`code\_seg`、`deprecated`、`dllexport`、`dllimport`、`jit intrinsic`、`naked`、`noalias`、`noinline`、`noreturn`、`nothrow`、`novtable`、`process`、`restrict`、`safebuffers`、`selectany`、`spectre` 和 `thread`。

它也支援下列 COM 物件屬性：`property` 和 `uuid`。

`code\_seg`、`dllexport`、`dllimport`、`naked`、`noalias`、`nothrow`、`property`、`restrict`、`selectany`、`thread` 和 `uuid` 儲存類別屬性僅為套用它們之物件或函式的宣告屬性。`thread` 屬性只會影響資料和物件。`naked` 和 `spectre` 屬性只會影響函式。`dllimport` 和 `dllexport` 屬性會影響函數、資料和物件。`property`、`selectany` 和 `uuid` 屬性會影響 COM 物件。

為了與舊版相容，`_declspec` 除非指定了編譯器選項 /za (停用語言擴充功能)，否則會是同義字。

`_declspec` 關鍵字應該放在簡單宣告的開頭。編譯器會忽略在宣告 `_declspec` 中的 \* 或 & 後面加上變數識別碼前方的任何關鍵字，而不會發出警告。

`_declspec` 在使用者自訂類型宣告的開頭指定的屬性會套用至該類型的變數。例如：

```
_declspec(dllexport) class X {} varX;
```

在本案例中，屬性會套用至 `varX`。在 `_declspec` 或關鍵字之後放置的屬性會 `class`、`struct` 套用至使用者定義型別。例如：

```
class _declspec(dllexport) X {};
```

在本案例中，屬性會套用至 `X`。

針對簡單宣告使用屬性的一般指導方針如下所示 `_declspec`：

*extended-decl-modifier-seq-規範-seq init-宣告子-list*,

*Extended-decl-modifier-seq 規範-seq* 應該包含基底型別(例如`int`、`float`、`typedef` 或類別名稱)、儲存類別(例如 `static`、`extern`)或 `_declspec` 擴充功能，其中包括其他專案。在其他專案中，`init` 廣告子清單應該包含廣告的指標部分。例如：

```
_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
```

下列程式碼宣告整數執行緒區域變數，並使用值將它初始化：

```
// Example of the _declspec keyword
_declspec( thread ) int tls_i = 1;
```

結束 Microsoft 專有

另請參閱

關鍵字

C 擴充的儲存類別屬性

# align (C++)

2020/11/2 • [Edit Online](#)

在 Visual Studio 2015 和更新版本中，請使用 C++ 11 標準  `規範來控制對齊。如需詳細資訊，請參閱對齊。`

## Microsoft 特定的

使用 `__declspec(align(#))` 可精確控制使用者定義資料的對齊（例如，靜態配置或函式中的自動資料）。

## 語法

```
__declspec (align ( #)) declarator 告訴子
```

## 備註

撰寫使用最新處理器指令的應用程式會帶來一些新的限制和問題。許多新的指示需要與 16 位元組界限對齊的資料。此外，藉由將經常使用的資料與處理器的快取行大小進行對齊，您可以改善快取效能。例如，如果您定義的結構的大小小於 32 個位元組，您可能會想要 32 位元組對齊，以確保能夠有效率地快取該結構類型的物件。

#這是對齊值。有效項目是從 1 至 8192 (位元組) 的 2 乘幕整數，例如 2、4、8、16、32 或 64。`declarator` 這是您要宣告為對齊的資料。

如需如何傳回類型值（`size_t` 這是類型的對齊需求）的詳細資訊，請參閱 [alignof](#)。如需如何在以 64 位處理器為目標時宣告未對齊指標的詳細資訊，請參閱 [unaligned](#)。

`__declspec(align(#))` 當您定義 `struct`、`union` 或 `class` 時，或在宣告變數時，可以使用。

在複製或資料轉換作業期間，編譯器不保證或會嘗試保留資料的對齊屬性。例如，`memcpy` 可以將使用宣告的結構複製 `__declspec(align(#))` 到任何位置。一般配置器（例如，`malloc`、C++ `operator new` 和 Win32 配置器）通常會傳回對 `__declspec(align(#))` 結構或結構陣列而言不夠的記憶體。若要保證複製或資料轉換作業的目的地正確對齊，請使用 `_aligned_malloc`。或者，撰寫您自己的配置器。

您無法指定函數參數的對齊方式。當您以堆疊上的值來傳遞具有對齊屬性的資料時，其對齊方式會由呼叫慣例控制。如果資料對齊在呼叫的函式中很重要，請將參數複製到正確對齊的記憶體中，才使用該參數。

`__declspec(align(#))` 若沒有，編譯器通常會根據目標處理器和資料大小，將資料對齊自然界限，在 32 位處理器上最多 4 位元組的界限，64 位處理器上的 8 位元組界限。類別或結構中的資料會以其自然對齊和目前封裝設定（來自 `#pragma pack` 或編譯器選項）的最小程度，在類別或結構中對齊 `/zp`。

這個範例會示範 `__declspec(align(#))` 的使用方式。

```
__declspec(align(32)) struct Str1{
    int a, b, c, d, e;
};
```

這個類型現在具有 32 位元組對齊屬性。這表示所有的靜態和自動實例都是在 32 位元組的界限上啟動。以這個類型做為成員宣告的其他結構類型會保留這個類型的對齊屬性，也就是說，任何以 `Str1` 元素做為專案的結構都具有至少 32 的對齊屬性。

在這裡，`sizeof(struct Str1)` 等於 32。這表示如果建立物件的陣列 `Str1`，而且陣列的基底為 32 位元組對齊，則陣列的每個成員也會對齊 32 位元組。若要在動態記憶體中建立基底正確對齊的陣列，請使用 `_aligned_malloc`。

或者，撰寫您自己的配置器。

`sizeof` 任何結構的值都是最後一個成員的位移，加上該成員的大小，進位到最大成員對齊值的最接近倍數，或整個結構對齊值(以較大者為準)。

編譯器會為結構對齊使用這些規則：

- 除非以 `__declspec(align(#))` 覆寫，否則純量結構成員的對齊是其大小與目前封裝的最小值。
- 除非以 `__declspec(align(#))` 覆寫，否則結構的對齊是其成員的個別對齊最大值。
- 結構成員位於其父結構開頭的位移，這是其對齊的最小倍數大於或等於前一個成員結尾的位移。
- 結構的大小是其對齊的最小倍數(大於或等於其最後一個成員結尾的位移)。

`__declspec(align(#))` 只能增加對齊限制。

如需詳細資訊，請參閱

- [align 典型](#)
- [使用定義新類型 `\_\_declspec\(align\(#\)\)`](#)
- [對齊執行緒區域儲存區中的資料](#)
- [如何處理 `align` 資料封裝](#)
- [結構對齊範例\(x64 專用\)](#)

## 對齊範例

下列範例說明 `__declspec(align(#))` 如何影響資料結構的大小和對齊。這個範例會假設下列定義：

```
#define CACHE_LINE 32
#define CACHE_ALIGN __declspec(align(CACHE_LINE))
```

在這個範例中，`S1` 結構使用 `__declspec(align(32))` 定義。針對變數定義使用的所有 `S1`，或其他類型宣告中，都是對齊 32 位元組。`sizeof(struct S1)` 會傳回 32，而且 `S1` 在保留四個整數所需的 16 個位元組後面會有 16 個填補位元組。每個 `int` 成員需要 4 位元組對齊，但是結構本身的對齊會宣告為 32。然後，整體的對齊方式為 32。

```
struct CACHE_ALIGN S1 { // cache align all instances of S1
    int a, b, c, d;
};

struct S1 s1; // s1 is 32-byte cache aligned
```

在下列範例中，`sizeof(struct S2)` 會傳回 16，也就是成員大小的總和，因為這剛好是最大對齊需求的倍數(8 倍)。

```
__declspec(align(8)) struct S2 {
    int a, b, c, d;
};
```

在下列範例中，`sizeof(struct S3)` 會傳回 64。

```

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
};

```

在這個範例中，請注意 `a` 有其自然類型的對齊，在此情況下，為 4 個位元組。不過，`s1` 必須對齊 32 位元組。會跟隨 28 個位元組的填補 `a`，因此 `s1` 從位移 32 開始。`s4` 接著會繼承的對齊需求 `s1`，因為它是結構中最大的對齊需求。`sizeof(struct S4)` 會傳回 64。

```

struct S4 {
    int a;
    // 28 bytes padding
    struct S1 s1; // S4 inherits cache alignment requirement of S1
};

```

下列三個變數宣告也會使用 `__declspec(align(#))`。在每個案例中，變數必須對齊 32 位元組。在陣列中，陣列的基底位址(而不是每個陣列成員)會對齊 32 位元組。`sizeof` 當您使用時，每個陣列成員的值不會受到影響 `__declspec(align(#))`。

```

CACHE_ALIGN int i;
CACHE_ALIGN int array[128];
CACHE_ALIGN struct s2 s;

```

若要對齊陣列的每個成員，應使用如下的程式碼：

```

typedef CACHE_ALIGN struct { int a; } S5;
S5 array[10];

```

在這個範例中，請注意對齊結構本身和對齊第一個元素都有相同的效果：

```

CACHE_ALIGN struct S6 {
    int a;
    int b;
};

struct S7 {
    CACHE_ALIGN int a;
    int b;
};

```

`S6` 和 `S7` 具有相同的對齊、配置和大小特性。

在下列範例中，`a`、`b`、`c` 和 `d` 的對齊開始位址分別為 4、1、4 和 1。

```

void fn() {
    int a;
    char b;
    long c;
    char d[10]
}

```

如果已在堆積上配置記憶體，則對齊會取決於所呼叫的配置函式。例如，如果您使用 `malloc`，則結果會取決於運算元大小。如果 `arg >= 8`，則傳回的記憶體會對齊 8 位元組。如果 `arg < 8`，則傳回的記憶體的對齊會是小於 `arg` 之 2

的第一個乘幕。例如，如果您使用 `malloc(7)`，則對齊是4個位元組。

## 使用定義新類型 `__declspec(align(#))`

您可以定義具有對齊特性的類型。

例如，您可以透過下列 `struct` 方式定義具有對齊值的：

```
struct aType {int a; int b;};
typedef __declspec(align(32)) struct aType bType;
```

現在 `aType` 和 `bType` 都是相同的大小(8個位元組)，但類型的變數 `bType` 會對齊32位元組。

## 對齊線程區域儲存區中的資料

使用 `__declspec(thread)` 屬性建立並放入影像之 TLS 區段的靜態執行緒區域儲存區 (TLS) 可用於對齊，就像一般靜態資料一樣。為了建立 TLS 資料，作業系統會配置記憶體的 TLS 區段大小，並接受 TLS 區段對齊屬性。

此範例示範各種將對齊的資料放入執行緒區域儲存區中的方式。

```
// 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;
```

## 如何處理 `align` 資料封裝

`/Zp` 編譯器選項和 `pack` pragma 會影響結構和等位成員的資料封裝。這個範例會示範 `/Zp` 和如何 `__declspec(align(#))` 共同作業：

```
struct S {
    char a;
    short b;
    double c;
    CACHE_ALIGN double d;
    char e;
    double f;
};
```

下表列出每個成員在不同 `/Zp` (或) 值下的位移 `#pragma pack`，顯示這兩個互動的方式。

ff	/ZP1	/ZP2	/ZP4	/ZP8
a	0	0	0	0
b	1	2	2	2

<code>ff</code>	<code>/ZP1</code>	<code>/ZP2</code>	<code>/ZP4</code>	<code>/ZP8</code>
c	3	4	4	8
d	32	32	32	32
e	40	40	40	40
f	41	42	44	48
<code>sizeof(S)</code>	64	64	64	64

如需詳細資訊，請參閱 [/zp](#) (結構成員對齊)。

因此，物件的位移是根據前一個物件的位移與目前封裝設定，但如果物件具有 `__declspec(align(#))` 屬性，情況就不是這樣，此時對齊是根據前一個物件的位移與物件的 `__declspec(align(#))` 值。

結束 Microsoft 專有

## 另請參閱

[\\_\\_declspec](#)

[ARM ABI 慣例總覽](#)

[x64 軟體慣例](#)

# allocate

2020/11/2 • [Edit Online](#)

## Microsoft 特定的

宣告 `allocate` 規範會為將配置資料項目的資料區段命名。

## 語法

```
* __declspec(allocate("***segname*)) ***declarator宣告子
```

## 備註

名稱`segname`必須使用下列其中一個 pragma 來宣告：

- `code_seg`
- `const_seg`
- `data_seg`
- `init_seg`
- `截面`

## 範例

```
// allocate.cpp
#pragma section("mycode", read)
__declspec(allocate("mycode")) int i = 0;

int main() {
```

結束 Microsoft 專有

## 另請參閱

[\\_\\_declspec](#)

關鍵字

# allocator

2020/11/2 • [Edit Online](#)

## Microsoft 特定的

宣告 `allocator` 規範可以套用至自訂記憶體配置函式，以透過 Windows 事件追蹤(ETW)顯示配置。

## 語法

```
__declspec(allocator)
```

## 備註

Visual Studio 中的原生記憶體分析工具的運作方式，是收集在執行時間期間發出的配置 ETW 事件資料。在來源層級已註釋 CRT 和 Windows SDK 中的配置器，以便擷取其配置資料。如果您要撰寫自己的配置器，則傳回新配置堆積記憶體之指標的任何函式都可以使用裝飾 `__declspec(allocator)`，如下列 mymalloc 所示範例所示：

```
__declspec(allocator) void* myMalloc(size_t size)
```

如需詳細資訊，請參閱[測量 Visual Studio 中的記憶體使用量和自訂的原生 ETW 堆積事件](#)。

結束 Microsoft 專有

# appdomain

2020/3/25 • [Edit Online](#)

指定 Managed 應用程式的每個應用程式定義域都應該有自己的特定全域變數或靜態成員變數複本。如需詳細資訊，請參閱[應用程式域和視覺效果C++](#)。

每個應用程式定義域都有自己的 per-appdomain 變數複本。組件載入應用程式定義域時，appdomain 變數的建構函式就會執行，而應用程式定義域卸載時，解構函式就會執行。

如果您希望 Common Language Runtime 中某個處理序內的所有應用程式定義域共用全域變數，請使用 `_declspec(process)` 修飾詞。`_declspec(process)` 預設會在 /clr 底下生效。/Clr: pure 和 /clr: safe 編譯器選項在 Visual Studio 2015 中已被取代，在 Visual Studio 2017 中不支援。

只有在使用其中一個 /clr 編譯器選項時，`_declspec(appdomain)` 才有效。只有全域變數、靜態成員變數或靜態區域變數可以擁有 `_declspec(appdomain)` 標記。將 `_declspec(appdomain)` 套用至 Managed 類型的靜態成員會發生錯誤，因為這些成員會固定出現這類行為。

使用 `_declspec(appdomain)` 類似于使用 [執行緒區域儲存區\(TLS\)](#)。執行緒擁有自己的儲存區，應用程式定義域也一樣。使用 `_declspec(appdomain)` 可確保全域變數在針對這個應用程式建立的每個應用程式定義域中擁有自己的儲存區。

混合使用每個進程和每個 appdomain 變數的限制如下：如需詳細資訊，請參閱[進程](#)。

例如，在程式啟動時，會先初始化所有處理序專屬變數，然後才會初始化所有 per-appdomain 變數。因此，當處理序專屬變數初始化時，它就無法依據任何應用程式定義域專屬變數的值。不建議您混合使用 (指派) appdomain 專屬和處理序專屬變數。

如需如何在特定應用程式域中呼叫函式的詳細資訊，請參閱[Call\\_in\\_appdomain 函數](#)。

## 範例

```
// 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) CrossAppDomain appdomain_global = stateless;

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);
}

```

```
__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
```

## 另請參閱

[\\_\\_declspec](#)

[關鍵字](#)

# code\_seg (\_declspec)

2020/3/25 • [Edit Online](#)

## Microsoft 專屬

Code\_seg 告訴屬性會為 .obj 檔案中的可執行文字區段命名，其中會儲存函式或類別成員函式的物件程式碼。

## 語法

```
__declspec(code_seg("segname")) declarator
```

## 備註

`__declspec(code_seg(...))` 屬性可用於將程式碼分為單獨的具名區段，而且這些區段可個別分頁或鎖定在記憶體中。您可以使用這個屬性控制具現化的範本及編譯器產生之程式碼的位置。

「區段」(segment)是 .obj 檔案中的資料命名區塊，會當做一個單位載入至記憶體中。「文字區段」(text segment)是包含可執行程式碼的區段。「詞彙」一詞通常會與「區段」交換使用。

定義 `declarator` 時所產生的目的碼會在 `segname` (窄字串常值) 所指定的文字區段中。不需要在 `區段` pragma 中指定名稱 `segname`，就可以在宣告中使用它。根據預設，若未指定任何 `code_seg`，目的碼的位置會在區段具名 .text 中。Code\_seg 屬性會覆寫任何現有的 #pragma code\_seg 指示詞。套用至成員函式的 code\_seg 屬性會覆寫套用至封入類別的任何 code\_seg 屬性。

如果實體具有 code\_seg 屬性，則相同實體的所有宣告和定義都必須具有相同的 code\_seg 屬性。如果基類具有 code\_seg 屬性，衍生的類別必須具有相同的屬性。

當 code\_seg 屬性套用至命名空間範圍函數或成員函式時，該函式的物件程式碼會放在指定的文字區段中。當這個屬性套用至類別時，該類別及巢狀類別的所有成員函式 (包括編譯器產生的特殊成員函式) 都會在指定的區段中。本機定義的類別 (例如，在成員函式主體中定義的類別) 不會繼承封閉式範圍的 code\_seg 屬性。

當 code\_seg 屬性套用至樣板類別或範本函式時，會將範本的所有隱含特製化放在指定的區段中。明確或部分特製化並不會繼承主要範本中的 code\_seg 屬性。您可以在特製化上指定相同或不同的 code\_seg 屬性。Code\_seg 屬性無法套用至明確的範本具現化。

根據預設，編譯器產生的程式碼 (如特殊成員函式) 會在 .text 區段中。#pragma code\_seg 指示詞不會覆寫這個預設值。使用 [類別]、[類別樣板] 或 [函式] 範本上的 [code\_seg] 屬性，即可控制編譯器產生之程式碼的放置位置。

Lambda 會從其封閉範圍繼承 code\_seg 屬性。若要指定 lambda 的區段，請將 code\_seg 屬性套用在參數宣告子句之後，以及任何可變動或例外狀況規格、任何尾端傳回類型規格和 lambda 主體之前。如需詳細資訊，請參閱 [Lambda 運算式語法](#)。這個範例會在名為 PagedMem 的區段中定義 Lambda：

```
auto Sqr = [](int t) __declspec(code_seg("PagedMem")) -> int { return t*t; };
```

當您將特定成員函式 (尤其是虛擬成員函式) 放在不同的區段時，請特別小心。若基底類別方法位於非分頁區段，而您在位於分頁區段之衍生類別中定義虛擬函式，其他基底類別方法或使用者程式碼可能會假設叫用該虛擬方法不會觸發分頁錯誤。

## 範例

這個範例示範當使用隱含和明確樣板特製化時，code\_seg 屬性如何控制區段位置：

```

// 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);
}

```

END Microsoft 特定的

## 另請參閱

[\\_\\_declspec](#)

[關鍵字](#)

# deprecated (C++)

2020/11/2 • [Edit Online](#)

本主題是關於 Microsoft 特定的已淘汰 `declspec` 壓告。如需 C++ 14 屬性的相關資訊 [\[\[deprecated\]\]](#)，以及使用該屬性與 Microsoft 特定 `declspec` 或 `pragma` 之時機的指引，請參閱 [C++ 標準屬性](#)。

使用下面所述的例外狀況時，宣告 `deprecated` 提供的功能與被 [取代](#) 的 `pragma` 相同：

- 宣告 `deprecated` 可讓您將特定形式的函式多載指定為已被取代，而 `pragma` 表單會套用至函式名稱的所有多載形式。
- 宣告 `deprecated` 可讓您指定將在編譯時期顯示的訊息。訊息的文字可以來自巨集。
- 宏只能使用 `pragma` 標記為已淘汰 `deprecated`。

如果編譯器遇到使用已被取代的識別碼或標準 [\[\[deprecated\]\]](#) 屬性，則會擲回 [C4996](#) 警告。

## 範例

下列範例將示範使用 `deprecated` 函式時，如何將函式標示為取代以及如何指定要在編譯時期顯示的訊息。

```
// 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
}
```

下列範例將示範使用 `deprecated` 類別時，如何將類別標示為取代以及如何指定要在編譯時期顯示的訊息。

```
// 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
}
```

## 另請參閱

[\\_\\_declspec](#)

關鍵字

# dllexport、dllimport

2020/11/2 • [Edit Online](#)

## Microsoft 特定的

`dllexport` 和 `dllimport` 儲存類別屬性是 Microsoft 特有的 C 和 c++ 語言擴充功能。您可以使用它們在 DLL 中匯出和匯入函式、資料和物件。

## 語法

```
_declspec( dllimport ) declarator  
_declspec( dllexport ) declarator
```

## 備註

這些屬性明確定義其用戶端的 DLL 介面 (可以是可執行檔或另一個 DLL)。將函式宣告為不 `dllexport` 需要模組定義 (.def) 檔案，至少與匯出函數的規格有關。`dllexport` 屬性會取代 `_export` 關鍵字。

如果類別已標記為 `declspec(dllexport)`，在類別階層中類別樣板的特製化會隱含標記為 `declspec(dllexport)`。這表示類別樣板是明確具現化，必須定義類別的成員。

`dllexport` 函式會以裝飾名稱來公開函式。對於 C++ 函式，這包括名稱修飾 (Name Mangling)。對於 C 函式或宣告為 `extern "C"` 的函式，其中包括根據呼叫慣例的平台專屬裝飾。如需 C/c++ 程式碼中名稱裝飾的相關資訊，請參閱 [裝飾名稱](#)。`extern "C"` 使用呼叫慣例，不會將名稱裝飾套用至匯出的 C 函式或 c++ 函式 `_cdecl`。

若要匯出未裝飾的名稱，則連結方式是使用在 EXPORTS 區段中定義未裝飾名稱的模組定義 (.def) 檔。如需詳細資訊，請參閱 [匯出](#)。另一個匯出未裝飾名稱的方法，是 `#pragma comment(linker, "/export:alias=decorated_name")` 在原始程式碼中使用指示詞。

當您宣告 `dllexport` 或時 `dllimport`，您必須使用 [擴充屬性語法](#) 和 `_declspec` 關鍵字。

## 範例

```
// Example of the dllimport and dllexport class attributes  
_declspec( dllimport ) int i;  
_declspec( dllexport ) void func();
```

或者，您可以使用巨集定義讓程式碼更容易讀取：

```
#define DllImport _declspec( dllimport )  
#define DllExport _declspec( dllexport )  
  
DllExport void func();  
DllImport int i = 10;  
DllImport int j;  
DllExport int n;
```

如需詳細資訊，請參閱

- [定義和宣告](#)

- 使用 `__declspec(dllexport)` 和 `__declspec(dllimport)` 定義內嵌 C++ 函式
- 一般規則和限制
- 在 C++ 類別中使用 `__declspec(dllimport)` 和 `__declspec(dllexport)`

結束 Microsoft 專有

## 另請參閱

[\\_\\_declspec](#)

關鍵字

# 定義和宣告 (C++)

2020/11/2 • [Edit Online](#)

## Microsoft 特定的

DLL 介面會參考系統中某些程式所能匯出的所有專案(函式和資料);也就是宣告為或的所有專案 `dllimport` `dllexport`。包含在 DLL 介面中的所有宣告都必須指定 `dllimport` 或 `dllexport` 屬性。不過, 定義必須僅指定 `dllexport` 屬性。例如, 下列函式定義會產生編譯器錯誤:

```
_declspec( dllimport ) int func() { // Error; dllimport
                                    // prohibited on definition.
    return 1;
}
```

此程式碼也會產生錯誤:

```
_declspec( dllimport ) int i = 10; // Error; this is a definition.
```

不過, 這個語法是正確的:

```
_declspec( dllexport ) int i = 10; // Okay--export definition
```

使用 `dllexport` 表示定義, 而則表示宣告 `dllimport`。您必須搭配使用 `extern` 關鍵字 `dllexport` 來強制執行宣告, 否則會隱含定義。因此, 下列範例是正確的:

```
#define DllImport _declspec( dllimport )
#define DllExport _declspec( dllexport )

extern DllExport int k; // These are both correct and imply a
DllImport int j;       // declaration.
```

下列範例釐清前述範例:

```
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.
}
```

**另請參閱**

[dllexport](#)、[dllimport](#)

# 使用 `__declspec(dllexport)` 和 `__declspec(dllimport)` 定義內嵌 C++ 函式

2020/11/2 • [Edit Online](#)

## Microsoft 特定的

您可以使用屬性，將函式定義為內嵌函數 `__declspec(dllexport)`。在這種情況下，無論程式中是否有任何模組參考函式，函式都一定會具現化並匯出。函式會假定為由其他程式匯入。

您也可以將以屬性宣告的函式定義為內嵌 `__declspec(dllimport)`。在這種情況下，函式可以展開(須遵循 /Ob 規格)，但絕不會具現化。特別是，如果接受匯入的內嵌函式位址，則會傳回位於 DLL 中函式的位址。這種行為與接受匯入的非內嵌函式位址相同。

這些規則適用於其定義出現在類別定義內的內嵌函式。此外，內嵌函式中的靜態區域資料和字串會在 DLL 和用戶端之間維護相同的識別，就像在單一程式(也就是沒有 DLL 介面的可執行檔)中一樣。

提供匯入的內嵌函式時務必特別小心。例如，如果您更新 DLL，請不要假設用戶端將會使用變更後的 DLL 版本。若要確保載入正確的 DLL 版本，請一併重建 DLL 的用戶端。

## 結束 Microsoft 專有

## 另請參閱

[\\_\\_declspec\(dllexport\)](#)、[\\_\\_declspec\(dllimport\)](#)

# 一般規則和限制

2020/11/2 • [Edit Online](#)

## Microsoft 特定的

- 如果您宣告不含或屬性的函式或物件 `dllimport` `dllexport`，則函式或物件不會被視為 DLL 介面的一部分。因此，函式或物件的定義必須存在該模組中，或是相同程式的另一個模組中。若要讓函式或物件成為 DLL 介面的一部分，您必須在另一個模組中將函式或物件的定義宣告為 `dllexport`。否則會產生連結器錯誤。

如果您宣告具有屬性的函式或物件 `dllexport`，其定義必須出現在同一個程式的某些模組中。否則會產生連結器錯誤。

- 如果程式中的單一模組同時包含相同函式 `dllimport` 或物件的和宣告 `dllexport`，則 `dllexport` 屬性會優先于 `dllimport` 屬性。不過，這樣會產生編譯器警告。例如：

```
__declspec( dllimport ) int i;
__declspec( dllexport ) int i; // Warning; inconsistent;
                             // dllexport takes precedence.
```

- 在 C++ 中，您可以初始化全域宣告或靜態本機資料指標，或使用以屬性宣告之資料物件的位址 `dllimport`，這會在 C 中產生錯誤。此外，您可以使用以屬性宣告的函式位址來初始化靜態區域函式指標 `dllimport`。在 C 中，此類指派會將指標設定為 DLL 匯入 Thunk (將控制權傳送至函式的程式碼 Stub) 的位址，而不是設定為函式的位址。在 C++ 中，它會將指標設定為函式的位址。例如：

```
__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++
}
```

不過，因為在 `dllexport` 物件的宣告中包含屬性的程式必須在程式中的某處提供該物件的定義，您可以使用函式的位址來初始化全域或區域靜態函式指標 `dllexport`。同樣地，您可以使用資料物件的位址來初始化全域或本機靜態資料指標 `dllexport`。例如，下列程式碼不會在 C 或 C++ 中產生錯誤：

```
__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
}
```

- 如果您將套用 `__declspec(dllexport)` 至具有基類但未標記為的一般類別 `__declspec(dllexport)`，則編譯器會產生 C4275。

如果基底類別是類別樣板的特製化，則編譯器會產生相同的警告。若要解決這個情況，請將基類標示為 `__declspec(dllexport)`。類別樣板的特製化問題在於放置的位置，`__declspec(dllexport)` 不允許您標記類別樣板。相反地，請明確將類別樣板具現化，並將這個明確具現化標示為 `__declspec(dllexport)`。例如：

```
template class __declspec(dllexport) B<int>;
class __declspec(dllexport) D : public B<int> {
// ...
```

如果樣板引數為衍生類別，則這個解決方法會失敗。例如：

```
class __declspec(dllexport) D : public B<D> {
// ...
```

因為這是使用樣板的通用模式，所以編譯器會在套用 `__declspec(dllexport)` 至具有一或多個基類的類別，以及當一個或多個基類是類別樣板的特製化時，將的語義變更為。在此情況下，編譯器會隱含地套用 `__declspec(dllexport)` 至類別樣板的特製化。您可以執行下列動作，而不會收到警告：

```
class __declspec(dllexport) D : public B<D> {
// ...
```

結束 Microsoft 專有

## 另請參閱

[\\_\\_declspec\(dllexport\)](#), [\\_\\_declspec\(dllimport\)](#)

# 在 C++ 類別中使用 `dllimport` 和 `dllexport`

2020/11/2 • [Edit Online](#)

## Microsoft 特定的

您可以使用或屬性宣告 C++ 類別 `dllimport` `dllexport`。這些形式表示會將整個類別匯入或匯出。以這種方式匯出的類別稱為可匯出類別。

下列範例將定義可匯出類別。它的所有成員函式和靜態資料都會匯出：

```
#define DllExport __declspec( dllexport )

class DllExport C {
    int i;
    virtual int func( void ) { return 1; }
};
```

請注意，`dllimport` `dllexport` 禁止在可匯出類別的成員上明確使用和屬性。

## `dllexport` 類別

當您宣告類別時 `dllexport`，它的所有成員函式和靜態資料成員都會匯出。您必須提供相同程式中所有這類成員的定義。否則會產生連結器錯誤。這項規則的例外狀況適用於純虛擬函式，您不需要為其提供明確定義。不過，由於抽象類別的解構函式一定是由基底類別的解構函式所呼叫，因此純虛擬解構函式一定要提供定義。請注意，這些規則同樣適用於不可匯出的類別。

如果您匯出類別類型的資料或傳回類別的函式，則務必匯出類別。

## `dllimport` 類別

當您宣告類別時 `dllimport`，它的所有成員函式和靜態資料成員都會匯入。不同于 `dllimport` 和 `dllexport` 在非類別類型上的行為，靜態資料成員無法在定義類別的相同程式中指定定義 `dllimport`。

## 繼承和可匯出類別

可匯出類別的所有基底類別都必須為可匯出。否則會產生編譯器警告。此外，同樣為類別的所有可存取成員也都必須為可匯出。此規則允許 `dllexport` 類別繼承自 `dllimport` 類別，而類別則是繼承自 `dllimport` `dllexport` 類別（但不建議後者）。通常，DLL 用戶端可以存取的所有項目（根據 C++ 存取規則），都必須是可匯出介面的一部分。這包括內嵌函式中參考的 private 資料成員。

## 選擇性成員匯入/匯出

因為成員函式和類別內的靜態資料會隱含具有外部連結，所以您可以使用 `dllimport` 或屬性來宣告它們 `dllexport`，除非匯出整個類別。如果匯入或匯出整個類別，則禁止將成員函式和資料明確宣告為 `dllimport` 或 `dllexport`。如果您將類別定義中的靜態資料成員宣告為 `dllexport`，則定義必須發生在相同程式中的某處（如同非類別外部連結）。

同樣地，您可以使用或屬性宣告成員函式 `dllimport` `dllexport`。在此情況下，您必須在 `dllexport` 相同程式內的某個位置提供定義。

關於選擇性成員匯入和匯出，有幾項重點值得注意：

- 選擇性成員匯入/匯出最適合用於提供較嚴格的匯出類別介面版本，也就是您可以為這個介面設計 DLL，公開比語言所允許更少的公用或私用功能。另外也很適合用於微調可匯出介面：當您根據定義得知用戶端無法存取某些私用資料時，您不需要匯出整個類別。
- 如果您匯出類別中的某一個虛擬函式，就必須匯出所有虛擬函式，或是至少要提供用戶端可以直使用版本。
- 如果您在所擁有的類別中使用具有虛擬函式的選擇性成員匯入/匯出，這些函式必須位於可匯出介面中，或是定義為內嵌（用戶端能夠看見）。
- 如果您將成員定義為 `__declspec(dllexport)`，但未將它包含在類別定義中，則會產生編譯器錯誤。您必須在類別標頭中定義成員。
- 雖然允許將類別成員定義為 `__declspec(dllimport)` 或 `__declspec(dllexport)`，但您不能覆寫在類別定義中指定的介面。
- 如果您在宣告它的類別定義主體以外的地方定義成員函式，則會在函式定義為 `__declspec(dllexport)` 或 `__declspec(dllimport)`（如果此定義與類別宣告中所指定的不同）時產生警告。

結束 Microsoft 專有

## 另請參閱

[\\_\\_declspec\(dllexport\)](#)、[\\_\\_declspec\(dllimport\)](#)

# jitintrinsic

2020/11/2 • [Edit Online](#)

將函式標記為 64 位元 Common Language Runtime 的必要項。這種方式會在 Microsoft 所提供程式庫中的某些函式上使用。

## 語法

```
_declspec(jitintrinsic)
```

## 備註

`jitintrinsic` 將 MODOPT () 加入至函式簽章 `IsJitIntrinsic`。

不鼓勵使用者使用此 `_declspec` 修飾詞，因為可能會發生非預期的結果。

## 另請參閱

[\\_declspec](#)

[關鍵字](#)

# naked (C++)

2020/11/2 • [Edit Online](#)

## Microsoft 特定的

若為以屬性宣告的函式 `naked`，編譯器會產生程式碼，而不含初構和終解程式碼。利用此功能就可以使用內嵌組合語言程式碼撰寫您自己的初構/終解程式碼序列。naked 函式在撰寫虛擬裝置驅動程式方面特別實用。請注意，此 `naked` 屬性僅適用於 x86 和 ARM，而且在 x64 上無法使用。

## 語法

```
__declspec(naked) declarator
```

## 備註

因為屬性僅與函式 `naked` 的定義相關，而且不是類型修飾詞，所以 naked 函數必須使用擴充屬性語法和 `_declspec` 關鍵字。

編譯器無法針對以 naked 屬性標記的函式產生內嵌函式，即使函式也以 `_forceinline` 關鍵字標記也一樣。

如果 `naked` 屬性套用至非成員方法定義以外的任何專案，則編譯器會發出錯誤。

## 範例

此程式碼會定義具有屬性的函式 `naked`：

```
__declspec( naked ) int func( formal_parameters ) {}
```

或者，或者：

```
#define Naked __declspec( naked )
Naked int func( formal_parameters ) {}
```

`naked` 屬性只會影響編譯器的初構和終解序列之程式碼產生的本質。它不會影響針對呼叫這類函式所產生的程式碼。因此，`naked` 屬性不會被視為函式類型的一部分，而且函式指標不能有 `naked` 屬性。此外，`naked` 屬性無法套用至資料定義。例如，此程式碼範例會產生錯誤：

```
__declspec( naked ) int i;
// Error--naked attribute not permitted on data declarations.
```

屬性只與函式的 `naked` 定義相關，而且不能在函數的原型中指定。例如，此宣告會產生編譯器錯誤：

```
__declspec( naked ) int func(); // Error--naked attribute not permitted on function declarations
```

結束 Microsoft 專有

另請參閱

`_declspec`

關鍵字

[Naked 函式呼叫](#)

# noalias

2020/11/2 • [Edit Online](#)

## Microsoft 特定

`noalias` 表示函式呼叫不會修改或參考可見的全域狀態，而且只會修改指標參數(第一層間接取值)直接指向的記憶體。

如果函式標注為 `noalias`，則優化工具可以假設函式內只會參考或修改參數本身，而只有第一層間接取值指標參數。

`noalias` 注釋僅適用於批註函式的主體內。將函式標記為 `__declspec(noalias)` 不會影響函式所傳回之指標的別名。

如需可能會影響別名的另一個批註，請參閱 [`\_\_declspec\(restrict\)`](#)。

## 範例

下列範例示範的用法 `__declspec(noalias)`。

在 `multiply` 批註存取記憶體的函式時 `__declspec(noalias)`，它會告知編譯器，此函式不會修改全域狀態，除非透過其參數清單中的指標。

```

// 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);
}

```

`__declspec`

關鍵字

`__declspec(restrict)`

# noinline

2020/11/2 • [Edit Online](#)

## Microsoft 特定的

`__declspec(noinline)` 告知編譯器永遠不會內嵌特定成員函式(類別中的函式)。

如果函式不大，而且對程式碼的效能不重要，建議不要內嵌函式。也就是，如果函式不大且可能不常被呼叫，例如處理錯誤條件的函式。

請記住，如果函式已標記 `noinline`，則呼叫函式將會較小，因此本身是編譯器內嵌的候選項。

```
class X {  
    __declspec(noinline) int mbrfunc() {  
        return 0;  
    } // will not inline  
};
```

## 結束 Microsoft 專有

## 另請參閱

[\\_\\_declspec](#)

[關鍵字](#)

[內嵌、\\_\\_inline、\\_\\_forceinline](#)

# noreturn

2020/11/2 • [Edit Online](#)

## Microsoft 特定的

這個 `__declspec` 屬性會告知編譯器，函數不會傳回。因此，編譯器知道呼叫函式之後的程式碼 `__declspec(noreturn)` 無法連線。

如果編譯器發現某個函式包含的控制路徑不會傳回值，則會產生警告 (C4715) 或錯誤訊息 (C2202)。如果因為函式永遠不會傳回而無法到達控制路徑，您可以使用 `__declspec(noreturn)` 來避免這個警告或錯誤。

### NOTE

新增 `__declspec(noreturn)` 至預期會傳回的函式可能會導致未定義的行為。

## 範例

在下列範例中，`else` 子句不包含 `return` 語句。宣告 `fatal` 為 `__declspec(noreturn)` 可避免出現錯誤或警告訊息。

```
// noreturn2.cpp
__declspec(noreturn) extern void fatal () {}

int main() {
    if(1)
        return 1;
    else if(0)
        return 0;
    else
        fatal();
}
```

結束 Microsoft 專有

## 另請參閱

[\\_\\_declspec](#)

[關鍵字](#)

# nothrow (C++)

2020/11/2 • [Edit Online](#)

Microsoft 特定的

`__declspec` 擴充屬性，可用於函數的宣告。

## 語法

傳回類型`__declspec (nothrow) [呼叫慣例] 函式名稱([引數清單])`

## 備註

我們建議所有新的程式碼都使用`noexcept`運算子，而不是`__declspec(nothrow)`。

這個屬性會告知編譯器其所呼叫的已宣告函式絕對不會擲回例外狀況。不過，它不會強制執行指示詞。換句話說，它絕不會導致叫用`std::terminate`，而不是`noexcept`，或在`std: c + + 17`模式中(Visual Studio 2017 15.5 版和更新版本)`throw()`。

使用同步例外狀況處理模型時(現在為預設值)，編譯器可以排除在這類函式中追蹤某些無法還原物件存留期的機制，並大幅降低程式碼大小。假設有下列的預處理器指示詞，下列三個函式宣告在`/std: c + + 14`模式中是相等的：

```
#define WINAPI __declspec(nothrow) __stdcall  
  
void WINAPI f1();  
void __declspec(nothrow) __stdcall f2();  
void __stdcall f3() throw();
```

在`/std: c + + 17`模式中，`throw()`不等於其他使用的，因為如果函式擲回例外狀況，則會導致叫用`__declspec(nothrow) std::terminate`。

宣告會`void __stdcall f3() throw();`使用`c + +`標準所定義的語法。在`c + + 17`中，`throw()`關鍵字已被取代。

結束 Microsoft 專有

## 另請參閱

[\\_\\_declspec](#)

[noexcept](#)

[關鍵字](#)

# novtable

2020/11/2 • [Edit Online](#)

## Microsoft 特定的

這是 `__declspec` 擴充屬性。

這種形式的 `__declspec` 可以套用至任何類別宣告，但只能套用至純介面類別別，也就是永遠不會自行具現化的類別。會 `__declspec` 阻止編譯器產生程式碼，以初始化函式中的 vptr 和類別的析構函數。在大部分情況下，這樣只能移除與類別相關的 vtable 參考，因此連結器會將它移除。使用這種形式的 `__declspec` 可能會大幅減少程式碼大小。

如果您嘗試將標記為的類別具現化，`novtable` 然後再存取類別成員，您將會收到存取違規(AV)。

## 範例

```
// 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();
}
```

In Y

## 結束 Microsoft 專有

## 另請參閱

[\\_\\_declspec](#)

[關鍵字](#)

process

2020/11/2 • [Edit Online](#)

指定您的 Managed 應用程式處理序應該使用特定全域變數的單一複本、靜態成員變數，或在處理序中的所有應用程式定義域中共用的靜態區域變數。這主要是在使用進行編譯時使用 `/clr:pure`，它在 Visual Studio 2015 中已被取代，在 Visual Studio 2017 中不支援。使用進行編譯時 `/clr`，全域和靜態變數預設為每個進程，且不需要使用 `__declspec(process)`。

只有全域變數、靜態成員變數或原生類型的靜態區域變數可以用標記 `__declspec(process)`。

`process` 只有在使用進行編譯時才有效 `/clr`。

如果您想要讓每個應用程式域都有自己的全域變數複本，請使用[appdomain](#)。

如需詳細資訊，請參閱[應用程式域和 Visual C++](#)。

## 另請參閱

`__declspec`

關鍵字

# property (C++)

2020/3/25 • [Edit Online](#)

## Microsoft 專屬

這個屬性可以套用至類別或結構定義中的非靜態「虛擬資料成員」。編譯器會將這些「虛擬資料成員」的參考變更為函式呼叫，將它們視為資料成員。

## 語法

```
__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
```

## 備註

當編譯器在成員選取運算子("." 或 "->")右邊看到以這個屬性宣告的資料成員時，它會根據這類運算式是左值或右值，將作業轉換成 `get` 或 `put` 函數。在更複雜的內容中(例如 "`+=`")，會同時執行 `get` 和 `put` 來進行重寫。

在類別或結構定義中也可以使用這個屬性宣告空陣列。例如：

```
__declspec(property(get=GetX, put=PutX)) int x[];
```

上述陳述式表示可以同時使用 `x[]` 和一個或多個陣列索引。在這種情況下，`i=p->x[a][b]` 會轉換為 `i=p->GetX(a, b)`，而 `p->x[a][b] = i` 則會轉換為 `p->PutX(a, b, i);`

END Microsoft 特定的

## 範例

```
// 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;  
}
```

## 另請參閱

`_declspec`

關鍵字

# restrict

2020/11/2 • [Edit Online](#)

## Microsoft 特定的

套用至傳回指標類型的函式聲明或定義時，`restrict` 會告知編譯器函式傳回未加上別名的物件，也就是由任何其他指標所參考。這可讓編譯器執行其他優化。

## 語法

```
* __declspec(restrict) ***pointer_return_type 函數();
```

## 備註

編譯器會傳播 `__declspec(restrict)`。例如，CRT 函式 `malloc` 具有 `__declspec(restrict)` 裝飾，因此，編譯器會假設初始化為記憶體位置的指標，`malloc` 也不是先前現有指標的別名。

編譯器不會檢查傳回的指標實際上是否為別名。開發人員必須負責確保程式不會以 [限制 `__declspec`] 修飾詞標示指標的別名。

如需變數的類似語義，請參閱 [\\_restrict](#)。

如需另一個適用于函式中別名的注釋，請參閱 [\\_declspec \(noalias\)](#)。

如需 C++ AMP 的關鍵字的詳細資訊 `restrict`，請參閱 [restrict \(C++ AMP\)](#)。

## 範例

下列範例示範的用法 `__declspec(restrict)`。

當套用至傳回指標的函式時 `__declspec(restrict)`，這會告訴編譯器傳回值所指向的記憶體沒有別名。在此範例中，指標 `mempool` 和 `memptr` 為全域，因此編譯器無法確定它們所參考的記憶體沒有別名。不過，它們會用在 `ma` 和其呼叫者 `init` 中，以傳回程序未以其他方式參考的記憶體，因此 `__declspec (限制)` 用來協助優化工具。這類似于 CRT 標頭如何裝飾配置函式，例如 `malloc` 使用 `__declspec(restrict)` 來表示它們一律會傳回無法以現有指標做為別名的記憶體。

```

// declspec_restrict.c
// Compile with: cl /W4 declspec_restrict.c
#include <stdio.h>
#include <stdlib.h>

#define M 800
#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);
}

```

## 另請參閱

關鍵字

[\\_\\_declspec](#)

[\\_\\_declspec \(noalias\)](#)

# safebuffers

2020/11/2 • [Edit Online](#)

## Microsoft 特定的

告知編譯器不要插入函式的緩衝區滿溢安全性檢查。

## 語法

```
_declspec( safebuffers )
```

## 備註

/Gs 編譯器選項會藉由在堆疊上插入安全性檢查，讓編譯器測試緩衝區溢位。[/Gs \(緩衝區安全性檢查\)](#) 中會說明適用於安全性檢查的資料結構類型。如需有關緩衝區溢位偵測的詳細資訊，請參閱[MSVC 中的安全性功能](#)。

某位專業人員手動檢閱程式碼或進行外部分析後，可能會判斷函式不會發生緩衝區滿溢。在這種情況下，您可以藉由將 `_declspec(safebuffers)` 關鍵字套用至函式宣告來隱藏函式的安全性檢查。

### Caution

緩衝區安全性檢查提供了重要的安全性保護，且幾乎不會對效能造成影響。因此，除了在少數函式的效能為重要考量，以及函式已知為安全的情況以外，建議您不要抑制這些檢查。

## 內嵌函式

主要函式可以使用[內嵌](#)關鍵字來插入次要函數的複本。如果關鍵字套用至函式 `_declspec(safebuffers)`，則會抑制該函數的緩衝區溢位偵測。不過，內嵌會 `_declspec(safebuffers)` 以下列方式影響關鍵字。

假設已針對這兩個函式指定 /gs 編譯器選項，但主要函式指定 `_declspec(safebuffers)` 關鍵字。第二個函式中的資料結構會使其進行安全性檢查，因此函式不會抑制這些檢查。在此案例中：

- 在次要函式上指定[\\_forceinline](#)關鍵字，強制編譯器內嵌該函式，而不論編譯器優化。
- 因為次要函式符合安全性檢查的資格，即使它指定關鍵字，也會將安全性檢查套用至主要功能 `_declspec(safebuffers)`。

## 範例

下列程式碼顯示如何使用 `_declspec(safebuffers)` 關鍵字。

```
// 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;
}
```

結束 Microsoft 專有

## 另請參閱

[\\_\\_declspec](#)

[關鍵字](#)

[內嵌、\\_\\_inline、\\_\\_forceinline](#)

[strict\\_gs\\_check](#)

# selectany

2020/11/2 • [Edit Online](#)

## Microsoft 特定的

告訴編譯器宣告的全域資料項目 (變數或物件) 是挑選任一個 (Pick-any) COMDAT (封裝函式)。

## 語法

```
* __declspec( selectany ) ***declarator宣告子
```

## 備註

在連結時，如果看見多個定義的 COMDAT，則連結器會挑選一個定義並捨棄其餘定義。如果選取了 (優化) 的連結器選項 [/OPT:REF](#)，則會進行 COMDAT 刪除，以移除連結器輸出中所有未參考的資料項目。

宣告中的建構函式以及由全域函式或靜態方法進行的指派不會建立參考，也不會阻止 /OPT:REF 刪除作業。來自這類程式碼的副作用不應取決於不存在其他資料參考時。

對於動態初始化的全域物件，也 `selectany` 會捨棄未參考物件的初始化程式碼。

全域資料項目通常只能在 EXE 或 DLL 專案中初始化一次。`selectany` 當相同標頭出現在多個原始程式檔時，可用於初始化標頭所定義的全域資料。`selectany` C 和 c + + 編譯器都有提供。

### NOTE

`selectany` 只能套用至外部可見的全域資料項目實際初始化。

## 範例：`selectany` 屬性

此程式碼說明如何使用 `selectany` 屬性：

```

//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);

```

## 範例：使用 `selectany` 屬性來確保資料 COMDAT 折迭

這段程式碼示範如何使用 `selectany` 屬性，以確保當您同時使用連結器選項時的資料 COMDAT 折迭 [/OPT:ICF](#)。請注意，資料必須標記 `selectany` 並放在 `const` (readonly) 區段中。您必須明確指定唯讀區段。

```

// 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;
}

```

結束 Microsoft 專有

## 另請參閱

[\\_\\_declspec](#)

關鍵字

# spectre

2019/12/2 • [Edit Online](#)

## Microsoft 專屬

告訴編譯器不要為函式插入 Spectre variant 1 的推測執行屏障指示。

## 語法

```
__declspec( spectre(nomitigation) )
```

## 備註

[/Qspectre](#) 編譯器選項會使編譯器插入推測性執行屏障指示。它們會插入，其中分析指出有 Spectre variant 1 安全性弱點存在。所發出的特定指示取決於處理器。雖然這些指示對程式碼大小或效能應該會有最小的影響，但有時候您的程式碼不會受到弱點的影響，而且需要最大效能。

專家分析可能會判斷函式不會受到 Spectre 變異 1 界限檢查略過瑕疵的保護。在這種情況下，您可以將套用

```
__declspec(spectre(nomitigation))
```

 至函式宣告，以隱藏函式中的風險降低程式碼產生。

### Caution

[/Qspectre](#) 的推測性執行屏障指示提供重要的安全性保護，而且對效能的影響也不明顯。因此，除了在少數函式的效能為重要考量，以及函式已知為安全的情況以外，建議您不要抑制這些檢查。

## 範例

下列程式碼顯示如何使用 `__declspec(spectre(nomitigation))`。

```
// compile with: /c /Qspectre
static __declspec(spectre(nomitigation))
int noSpectreIssues() {
    // No Spectre variant 1 vulnerability here
    // ...
    return 0;
}

int main() {
    noSpectreIssues();
    return 0;
}
```

結束 Microsoft 專屬

## 另請參閱

[\\_\\_declspec](#)

[關鍵字](#)

[/Qspectre](#)

# 執行緒

2020/11/2 • [Edit Online](#)

## Microsoft 特定的

`thread` 擴充儲存類別修飾詞是用來宣告執行緒區域變數。如需 C++ 11 和更新版本中的可移植對等用法，請使用可移植程式碼的`thread_local` 儲存類別規範。Windows 上的 `thread_local` 會使用來執行 `__declspec(thread)`。

## 語法

\* `__declspec(thread) ***declarator` 告子

## 備註

執行緒區域儲存區 (Thread Local Storage, TLS) 是一種機制，讓多執行緒處理序中的每個執行緒用來配置儲存區，以儲存執行緒特定資料。在標準多執行緒程式中，資料是在特定處理序的所有執行緒之間共用，而執行緒區域儲存區則是用於配置每個執行緒資料的機制。如需執行緒的完整討論，請參閱[多執行緒](#)。

執行緒區域變數的宣告必須使用擴充屬性語法和關鍵字搭配 `__declspec` `thread` 關鍵字。例如，下列程式碼宣告整數執行緒區域變數，並使用值將它初始化：

```
__declspec( thread ) int tls_i = 1;
```

在動態載入的程式庫中使用執行緒區域變數時，您必須留意可能導致執行緒區域變數無法正確初始化的因素：

- 如果變數是使用函式呼叫（包括函式）進行初始化，則只會針對導致二進位/DLL 載入進程的執行緒，以及在載入二進位/DLL 之後啟動的執行緒呼叫此函數。載入 DLL 時，任何已執行的其他執行緒都不會呼叫初始化函數。動態初始化會在 `DLL_THREAD_ATTACH` 的 `DllMain` 呼叫上進行，但是如果 DLL 線上程啟動時不在進程中，則 DLL 永遠不會取得該訊息。
- 使用常數值以靜態方式初始化的執行緒區域變數，通常會在所有線程上正確地初始化。不過，從 2017 年 12 月起，Microsoft C++ 編譯器中有已知的一致性問題，`constexpr` 而變數會接收動態而非靜態初始化。

注意：在未來的編譯器更新中，這兩個問題都應該是固定的。

此外，在宣告執行緒區域物件和變數時，您必須遵守下列方針：

- 您 `thread` 只能將屬性套用至類別和資料宣告和定義；`thread` 不能用在函式宣告或定義上。
- 您 `thread` 只能在具有靜態儲存期的資料項目上指定屬性。這包括全域資料物件（`static` 和 `extern`）、本機靜態物件，以及類別的靜態資料成員。您無法使用屬性來宣告自動資料物件 `thread`。
- `thread` 不論宣告和定義發生在相同的檔案還是不同的檔案中，您都必須將屬性用於執行緒區域物件的宣告和定義。
- 您不能使用 `thread` 屬性做為型別修飾詞。
- 因為允許使用屬性的物件宣告 `thread`，所以這兩個範例在語義上是相等的：

```
// 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.
```

- 標準 C 允許將物件或變數初始化為涉及本身參考的運算式，但僅適用於非靜態物件。雖然 C++ 通常允許物件的這類動態初始化具有涉及本身參考的運算式，但這種初始化不允許用於執行緒區域物件。例如：

```
// 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++
```

`sizeof` 包含所要初始化物件的運算式並不會構成其本身的參考，並可在 C 和 C++ 中使用。

結束 Microsoft 專有

## 另請參閱

[\\_\\_declspec](#)

[關鍵字](#)

[執行緒區域儲存區\(TLS\)](#)

# uuid (C++)

2020/11/2 • [Edit Online](#)

## Microsoft 特定的

編譯器會使用屬性，將 GUID 附加至宣告或定義的類別或結構(僅限完整的 COM 物件定義) `uuid`。

## 語法

```
__declspec( uuid("ComObjectGUID") ) declarator
```

## 備註

`uuid` 屬性會採用字串做為其引數。這個字串會以一般登錄格式來命名 GUID，其中包含或不含 {} 分隔符號。例如：

```
struct __declspec(uuid("00000000-0000-0000-c000-000000000046")) IUnknown;
struct __declspec(uuid("{00020400-0000-0000-c000-000000000046}") IDispatch;
```

這個屬性可以在重新宣告中套用。這可讓系統標頭提供介面的定義(例如 `IUnknown`)，並在其他標頭(例如)中重新宣告 `<comdef.h>` 以提供 GUID。

關鍵字 `_uuidof` 可以套用來抓取附加至使用者自訂類型的常數 GUID。

結束 Microsoft 專有

## 另請參閱

[\\_\\_declspec](#)

[關鍵字](#)

# `_restrict`

2020/12/10 • [Edit Online](#)

如同 `__declspec` (`restrict`) 修飾詞, `_restrict` 關鍵字 (兩個前置底線 '\_') 表示符號在目前範圍中沒有別名。`_restrict` 關鍵字與修飾詞的差異有 `__declspec (restrict)` 下列幾種:

- `_restrict` 關鍵字只適用於變數, 而且 `__declspec (restrict)` 只在函式宣告和定義上有效。
- `_restrict` 類似於 `restrict` C99 中啟動的 c, 但 `_restrict` 可用於 c++ 和 c 程式。
- `_restrict` 使用時, 編譯器不會傳播變數的無別名屬性。也就是說, 如果您將變數指派 `_restrict` 至非 `_restrict` 變數, 編譯器仍會允許非 `_restrict` 變數進行別名。這與 C99 C language 關鍵字的行為不同 `restrict`。

一般來說, 如果您想要影響整個函式的行為, 請使用 `__declspec (restrict)` 而不是關鍵字。

為了與舊版相容, `_restrict` `_restrict` 除非指定了編譯器選項 `/Za` (停用語言延伸), 否則是的同義字。

在 Visual Studio 2015 和更新版本中, `_restrict` 可以用於 c++ 參考。

## NOTE

在同時具有關鍵詞的變數上使用時 `volatile`, `volatile` 將會優先使用。

## 範例

```
// __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;
};
```

## 另請參閱

[關鍵字](#)

# `_sptr`、`_uptr`

2020/11/2 • [Edit Online](#)

## Microsoft 特定的

`_sptr` `_uptr` 在 32 位指標宣告上使用或修飾詞，指定編譯器如何將 32 位指標轉換為 64 位指標。例如，將 32 位元指標指派給 64 位元指標變數，或在 64 位元平台取值時，就會轉換 32 位元指標。

有關 64 位元平台支援的 Microsoft 文件有時會將 32 位元指標的最高有效位元稱為正負號位元。根據預設，編譯器會使用正負號擴充項目將 32 位元指標轉換為 64 位元指標。也就是說，64 位元指標的最低有效 32 位元設為 32 位元指標的值，而最高有效 32 位元則設為 32 位元指標的正負號位元的值。如果正負號位元是 0，此類轉換會產生正確的結果，如果正負號位元是 1，則不會產生正確結果。例如，32 位元位址 0x7FFFFFFF 會產生相等的 64 位元位址 0x00000007FFFFFF，但會將 32 位元位址 0x80000000 錯誤地變更為 0xFFFFFFFF80000000。

`_sptr`（或帶正負號的指標）修飾詞會指定指標轉換，將 64 位指標的最高有效位設為 32 位指標的符號位。`_uptr`（或不帶正負號的指標）修飾詞指定轉換將最重要的位設定為零。下列宣告顯示 `_sptr` `_uptr` 搭配兩個非限定指標使用的和修飾詞、兩個以 `_ptr32` 類型限定的指標，以及函式參數。

```
int * __sptr psp;
int * __uptr pup;
int * __ptr32 __sptr psp32;
int * __ptr32 __uptr pup32;
void MyFunction(char * __uptr __ptr32 myValue);
```

使用和修飾詞搭配 `_sptr` `_uptr` 指標宣告。請在 [指標類型限定詞](#) 的位置使用修飾詞，這表示修飾詞必須在星號後面。您不能將修飾詞與 [成員的指標](#) 搭配使用。修飾詞不影響非指標宣告。

為了與舊版相容，`_sptr` 和 `_uptr` 都是和的同義字，`_sptr` `_uptr` 除非指定了編譯器選項 [/za 停用（語言擴充功能）](#)。

## 範例

下列範例會宣告使用和修飾詞的 32 位 `_sptr` 指標 `_uptr`、將每個 32 位指標指派給 64 位指標變數，然後顯示每個 64 位指標的十六進位值。此範例是以原生 64 位元編譯器編譯，並且在 64 位元平台上執行。

```

// 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;
}

```

```

Display each 32-bit pointer (as an unsigned 64-bit pointer):
p32d:      000000087654321
p32s:      000000087654321
p32u:      000000087654321

```

```

Display the 64-bit pointer created from each 32-bit pointer:
p32d: p64 = FFFFFFFF87654321
p32s: p64 = FFFFFFFF87654321
p32u: p64 = 000000087654321

```

結束 Microsoft 專有

**另請參閱**

[Microsoft 專有的修飾詞](#)

# `_unaligned`

2020/11/2 • [Edit Online](#)

Microsoft 特有。當您使用修飾詞宣告指標時 `_unaligned`，編譯器會假設指標會定址未對齊的資料。因此，會產生平臺適當的程式碼，以透過指標來處理未對齊的讀取和寫入。

## 備註

這個修飾詞會描述指標所定址之資料的對齊方式。指標本身會假設為對齊。

關鍵字的必要條件會 `_unaligned` 因平臺和環境而異。若未適當地標記資料，可能會導致問題，範圍從效能損失到硬體錯誤。`_unaligned` 修飾詞對 x86 平臺無效。

為了與舊版相容，`_unaligned` `_unaligned` 除非指定了編譯器選項 [停用 [/za](#) (語言擴充功能)]，否則會是同義字。

如需對齊的詳細資訊，請參閱：

- [align](#)
- [alignof 操作](#)
- [pack](#)
- [/zp \(結構成員對齊\)](#)
- [結構對齊範例](#)

## 另請參閱

[關鍵字](#)

# `_w64`

2020/11/2 • [Edit Online](#)

這個 Microsoft 專有關鍵詞已經過時。在早于 Visual Studio 2013 的 Visual Studio 版本中，這可讓您標記變數，因此當您使用 `/Wp64` 進行編譯時，編譯器會回報當您使用 64 位編譯器進行編譯時，將會報告的任何警告。

## 語法

類型 \* `_w64` \*\*\*識別碼

### 參數

#### *type*

這三種類型的其中一種，可能會在程式碼從 32 位移植到 64 位編譯器時發生問題：`int`、`long` 或指標。

#### 標識

您建立的變數的識別項。

## 備註

### IMPORTANT

`/Wp64` 編譯器選項和 `_w64` 關鍵字在 Visual Studio 2010 和 Visual Studio 2013 中已被取代，並從 Visual Studio 2013 開始移除。如果您在 `/Wp64` 命令列上使用編譯器選項，則編譯器會發出命令列警告 D9002。會以無訊息方式 `_w64` 忽略關鍵字。請改用以 64 位平臺為目標的 Microsoft C++ 編譯器，而不是使用此選項和關鍵字來偵測 64 位可攜性問題。如需詳細資訊，請參閱 [設定 64 位、x64 目標的 Visual C++](#)。

其上的任何 `typedef _w64` 必須是 x86 上的 32 位和 x64 上的 64 位。

若要使用早于 Visual Studio 2010 的 Microsoft C++ 編譯器版本來偵測可攜性問題，您 `_w64` 應該在任何在 32 位和 64 位平臺之間變更大小的 `typedef` 上指定關鍵字。針對任何這類類型，`_w64` 必須只出現在 `typedef` 的 32 位定義上。

為了與舊版相容，`_w64` `_w64` 除非指定了編譯器選項 `/za` 停用（語言擴充）功能，否則 `_w64` 是同義字。

`_w64` 如果編譯不使用，則會忽略關鍵字 `/Wp64`。

如需移植至 64 位元的詳細資訊，請參閱下列主題：

- [MSVC 編譯器選項](#)
- [將 32 位元程式碼移植到 64 位元程式碼](#)
- [針對 64 位元 x64 目標設定 Visual C++](#)

## 範例

```
// __w64.cpp
// compile with: /W3 /WP64
typedef int Int_32;
#ifndef _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
}
```

## 另請參閱

[關鍵字](#)

# func

2020/3/25 • [Edit Online](#)

(C++11) 預先定義的 `__func__` 會以隱含方式定義為字串，其中包含封入函式的不合格和未名稱。`__func__`是由C++標準強制的，而且不是 Microsoft 擴充功能。

## 語法

```
__func__
```

## 傳回值

傳回包含函式名稱之以 null 結束的 const char 字元陣列。

## 範例

```
#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;
}
```

## 需求

C++11

# 編譯器 COM 支援

2020/11/2 • [Edit Online](#)

## Microsoft 特定的

Microsoft c + + 編譯器可以直接讀取元件物件模型(COM)類型程式庫，並將內容轉譯成可包含在編譯中的 c + + 原始程式碼。語言延伸模組可用來加速桌面應用程式用戶端上的 COM 程式設計。

藉由使用 `#import` 預處理器指示詞，編譯器可以讀取類型程式庫，並將它轉換成 c + + 標頭檔，將 COM 介面描述為類別。一組 `#import` 屬性可供使用者控制產生類別程式庫標頭檔的內容。

您可以使用 `_declspec` 擴充屬性 UUID 來指派 COM 物件的全域唯一識別碼(GUID)。關鍵字 `_uuidof` 可以用來解壓縮與 COM 物件相關聯的 GUID。另一個 `_declspec` 屬性(`property`)可以用來指定 `get` `set` COM 物件之資料成員的和方法。

提供一組 COM 支援全域函式和類別，以支援 `VARIANT` 和 `BSTR` 類型、執行智慧型指標，以及封裝所擲回的錯誤物件 `_com_raise_error`：

- [編譯器 COM 全域函式](#)
- [\\_bstr\\_t](#)
- [\\_com\\_error](#)
- [\\_com\\_ptr\\_t](#)
- [\\_variant\\_t](#)

結束 Microsoft 專有

## 另請參閱

[編譯器 COM 支援類別](#)

[編譯器 COM 全域函式](#)

# 編譯器 COM 全域函式

2020/3/25 • [Edit Online](#)

## Microsoft 專屬

下列為可用的處理常式：

名稱	說明
<a href="#">_com_raise_error</a>	會擲回 <a href="#">_com_error</a> 以回應失敗。
<a href="#">_set_com_error_handler</a>	取代 COM 錯誤處理所使用的預設函式。
<a href="#">ConvertBSTRToString</a>	將 <code>BSTR</code> 值轉換成 <code>char *</code> 。
<a href="#">ConvertStringToBSTR</a>	將 <code>char *</code> 值轉換成 <code>BSTR</code> 。

END Microsoft 特定的

## 另請參閱

[編譯器 COM 支援類別](#)

[編譯器 COM 支援](#)

# \_com\_raise\_error

2020/4/22 • [Edit Online](#)

## Microsoft 特定的

引發[\\_com\\_error](#)以回應故障。

## 語法

```
void __stdcall _com_raise_error(
    HRESULT hr,
    IErrorInfo* perrinfo = 0
);
```

參數

人力資源

HRESULT 資訊。

佩里*info*

IErrorInfo 物件。

## 備註

`_com_raise_error`( 在 `comdef.h` 中 <定義>) 可以替換為同名和原型的使用者編寫的版本。如果您要使用 `#import`，但是不想要使用 C++ 例外狀況處理，則可以這樣做。在這種情況下，`_com_raise_error` 的使用者版本可能會決定執行 `longjmp` 或顯示消息框並停止。不過，使用者版本不應傳回，因為編譯器 COM 支援程式碼不會預期它傳回。

您還可以使用[\\_set\\_com\\_error\\_handler](#)來替換預設的錯誤處理函數。

預設情況下，`_com_raise_error` 的定義如下：

```
void __stdcall _com_raise_error(HRESULT hr, IErrorInfo* perrinfo) {
    throw _com_error(hr, perrinfo);
}
```

結束微軟的

## 需求

標題: <comdef.h>

Lib: 如果 `wchar_t` 是本機類型編譯器選項，請使用 `comsuppw.lib` 或 `comsuppwd.lib`。如果 `wchar_t` 是本機類型關閉，請使用 `comsupp.lib`。如需詳細資訊，請參閱 [/Zc:wchar\\_t \(wchar\\_t 是原生類型\)](#)。

## 另請參閱

[編譯器 COM 全域函式](#)

[\\_set\\_com\\_error\\_handler](#)

# ConvertStringToBSTR

2020/3/25 • • [Edit Online](#)

## Microsoft 專屬

將 `char *` 值轉換成 `BSTR`。

## 語法

```
BSTR __stdcall ConvertStringToBSTR(const char* pSrc)
```

### 參數

*pSrc*

`char *` 變數。

## 範例

```
// 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);
}
```

```
char * text: Test
BSTR text: Test
```

## END Microsoft 特定的

## 需求

標頭：`<comutil.h>`

Lib：comsuppw.lib 或 comsuppwd.lib（請參閱[/zc: Wchar\\_t](#) (`Wchar_t` 是原生類型) 以取得詳細資訊）

## 另請參閱

[編譯器 COM 全域函式](#)

# ConvertBSTRToString

2020/3/25 • [Edit Online](#)

## Microsoft 專屬

將 `BSTR` 值轉換成 `char *`。

## 語法

```
char* __stdcall ConvertBSTRToString(BSTR pSrc);
```

### 參數

*pSrc*

BSTR 變數。

## 備註

`ConvertBSTRToString` 會配置您必須刪除的字串。

## 範例

```
// 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;
}
```

```
BSTR text: Test
char * text: Test
```

## END Microsoft 特定的

## 需求

標頭：`<comutil.h.h>`

Lib：`comsuppw.lib` 或 `comsuppwd.lib` (請參閱[/zc: Wchar\\_t \(Wchar\\_t 是原生類型\)](#)以取得詳細資訊)

## 另請參閱

[編譯器 COM 全域函式](#)



# \_set\_com\_error\_handler

2020/4/22 • [Edit Online](#)

取代 COM 錯誤處理所使用的預設函式。`_set_com_error_handler`特定於微軟。

## 語法

```
void __stdcall _set_com_error_handler(
    void (__stdcall *pHandler)(
        HRESULT hr,
        IErrorInfo* perrinfo
    )
);
```

### 參數

*pHandler*

取代函式的指標。

**人力資源**

HRESULT 資訊。

**佩里*info***

IErrorInfo 物件。

## 備註

預設情況下，`_com_raise_error`處理所有 COM 錯誤。您可以使用 `_set_com_error_handler`來呼叫自己的錯誤處理函數來更改此行為。

取代函式擁有的簽章必須相當於 `_com_raise_error` 的簽章。

## 範例

```

// _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;
}

```

Exception raised: Unable to establish the connection!

## 需求

**標題:** <comdef.h>

**Lib:** 如果指定 /Zc:wchar\_t 編譯器選項(預設值),請使用 comsuppw.lib 或 comsuppwd.lib。如果指定了 /Zc:wchar\_t- 編譯器選項,請使用 comsupp.lib。有關詳細資訊(包括如何在IDE中設置此選項),請參閱[/Zc:wchar\\_t \(wchar\\_t本機類型\)](#)。

## 另請參閱

[編譯器 COM 全域函式](#)

# 編譯器 COM 支援類別

2020/3/25 • [Edit Online](#)

## Microsoft 專屬

標準類別用來支援某些 COM 類型。類別定義于 <comdef.h 中, > 和從類型程式庫產生的標頭檔。

II	II
<code>_bstr_t</code>	包裝 <code>BSTR</code> 類型可提供有用的運算子和方法。
<code>_com_error</code>	定義在大多數失敗中 <code>_com_raise_error</code> 擲回的錯誤物件。
<code>_com_ptr_t</code>	封裝 COM 介面指標, 並將 <code>AddRef</code> 、 <code>Release</code> 和 <code>QueryInterface</code> 所需的呼叫自動化。
<code>_variant_t</code>	包裝 <code>VARIANT</code> 類型可提供有用的運算子和方法。

END Microsoft 特定的

## 另請參閱

[編譯器 COM 支援](#)

[編譯器 COM 全域函式](#)

[C++ 語言參考](#)

# \_bstr\_t 類別

2020/11/2 • [Edit Online](#)

## Microsoft 特定的

`_bstr_t` 物件封裝 **BSTR 資料類型**。類別會 `SysAllocString`、`SysFreeString`、`BSTR` 在適當的情況之下，透過和和其他 API 的函式呼叫來管理資源配置和解除配置。`_Bstr_t` 類別會使用參考計數來避免過多的額外負荷。

## 營造

```	```
<code>_bstr_t</code>	建構 <code>_bstr_t</code> 物件。

## Operations

```	```
<b>指派</b>	將 <code>BSTR</code> 複製到 <code>BSTR</code> 所包裝的 <code>_bstr_t</code> 中。
<b>附加</b>	將 <code>_bstr_t</code> 包裝函式連結至 <code>BSTR</code> 。
<b>copy</b>	建構已封裝 <code>BSTR</code> 的複本。
<b>卸離</b>	傳回 <code>BSTR</code> 所包裝的 <code>_bstr_t</code> ，並將 <code>BSTR</code> 與 <code>_bstr_t</code> 中斷連結。
<b>GetAddress</b>	指向 <code>BSTR</code> 所包裝的 <code>_bstr_t</code> 。
<b>GetBSTR</b>	指向由 <code>BSTR</code> 所包裝之 <code>_bstr_t</code> 的開頭。
<b>length</b>	傳回 <code>_bstr_t</code> 中的字元數。

## 運算子

```	```
<b>運算子 =</b>	將新值指派給現有的 <code>_bstr_t</code> 物件。
<b>運算子 +=</b>	將字元附加至 <code>_bstr_t</code> 物件的結尾。
<b>運算子 +</b>	串連兩個字串。
<b>運算子 !</b>	檢查封裝 <code>BSTR</code> 是否為 Null 字串。
<b>operator ==、!=、&lt;、&gt;、&lt;=、&gt;=</b>	比較兩個 <code>_bstr_t</code> 物件。
<b>operator wchar_t *   char*</b>	將指標擷取至封裝的 Unicode 或多位元組的 <code>BSTR</code> 物件。

## 規格需求

標頭 : <comutil.h>

Lib: comsuppw.lib 或 comsuppwd.lib.lib (請參閱 [/zc: Wchar\\_t \(Wchar\\_t 是原生類型\)](#) 以取得詳細資訊)

## 另請參閱

[編譯器 COM 支援類別](#)

# `_bstr_t` 成員函式

2020/3/25 • [Edit Online](#)

如需 `_bstr_t` 成員函式的詳細資訊，請參閱[\\_Bstr\\_t 類別](#)。

## 另請參閱

[\\_bstr\\_t 類別](#)

# \_bstr\_t::Assign

2020/11/2 • [Edit Online](#)

Microsoft 特定的

將 `BSTR` 複製到 `BSTR` 所包裝的 `_bstr_t` 中。

## 語法

```
void Assign(  
    BSTR s  
)
```

參數

今日

要複製到 `BSTR` 所包裝之 `BSTR` 中的 `_bstr_t`。

## 備註

`Assign` 會執行二進位複製，這表示 `BSTR` 不論內容為何，都會複製的整個長度。

## 範例

```

// _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("SysAllocatedString")));
    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);

    // reuse 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));
}

```

```
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
```

結束 Microsoft 專有

另請參閱

[\\_bstr\\_t 類別](#)

# \_bstr\_t::Attach

2020/4/22 • [Edit Online](#)

Microsoft 特定的

將 `_bstr_t` 包裝函式連結至 `BSTR`。

## 語法

```
void Attach(  
    BSTR s  
)
```

參數

`s`

要與 `BSTR` 變數產生關聯或指派給該變數的 `_bstr_t`。

## 備註

如果 `_bstr_t` 先前已附加至另一個 `BSTR`，且其他 `_bstr_t` 變數都不會使用 `BSTR`，則 `_bstr_t` 將會清除 `BSTR` 資源。

## 範例

有關使用額外的範例,請參閱[\\_bstr\\_t::分配](#)。

結束微軟的

## 另請參閱

[\\_bstr\\_t 類別](#)

# `_bstr_t::_bstr_t`

2020/11/2 • [Edit Online](#)

Microsoft 特定的

建構 `_bstr_t` 物件。

## 語法

```
_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
);
```

參數

*s1*

要複製的 `_bstr_t` 物件。

*s2*

多位元組字串。

*s3*

Unicode 字串

*var*

`_Variant_t` 物件。

*bstr*

現有的 `BSTR` 物件。

*fCopy*

如果為 `false`，則會將 *bstr* 引數附加至新的物件，而不需要藉由呼叫來建立複本 `SysAllocString`。

## 備註

下表將說明 `_bstr_t` 建構函式。

參數	說明
<code>_bstr_t( )</code>	構造 <code>_bstr_t</code> 封裝 null 物件的預設物件 <code>BSTR</code> 。

||||

||

`_bstr_t( _bstr_t& s1 )`

將 `_bstr_t` 物件建構為另一個物件的複本。

這是淺層複製，會遞增封裝物件的參考計數，`BSTR` 而不是建立新的。

`_bstr_t( char* s2 )`

透過呼叫 `_bstr_t` 建構 `SysAllocString` 以建立新的 `BSTR` 物件並加以封裝。

這個建構函式會先執行多位元組對 Unicode 的轉換。

`_bstr_t( wchar_t* s3 )`

透過呼叫 `_bstr_t` 建構 `SysAllocString` 以建立新的 `BSTR` 物件並加以封裝。

`_bstr_t( _variant_t& var )`

先從封裝的 VARIANT 物件擷取 `_bstr_t` 物件，以從 `_variant_t` 物件建構 `BSTR` 物件。

`_bstr_t( BSTR bstr, bool fCopy )`

從現有的 `_bstr_t` 建構 `BSTR` 物件 (而非 `wchar_t*` 字串)。如果 `fCopy` 為 `false`，則提供的 `BSTR` 會附加至新的物件，而不會使用建立新的複本 `SysAllocString`。

類型程式庫標題中的包裝函式會使用此建構函式封裝，並取得以介面方法傳回之 `BSTR` 的擁有權。

結束 Microsoft 專有

## 另請參閱

[\\_bstr\\_t 類別](#)

[\\_variant\\_t 類別](#)

# \_bstr\_t::copy

2020/11/2 • • [Edit Online](#)

Microsoft 特定的

建構已封裝 `BSTR` 的複本。

## 語法

```
BSTR copy( bool fCopy = true ) const;
```

參數

*fCopy*

如果為 `true`，則`copy`會傳回包含的複本 `BSTR`，否則`COPY`會傳回實際的 `BSTR`。

## 備註

傳回新配置已封裝 `BSTR` 物件的複本。

## 範例

```
STDMETHODIMP CAlertMsg::get_ConnectionStr(BSTR *pVal){ // m_bsConStr is _bstr_t
    *pVal = m_bsConStr.copy();
}
```

結束 Microsoft 專有

## 另請參閱

[\\_bstr\\_t 類別](#)

# \_bstr\_t::Detach

2020/3/25 • • [Edit Online](#)

## Microsoft 專屬

傳回 `BSTR` 所包裝的 `_bstr_t`，並將 `BSTR` 與 `_bstr_t` 中斷連結。

## 語法

```
BSTR Detach( ) throw;
```

## 傳回值

`BSTR` 包裝 `_bstr_t`。

## 範例

如需使用卸離的範例，請參閱[\\_Bstr\\_t::Assign](#)。

END Microsoft 特定的

## 另請參閱

[\\_bstr\\_t 類別](#)

# \_bstr\_t::GetAddress

2020/3/25 • [Edit Online](#)

Microsoft 專屬

釋放任何現有字串並傳回新配置字串的位址。

## 語法

```
BSTR* GetAddress( );
```

## 傳回值

由 `BSTR` 包裝的 `_bstr_t` 指標。

## 備註

`GetAddress` 會影響共用 `BSTR` 的所有 `_bstr_t` 物件。有一個以上的 `_bstr_t` 可以透過使用複製的函式和 `operator =` 來共用 `BSTR`。

## 範例

如需使用 `GetAddress` 的範例，請參閱 [\\_Bstr\\_t::Assign](#)。

END Microsoft 特定的

## 另請參閱

`_bstr_t` 類別

# `_bstr_t::GetBSTR`

2020/3/25 • [Edit Online](#)

## Microsoft 專屬

指向由 `BSTR` 所包裝之 `_bstr_t` 的開頭。

## 語法

```
BSTR& GetBSTR( );
```

## 傳回值

由 `BSTR` 所包裝之 `_bstr_t` 的開頭。

## 備註

`GetBSTR` 會影響共用 `BSTR` 的所有 `_bstr_t` 物件。有一個以上的 `_bstr_t` 可以透過使用複製的函式和 `operator =` 來共用 `BSTR`。

## 範例

如需使用`GetBSTR`的範例，請參閱[\\_Bstr\\_t::Assign](#)。

END Microsoft 特定的

## 另請參閱

[\\_bstr\\_t 類別](#)

# \_bstr\_t::length

2020/3/25 • [Edit Online](#)

Microsoft 專屬

在 `_bstr_t` 中傳回封裝 `BSTR` 的字元數目，不計結束的 null 字元。

## 語法

```
unsigned int length( ) const throw( );
```

## 備註

END Microsoft 特定的

## 另請參閱

[\\_bstr\\_t 類別](#)

# `_bstr_t` 運算子

2020/3/25 • [Edit Online](#)

如需 `_bstr_t` 運算子的詳細資訊，請參閱[\\_Bstr\\_t 類別](#)。

## 另請參閱

[\\_bstr\\_t 類別](#)

# `_bstr_t::operator =`

2020/3/25 • [Edit Online](#)

Microsoft 專屬

將新值指派給現有的 `_bstr_t` 物件。

## 語法

```
_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);
```

參數

*s1*

`_bstr_t` 物件，將被指派給現有的 `_bstr_t` 物件。

*s2*

多位元組字串，將被指派給現有的 `_bstr_t` 物件。

*s3*

Unicode 字串，將被指派給現有的 `_bstr_t` 物件。

*var*

`_variant_t` 物件，將被指派給現有的 `_bstr_t` 物件。

END Microsoft 特定的

## 範例

如需使用`operator =` 的範例，請參閱[\\_Bstr\\_t::Assign](#)。

## 另請參閱

[\\_bstr\\_t 類別](#)

# \_bstr\_t::operator += 、 +

2020/3/25 • [Edit Online](#)

## Microsoft 專屬

將字元附加至 `_bstr_t` 物件的結尾或串連兩個字串。

## 語法

```
_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);
```

### 參數

*s1*

`_bstr_t` 物件。

*s2*

多位元組字串。

*s3*

Unicode 字串。

## 備註

這些運算子會執行字串串連：

- `operator += ( s1 )` 將封裝 `BSTR` 中的字元附加至此物件封裝 `BSTR` 的結尾。
- `operator + ( s1 )` 傳回新的 `_bstr_t`，這是藉由串連這個物件的 `BSTR` 與 *s1* 所形成。
- `operator + ( s2 | s1 )` 傳回新的 `_bstr_t`，這是藉由串連多位元組字元串 *s2*（轉換成 Unicode），並將 `BSTR` 封裝在 *s1* 中所形成。
- `operator + ( s3, s1 )` 傳回新的 `_bstr_t`，其由串連 Unicode 字串 *s3* 與以 *s1* 封裝的 `BSTR` 形成。

END Microsoft 特定的

## 另請參閱

`_bstr_t` 類別

# \_bstr\_t::operator !

2020/11/2 • • [Edit Online](#)

Microsoft 特定的

檢查封裝的 `BSTR` 是否為 Null 字串。

## 語法

```
bool operator!( ) const throw( );
```

## 傳回值

如果是，則會傳回 `true`，否則會傳回 `false`。

結束 Microsoft 專有

## 另請參閱

`_bstr_t` 類別

# \_bstr\_t 關係運算子

2020/11/2 • [Edit Online](#)

Microsoft 特定的

比較兩個 `_bstr_t` 物件。

## 語法

```
bool operator!( ) 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( );
bool operator>=(const _bstr_t& str) const throw( );
```

## 備註

這些運算子會針對兩個 `_bstr_t` 物件進行字彙上的比較。`true` 如果比較保留，則運算子會傳回，否則會傳回 `false`。

結束 Microsoft 專有

## 另請參閱

`_bstr_t` 類別

# \_bstr\_t::wchar\_t\*、\_bstr\_t::char\*

2020/3/25 • • [Edit Online](#)

Microsoft 專屬

將 BSTR 字元做為窄字元或寬字元陣列傳回。

## 語法

```
operator const wchar_t*( ) const throw( );
operator wchar_t*( ) const throw( );
operator const char*( ) const;
operator char*( ) const;
```

## 備註

這些運算子可用來擷取 **BSTR** 物件所封裝的字元資料。指派新值給傳回的指標並不會修改原始 BSTR 資料。

END Microsoft 特定的

## 另請參閱

[\\_bstr\\_t 類別](#)

# \_com\_error 類別

2020/11/2 • [Edit Online](#)

## Microsoft 特定的

\_Com\_error物件代表由類型程式庫或其中一個 com 支援類別所產生的標頭檔中，錯誤處理包裝函式所偵測到的例外狀況條件。\_Com\_error類別會封裝 HRESULT 錯誤碼和任何相關聯的 `IErrorInfo Interface` 物件。

### 營造

參數	說明
<code>_com_error</code>	結構 _com_error 物件。

### 運算子

參數	說明
<code>運算子 =</code>	將現有的 _com_error 物件指派給另一個物件。

### 解壓縮函數

參數	說明
<code>錯誤</code>	抓取傳遞給函式的 HRESULT。
<code>ErrorInfo</code>	抓取傳遞給函式的 <code>IErrorInfo</code> 物件。
<code>WCode</code>	抓取對應至封裝 HRESULT 的16位錯誤碼。

### IErrorInfo 函式

參數	說明
<code>說明</code>	呼叫 <code>IErrorInfo::GetDescription</code> 函數。
<code>HelpCoNtext</code>	呼叫 <code>IErrorInfo::GetHelpContext</code> 函數。
<code>HelpFile</code>	呼叫 <code>IErrorInfo::GetHelpFile</code> 函式
<code>Source</code>	呼叫 <code>IErrorInfo::GetSource</code> 函數。
<code>GUID</code>	呼叫 <code>IErrorInfo::GetGUID</code> 函數。

### 格式化訊息解壓縮

參數	說明
<code>ErrorMessage</code>	抓取儲存在 _com_error 物件中之 HRESULT 的字串訊息。

### ExepInfo.wCode 至 HRESULT 對應程式數目

HRESULTToWCode	將32位 HRESULT 對應至16位 <code>wCode</code> 。
WCodeToHRESULT	將16位對應 <code>wCode</code> 至32位 HRESULT。

結束 Microsoft 專有

## 規格需求

標頭 : <comdef.h>

Lib: 如需詳細資訊, comsuppw.lib 或 comsuppwd.lib (參閱/zc: wchar\_t (Wchar\_t 是原生類型))

## 另請參閱

編譯器 COM 支援類別

IErrorInfo 介面

# \_com\_error 成員函式

2020/3/25 • [Edit Online](#)

如需 \_com\_error 成員函式的詳細資訊，請參閱 [\\_com\\_error 類別](#)。

[另請參閱](#)

[\\_com\\_error 類別](#)

# \_com\_error::\_com\_error

2020/3/25 • • [Edit Online](#)

Microsoft 專屬

結構 \_com\_error 物件。

## 語法

```
_com_error(
    HRESULT hr,
    IErrorInfo* perrinfo = NULL,
    bool fAddRef=false) throw( );

_com_error( const _com_error& that ) throw( );
```

參數

工時

HRESULT 資訊。

*perrinfo*

IErrorInfo 物件。

*fAddRef*

預設會使此函式在非 null 的 IErrorInfo 介面上呼叫 AddRef。這會在將介面擁有權傳遞至 \_com\_error 物件的一般情況下，提供正確的參考計數，例如：

```
throw _com_error(hr, perrinfo);
```

如果您不想讓程式碼將擁有權轉移給 \_com\_error 物件，而 AddRef 需要在 \_com\_error 的析構函式中位移

Release，請依照下列方式來建立物件：

```
_com_error err(hr, perrinfo, true);
```

對於

現有的 \_com\_error 物件。

## 備註

第一個函式會建立新的物件，並指定 HRESULT 和選擇性的 IErrorInfo 物件。第二個會建立現有 \_com\_error 物件的複本。

END Microsoft 特定的

## 另請參閱

[\\_com\\_error 類別](#)

# \_com\_error::Description

2020/3/25 • • [Edit Online](#)

## Microsoft 專屬

呼叫 `IErrorInfo::GetDescription` 函式。

## 語法

```
_bstr_t Description( ) const;
```

## 傳回值

傳回 `_com_error` 物件中所記錄之 `IErrorInfo` 物件的 `IErrorInfo::GetDescription` 結果。產生的 `BSTR` 會封裝在 `_bstr_t` 物件內。如果未記錄任何 `IErrorInfo`，則會傳回空的 `_bstr_t`。

## 備註

呼叫 `IErrorInfo::GetDescription` 函式，並抓取 `_com_error` 物件內記錄的 `IErrorInfo`。呼叫 `IErrorInfo::GetDescription` 方法時的任何失敗都會被忽略。

END Microsoft 特定的

## 另請參閱

[\\_com\\_error 類別](#)

# \_com\_error::Error

2020/3/25 • • [Edit Online](#)

Microsoft 專屬

抓取傳遞至此函數的 HRESULT。

## 語法

```
HRESULT Error( ) const throw( );
```

## 傳回值

傳遞至此函式的原始 HRESULT 專案。

## 備註

抓取 `_com_error` 物件中封裝的 HRESULT 專案。

END Microsoft 特定的

## 另請參閱

[\\_com\\_error 類別](#)

# \_com\_error::ErrorInfo

2020/3/25 • • [Edit Online](#)

Microsoft 專屬

抓取傳遞至此函式的原始 `IErrorInfo` 物件。

## 語法

```
IErrorInfo * ErrorInfo( ) const throw( );
```

## 傳回值

傳入建構函式的原始 `IErrorInfo` 項目。

## 備註

抓取 `_com_error` 物件中封裝的 `IErrorInfo` 專案，如果沒有記錄 `IErrorInfo` 專案，則為 Null。呼叫端完成使用時，必須在傳回的物件上呼叫 `Release`。

END Microsoft 特定的

## 另請參閱

[\\_com\\_error 類別](#)

# \_com\_error::ErrorMessage

2020/3/25 • [Edit Online](#)

Microsoft 專屬

抓取儲存在 `_com_error` 物件中之 HRESULT 的字串訊息。

## 語法

```
const TCHAR * ErrorMessage( ) const throw( );
```

## 傳回值

傳回 `_com_error` 物件中所記錄之 HRESULT 的字串訊息。如果 HRESULT 是對應的16位 `wCode`, 則會傳回一般訊息「`IDispatch error #<wCode>`」。如果找不到訊息, 則會傳回一般訊息「`Unknown error #<HRESULT>`」。傳回的字串會是 Unicode 字串或多位元組字串, 視 `_UNICODE` 巨集的狀態而定。

## 備註

針對在 `_com_error` 物件內記錄的 HRESULT, 抓取適當的系統郵件內文。系統郵件內文是藉由呼叫 Win32 `FormatMessage`函數來取得。傳回的字串會由 `FormatMessage` API 配置, 並且在終結 `_com_error` 物件時釋放。

END Microsoft 特定的

## 另請參閱

[\\_com\\_error 類別](#)

# \_com\_error::GUID

2020/3/25 • • [Edit Online](#)

Microsoft 專屬

呼叫 `IErrorInfo::GetGUID` 函式。

## 語法

```
GUID GUID( ) const throw( );
```

## 傳回值

傳回 `_com_error` 物件中所記錄之 `IErrorInfo` 物件的 `IErrorInfo::GetGUID` 結果。如果未記錄任何 `IErrorInfo` 物件，則會傳回 `GUID_NULL`。

## 備註

呼叫 `IErrorInfo::GetGUID` 方法時的任何失敗都會被忽略。

END Microsoft 特定的

## 另請參閱

[\\_com\\_error 類別](#)

# \_com\_error::HelpContext

2020/3/25 • • [Edit Online](#)

Microsoft 專屬

呼叫 `IErrorInfo::GetHelpContext` 函式。

## 語法

```
DWORD HelpContext( ) const throw( );
```

## 傳回值

傳回 `_com_error` 物件中所記錄之 `IErrorInfo` 物件的 `IErrorInfo::GetHelpContext` 結果。如果未記錄任何 `IErrorInfo` 物件，則會傳回零。

## 備註

呼叫 `IErrorInfo::GetHelpContext` 方法時的任何失敗都會被忽略。

END Microsoft 特定的

## 另請參閱

[\\_com\\_error 類別](#)

# \_com\_error::HelpFile

2020/3/25 • • [Edit Online](#)

## Microsoft 專屬

呼叫 `IErrorInfo::GetHelpFile` 函式。

## 語法

```
_bstr_t HelpFile() const;
```

## 傳回值

傳回 `_com_error` 物件中所記錄之 `IErrorInfo` 物件的 `IErrorInfo::GetHelpFile` 結果。產生的 BSTR 會封裝在 `_bstr_t` 物件內。如果未記錄任何 `IErrorInfo`，則會傳回空的 `_bstr_t`。

## 備註

呼叫 `IErrorInfo::GetHelpFile` 方法時的任何失敗都會被忽略。

END Microsoft 特定的

## 另請參閱

[\\_com\\_error 類別](#)

# \_com\_error::HRESULTToWCode

2020/3/25 • [Edit Online](#)

Microsoft 專屬

將 32-bit HRESULT 對應至16位 `wCode`。

## 語法

```
static WORD HRESULTToWCode(
    HRESULT hr
) throw( );
```

參數

工時

要對應至16位 `wCode` 的32位 HRESULT。

## 傳回值

從32位 HRESULT 對應的16位 `wCode`。

## 備註

如需詳細資訊，請參閱[\\_com\\_error:: WCode](#)。

END Microsoft 特定的

## 另請參閱

[\\_com\\_error::WCode](#)

[\\_com\\_error::WCodeToHRESULT](#)

[\\_com\\_error 類別](#)

# \_com\_error::Source

2020/3/25 • [Edit Online](#)

## Microsoft 專屬

呼叫 `IErrorInfo::GetSource` 函式。

## 語法

```
_bstr_t Source() const;
```

## 傳回值

傳回 `_com_error` 物件中所記錄之 `IErrorInfo` 物件的 `IErrorInfo::GetSource` 結果。產生的 `BSTR` 會封裝在 `_bstr_t` 物件內。如果未記錄任何 `IErrorInfo`，則會傳回空的 `_bstr_t`。

## 備註

呼叫 `IErrorInfo::GetSource` 方法時的任何失敗都會被忽略。

END Microsoft 特定的

## 另請參閱

[\\_com\\_error 類別](#)

# \_com\_error::WCode

2020/3/25 • [Edit Online](#)

Microsoft 專屬

抓取對應至封裝 HRESULT 的16位錯誤碼。

## 語法

```
WORD WCode( ) const throw( );
```

## 傳回值

如果 HRESULT 在0x80040200 到0X8004ffff 內的範圍內，`WCode` 方法會傳回 HRESULT 減 0x80040200;否則，它會傳回零。

## 備註

`WCode` 方法是用來復原在 COM 支援程式碼中發生的對應。`dispinterface` 屬性或方法的包裝函式，會呼叫封裝引數並呼叫 `IDispatch::Invoke` 的支援常式。傳回時，如果傳回 `DISP_E_EXCEPTION` 的失敗 HRESULT，則會從傳遞給 `IDispatch::Invoke` 的 `EXCEPINFO` 結構中取出錯誤資訊。錯誤碼可以是儲存在 `EXCEPINFO` 結構的 `wCode` 成員中的16位值，或 `EXCEPINFO` 結構之 `scode` 成員中的完整32位值。如果傳回16位 `wCode`，則必須先將它對應到32位的失敗 HRESULT。

END Microsoft 特定的

## 另請參閱

[\\_com\\_error::HRESULTToWCode](#)  
[\\_com\\_error::WCodeToHRESULT](#)  
[\\_com\\_error 類別](#)

# \_com\_error::WCodeToHRESULT

2020/3/25 • [Edit Online](#)

## Microsoft 專屬

將16位 *wCode* 對應至32位 HRESULT。

## 語法

```
static HRESULT WCodeToHRESULT(
    WORD wCode
) throw( );
```

### 參數

*wCode*

要對應至32位 HRESULT 的16位 *wCode*。

## 傳回值

從16位 *wCode* 對應的32位 HRESULT。

## 備註

請參閱 [WCode 成員函式](#)。

END Microsoft 特定的

## 另請參閱

[\\_com\\_error::WCode](#)

[\\_com\\_error::HRESULTToWCode](#)

[\\_com\\_error 類別](#)

# \_com\_error 運算子

2020/3/25 • [Edit Online](#)

如需 \_com\_error 運算子的詳細資訊，請參閱 [\\_com\\_error 類別](#)。

## 另請參閱

[\\_com\\_error 類別](#)

# \_com\_error::operator =

2020/3/25 • • [Edit Online](#)

Microsoft 專屬

將現有的 `_com_error` 物件指派給另一個物件。

## 語法

```
_com_error& operator = (
    const _com_error& that
) throw ( );
```

參數

對於

`_com_error` 物件。

END Microsoft 特定的

## 另請參閱

[\\_com\\_error 類別](#)

# \_com\_ptr\_t 類別

2020/11/2 • [Edit Online](#)

## Microsoft 特定的

\_Com\_ptr\_t 物件會封裝 com 介面指標，而且稱為「智慧型」指標。此範本類別會透過對成員函式的函式呼叫來管理資源配置和解除配置 `IUnknown` : `QueryInterface` 、 `AddRef` 和 `Release` 。

智慧型指標通常是由 `_COM_SMARTPTR_TYPEDEF` 巨集提供的 `typedef` 定義所參考。此宏會採用介面名稱和 `IID`，並使用介面的名稱加上後置詞，宣告 `_com_ptr_t` 的特製化 `Ptr` 。例如：

```
_COM_SMARTPTR_TYPEDEF(IMyInterface, __uuidof(IMyInterface));
```

宣告 `_com_ptr_t` 特製化 `IMyInterfacePtr` 。

一組函式 [範本](#)，而不是此樣板類別的成員，支援與比較運算子右邊的智慧型指標進行比較。

## 營造

II	II
<code>_com_ptr_t</code>	結構 <code>_com_ptr_t</code> 物件。

## 低階作業

II	II
<code>AddRef</code>	<code>AddRef</code> <code>IUnknown</code> 在封裝的介面指標上呼叫的成員函式。
<code>附加</code>	封裝這個智慧型指標類型的一般介面指標。
<code>CreateInstance</code>	使用指定的或，建立物件的新 <code>CLSID</code> 實例 <code>ProgID</code> 。
<code>卸離</code>	擷取和傳回封裝的介面指標。
<code>GetActiveObject</code>	附加至給定或之物件的現有實例 <code>CLSID</code> <code>ProgID</code> 。
<code>GetInterfacePtr</code>	傳回封裝的介面指標。
<code>QueryInterface</code>	<code>QueryInterface</code> <code>IUnknown</code> 在封裝的介面指標上呼叫的成員函式。
<code>版本</code>	<code>Release</code> <code>IUnknown</code> 在封裝的介面指標上呼叫的成員函式。

## 運算子

II	II
<code>運算子 =</code>	將新值指派給現有的 <code>_com_ptr_t</code> 物件。

operators ==、!=、<、>、<=、>=	將智慧型指標物件與另一個智慧型指標、一般介面指標或 NULL 進行比較。
擷取器	擷取封裝的 COM 介面指標。

結束 Microsoft 專有

## 規格需求

標頭: <comip.h>

Lib: comsuppw.lib 或 comsuppwd.lib.lib (請參閱 [/zc: Wchar\\_t \(Wchar\\_t 是原生類型\)](#) 以取得詳細資訊)

## 另請參閱

[編譯器 COM 支援類別](#)

# \_com\_ptr\_t 成員函式

2020/3/25 • [Edit Online](#)

如需 \_com\_ptr\_t 成員函式的詳細資訊，請參閱[\\_com\\_ptr\\_t 類別](#)。

## 另請參閱

[\\_com\\_ptr\\_t 類別](#)

# \_com\_ptr\_t::\_com\_ptr\_t

2020/11/2 • • [Edit Online](#)

Microsoft 特定的

結構 \_com\_ptr\_t 物件。

## 語法

```
// 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(  
    const string& strCLSID,
```

```

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
);

```

## 參數

*pInterface*

原始的介面指標。

*fAddRef*

如果為 `true`，`AddRef` 則會呼叫以遞增封裝之介面指標的參考計數。

*cp*

`_Com_ptr_t` 物件。

*p&id*

原始介面指標，其類型與這個 `_com_ptr_t` 物件的智慧型指標類型不同。

*varSrc*

`_variant_t` 物件。

*clsid*

`CLSID` Coclass 的。

*dwClsCoNtext*

執行中的可執行程式碼內容。

*lpcStr*

保存 `CLSID` (開頭為 "{}") 或的多位元組字元串 `ProgID`。

*pOuter*

匯總的外部未知。

結束 Microsoft 專有

## 另請參閱

[\\_com\\_ptr\\_t 類別](#)

# \_com\_ptr\_t::AddRef

2020/4/22 • • [Edit Online](#)

Microsoft 特定的

調用 `AddRef` 封裝介面指標 `IUnknown` 上的成員函數。

## 語法

```
void AddRef( );
```

## 備註

呼叫 `IUnknown::AddRef` 封裝的介面指標,如果指標為 `E_POINTER` `NULL`,則引發錯誤。

結束微軟的

## 另請參閱

`_com_ptr_t` 類

# \_com\_ptr\_t::Attach

2020/11/2 • • [Edit Online](#)

Microsoft 特定的

封裝這個智慧型指標類型的一般介面指標。

## 語法

```
void Attach( Interface* pInterface ) throw( );
void Attach( Interface* pInterface, bool fAddRef ) throw( );
```

參數

*pInterface*

原始的介面指標。

*fAddRef*

如果是 `true`，則 `AddRef` 會呼叫。如果是 `false`，物件會 `_com_ptr_t` 取得原始介面指標的擁有權，而不需要呼叫 `AddRef`。

## 備註

- `Attach( pInterface )` `AddRef` 不會呼叫。介面的擁有權會傳遞至這個 `_com_ptr_t` 物件。`Release` 呼叫以遞減先前封裝之指標的參考計數。
- `Attach( pInterface, fAddRef )` 如果 `fAddRef` 為 `true`，`AddRef` 則會呼叫來遞增封裝之介面指標的參考計數。如果 `fAddRef` 為 `false`，此 `_com_ptr_t` 物件會取得原始介面指標的擁有權，而不會呼叫 `AddRef`。  
`Release` 呼叫以遞減先前封裝之指標的參考計數。

結束 Microsoft 專有

## 另請參閱

[\\_com\\_ptr\\_t 類別](#)

# \_com\_ptr\_t::CreateInstance

2020/3/25 • • [Edit Online](#)

## Microsoft 專屬

指定 `CLSID` 或 `ProgID`，建立物件的新實例。

## 語法

```
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( );
```

### 參數

*rclsid*

物件的 `CLSID`。

*clsidString*

保存 `CLSID` (以 "{" 為開頭)或 `ProgID` 的 Unicode 字串。

*clsidStringA*

使用 ANSI 字碼頁的多位元組字元串，其中包含 `CLSID` (以 "{" 為開頭)或 `ProgID`。

*dwClCoNtext*

執行中的可執行程式碼內容。

*pOuter*

匯總的外部未知。

## 備註

這些成員函式會呼叫 `CoCreateInstance` 建立新的 COM 物件，然後查詢這個智慧型指標的介面類型。然後產生的指標就會封裝在這個 `_com_ptr_t` 物件內。呼叫 `Release` 以遞減先前封裝之指標的參考計數。此常式會傳回 `HRESULT` 以表示成功或失敗。

- `CreateInstance ( rclsid, dwClCoNtext)` 指定 `CLSID`，建立物件的新執行中實例。
- `CreateInstance ( clsidString, dwClCoNtext)` 指定保留 `CLSID` (以 "{" 為開頭)或 `ProgID` 的 Unicode 字串，建立物件的新執行中實例。
- `CreateInstance ( clsidStringA, dwClCoNtext)` 指定保存 `CLSID` (開頭為 "{ ")或 `ProgID` 的多位元組字元串，建立物件的新執行中實例。呼叫 `MultiByteToWideChar`，它會假設字串位於 ANSI 字碼頁，而不是 OEM 字碼頁。

END Microsoft 特定的

另請參閱

[\\_com\\_ptr\\_t 類別](#)

# \_com\_ptr\_t::Detach

2020/3/25 • • [Edit Online](#)

Microsoft 專屬

擷取和傳回封裝的介面指標。

## 語法

```
Interface* Detach( ) throw( );
```

## 備註

擷取和傳回封裝的介面指標，然後將封裝的指標儲存區清除為 NULL。這麼做會從封裝中移除介面指標。您可以視需要在傳回的介面指標上呼叫 `Release`。

END Microsoft 特定的

## 另請參閱

[\\_com\\_ptr\\_t 類別](#)

# \_com\_ptr\_t::GetActiveObject

2020/3/25 • • [Edit Online](#)

## Microsoft 專屬

指定 `CLSID` 或 `ProgID`，附加至物件的現有實例。

## 語法

```
HRESULT GetActiveObject(
    const CLSID& rclsid
) throw( );
HRESULT GetActiveObject(
    LPCWSTR clsidString
) throw( );
HRESULT GetActiveObject(
    LPCSTR clsidStringA
) throw( );
```

### 參數

*rclsid*

物件的 `CLSID`。

*clsidString*

保存 `CLSID` (以 "{" 為開頭)或 `ProgID` 的 Unicode 字串。

*clsidStringA*

使用 ANSI 字碼頁的多位元組字元串，其中包含 `CLSID` (以 "{" 為開頭)或 `ProgID`。

## 備註

這些成員函式會呼叫 `GetActiveObject`，以取得已向 OLE 註冊之執行中物件的指標，然後查詢此智慧型指標的介面類別型。然後產生的指標就會封裝在這個 `_com_ptr_t` 物件內。呼叫 `Release` 以遞減先前封裝之指標的參考計數。此常式會傳回 `HRESULT` 以表示成功或失敗。

- `GetActiveObject ( rclsid )` 指定 `CLSID`，附加至物件的現有實例。
- `GetActiveObject ( clsidString )` 附加至物件的現有實例，並指定保留 `CLSID` (以 "{" 為開頭)或 `ProgID` 的 Unicode 字串。
- `GetActiveObject ( clsidStringA )` 附加至物件的現有實例，並指定包含 `CLSID` (以 "{" 為開頭)或 `ProgID` 的多位元組字元字串。呼叫 [MultiByteToWideChar](#)，它會假設字串位於 ANSI 字碼頁，而不是 OEM 字碼頁。

END Microsoft 特定的

## 另請參閱

[\\_com\\_ptr\\_t 類別](#)

# \_com\_ptr\_t::GetInterfacePtr

2020/3/25 • [Edit Online](#)

Microsoft 專屬

傳回封裝的介面指標。

## 語法

```
Interface* GetInterfacePtr( ) const throw( );
Interface*& GetInterfacePtr() throw();
```

## 備註

傳回封裝的介面指標，可能為 NULL。

END Microsoft 特定的

## 另請參閱

[\\_com\\_ptr\\_t 類別](#)

# \_com\_ptr\_t::QueryInterface

2020/3/25 • [Edit Online](#)

Microsoft 專屬

在封裝的介面指標上呼叫 `IUnknown` 的 `QueryInterface` 成員函式。

## 語法

```
template<typename _InterfaceType> HRESULT QueryInterface (
    const IID& iid,
    _InterfaceType*& p
) throw( );
template<typename _InterfaceType> HRESULT QueryInterface (
    const IID& iid,
    _InterfaceType** p
) throw( );
```

參數

*iid*

介面指標的 `IID`。

*p*

原始介面指標。

## 備註

使用指定的 `IID` 在封裝的介面指標上呼叫 `IUnknown::QueryInterface`，並在 *p* 中傳回產生的原始介面指標。此常式會傳回 `HRESULT` 以表示成功或失敗。

END Microsoft 特定的

## 另請參閱

[\\_com\\_ptr\\_t 類別](#)

# \_com\_ptr\_t::Release

2020/4/22 • • [Edit Online](#)

Microsoft 特定的

調用封裝Release介面指標 `IUnknown` 上的釋放成員函數。

## 語法

```
void Release( );
```

## 備註

呼叫 `IUnknown::Release` 封裝的介面指標,如果此介面指標 `E_POINTER` 為 NULL,則引發錯誤。

結束微軟的

## 另請參閱

[\\_com\\_ptr\\_t類](#)

# \_com\_ptr\_t 運算子

2020/3/25 • [Edit Online](#)

如需 `_com_ptr_t` 運算子的詳細資訊，請參閱[\\_Com\\_ptr\\_t 類別](#)。

## 另請參閱

[\\_com\\_ptr\\_t 類別](#)

# \_com\_ptr\_t::operator =

2020/3/25 • • [Edit Online](#)

## Microsoft 專屬

將新值指派給現有的 `_com_ptr_t` 物件。

## 語法

```
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 );
```

## 備註

將介面指標指派至此 `_com_ptr_t` 物件。

END Microsoft 特定的

## 另請參閱

[\\_com\\_ptr\\_t 類別](#)

# \_com\_ptr\_t 關係運算子

2020/3/25 • [Edit Online](#)

Microsoft 專屬

將智慧型指標物件與另一個智慧型指標、一般介面指標或 NULL 進行比較。

## 語法

```
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 );
```

## 備註

將智慧型指標物件與另一個智慧型指標、一般介面指標或 NULL 進行比較。除了 Null 指標測試以外，這些運算子會先查詢 `IUnknown` 的指標，並比較結果。

END Microsoft 特定的

## 另請參閱

[\\_com\\_ptr\\_t 類別](#)

# `_com_ptr_t` 擷取器

2020/11/2 • [Edit Online](#)

## Microsoft 特定

擷取封裝的 COM 介面指標。

## 語法

```
operator Interface*( ) const throw( );
operator Interface&( ) const;
Interface& operator*( ) const;
Interface* operator->( ) const;
Interface** operator&( ) throw( );
operator bool( ) const throw( );
```

## 備註

- `operator Interface*` 傳回封裝的介面指標，可能是 Null。
- `operator Interface&` 傳回封裝介面指標的參考，並在指標為 Null 時發出錯誤。
- `operator*` 允許智慧型指標物件在取值時，其作用就如同實際的封裝介面。
- `operator->` 允許智慧型指標物件在取值時，其作用就如同實際的封裝介面。
- `operator&` 釋放任何封裝的介面指標，以 Null 取代它，並傳回封裝之指標的位址。這個運算子可讓您將智慧型指標 by 位址傳遞給具有 `out` 參數的函式，其會透過它傳回介面指標。
- `operator bool` 允許在條件運算式中使用智慧型指標物件。`true` 如果指標不是 Null，則這個運算子會傳回。

### NOTE

因為不 `operator bool` 是宣告為 `explicit`，所以 `_com_ptr_t` 會隱含地轉換為 `bool`，其可轉換為任何純量類型。這在您的程式碼中可能會產生非預期的結果。啟用[編譯器警告\(層級4\) C4800](#)，以防止意外使用此轉換。

## 另請參閱

[\\_com\\_ptr\\_t 類別](#)

# 關係函式樣板

2020/3/25 • [Edit Online](#)

Microsoft 專屬

語法

```

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
);

```

## 參數

*i*

原始的介面指標。

*p*

智慧型指標。

## 備註

這些函式樣板可以與比較運算子右側的智慧型指標進行比較。這些都不是 `_com_ptr_t` 的成員函式。

END Microsoft 特定的

## 另請參閱

[\\_com\\_ptr\\_t 類別](#)

# \_variant\_t 類別

2020/11/2 • [Edit Online](#)

## Microsoft 特定的

\_Variant\_t 物件會封裝 VARIANT 資料類型。類別會管理資源配置和解除配置，並 VariantInit 在適當的情況進行函式呼叫 VariantClear。

## 營造

II	II
<a href="#">_variant_t</a>	結構 _variant_t 物件。

## Operations

II	II
<a href="#">附加</a>	將 VARIANT 物件附加至 _variant_t 物件。
<a href="#">Clear</a>	清除封裝的 VARIANT 物件。
<a href="#">ChangeType</a>	將 _variant_t 物件的型別變更為指定的 VARTYPE。
<a href="#">卸離</a>	VARIANT 從這個 _variant_t 物件卸離封裝的物件。
<a href="#">SetString</a>	將字串指派給這個 _variant_t 物件。

## 運算子

II	II
<a href="#">運算子 =</a>	將新值指派給現有的 _variant_t 物件。
<a href="#">operator ==、!=</a>	比較兩個 _variant_t 物件的相等或不等。
<a href="#">擷取器</a>	從封裝的物件中取出資料 VARIANT。

結束 Microsoft 專有

## 規格需求

標頭: <comutil.h>

Lib: comsuppw.lib 或 comsuppwd.lib.lib (請參閱 [/zc: Wchar\\_t \(Wchar\\_t 是原生類型\)](#) 以取得詳細資訊)

## 另請參閱

[編譯器 COM 支援類別](#)

# \_variant\_t 成員函式

2019/12/2 • [Edit Online](#)

如需 \_variant\_t 成員函式，請參閱 [\\_variant\\_t 類別](#)。

## 另請參閱

[\\_variant\\_t 類別](#)

# \_variant\_t::variant\_t

2020/11/2 • • [Edit Online](#)

Microsoft 特定的

建構 `_variant_t` 物件。

## 語法

```
_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* pIUnknownSrc,
    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();

```

## 參數

*varSrc*

要複製到新的 `VARIANT` 物件中的 `_variant_t` 物件。

*pVarSrc*

要複製到 `VARIANT` 新物件中之物件的指標 `_variant_t` 。

*var\_t\_Src*

要複製到新的 `_variant_t` 物件中的 `_variant_t` 物件。

*fCopy*

如果為 `false`，則提供的 `VARIANT` 物件會附加至新的 `_variant_t` 物件，而不會建立新的複本 `VariantCopy` 。

*ISrc, sSrc*

要複製到新的 `_variant_t` 物件中的整數值。

*vtSrc*

`VARTYPE` 新 `_variant_t` 物件的。

*fltSrc, dblSrc*

要複製到新的 `_variant_t` 物件中的數值。

*cySrc*

要複製到新的 `CY` 物件中的 `_variant_t` 物件。

*bstrSrc*

要複製到新的 `_bstr_t` 物件中的 `_variant_t` 物件。

*strSrc, wstrSrc*

要複製到新的 `_variant_t` 物件中的字串。

*bSrc*

`bool` 要複製到新物件中的值 `_variant_t`。

*pIUnknownSrc*

要封裝至新物件之 VT\_UNKNOWN 物件的 COM 介面指標 `_variant_t`。

*pDispSrc*

要封裝至新物件之 VT\_DISPATCH 物件的 COM 介面指標 `_variant_t`。

*decSrc*

要複製到新的 `DECIMAL` 物件中的 `_variant_t` 值。

*bSrc*

要複製到新的 `BYTE` 物件中的 `_variant_t` 值。

*cSrc*

`char` 要複製到新物件中的值 `_variant_t`。

*usSrc*

`unsigned short` 要複製到新物件中的值 `_variant_t`。

*ulSrc*

`unsigned long` 要複製到新物件中的值 `_variant_t`。

*iSrc*

`int` 要複製到新物件中的值 `_variant_t`。

*uiSrc*

`unsigned int` 要複製到新物件中的值 `_variant_t`。

*i8Src*

`_int64` 要複製到新物件中的值 `_variant_t`。

*ui8Src*

要複製到新物件中的不帶正負號 `_int64` 值 `_variant_t`。

## 備註

- `_variant_t()` 建立空的 `_variant_t` 物件 `VT_EMPTY`。
- `_variant_t(variant& *varSrc*)` `_variant_t` 從物件的複本來構造物件 `VARIANT`。`variant` 類型會保留。
- `_variant_t(variant* pVarSrc***)` 會 `_variant_t` 從物件的複本來構造物件 `VARIANT`。`variant` 類型會保

留。

- `_variant_t (_variant_t& *var_t_Src*)` `_variant_t` 從另一個物件來構造物件 `_variant_t`。variant 類型會保留。
- `_variant_t (variant& varSrc, bool fCopy)` 會 `_variant_t` 從現有的物件來構造物件 `VARIANT`。如果 `fCopy` 為 `false`，則 `VARIANT` 物件會附加至新的物件，而不會建立複本。
- `*_variant_t (short***sSrc, VARTYPE vtSrc = VT_I2)` 會 `_variant_t` 從整數值中, `VT_I2` 或 `VT_BOOL` 結構來建立類型的物件 `short`。任何其他會 `VARTYPE` 導致 `E_INVALIDARG` 錯誤。
- `_variant_t (long lSrc, VARTYPE vtSrc = VT_I4)` 會 `_variant_t` 從整數值中, `VT_I4`、`VT_BOOL` 或 `VT_ERROR` 結構來建立類型的物件 `long`。任何其他會 `VARTYPE` 導致 `E_INVALIDARG` 錯誤。
- `_variant_t (float fltSrc)` 會 `_variant_t` 從數值結構 `VT_R4` 類型的物件 `float`。
- `_variant_t (double dblSrc, VARTYPE vtSrc = VT_R8)` 會 `_variant_t` 從數值中, `VT_R8` 或 `VT_DATE` 來構造類型的物件 `double`。任何其他會 `VARTYPE` 導致 `E_INVALIDARG` 錯誤。
- `_variant_t (CY& cySrc)` 會 `_variant_t` 從物件中, 建立類型 `VT_CY` 的物件 `CY`。
- `_variant_t (_bstr_t& bstrSrc)` 會 `_variant_t` 從物件中, 建立 `VT_BSTR` 類型的物件 `_bstr_t`。會配置新的 `BSTR`。
- `_variant_t (wchar_t* wstrSrc)` 會 `_variant_t` 從 Unicode 字串中, `VT_BSTR` 類型的物件進行結構。會配置新的 `BSTR`。
- `_variant_t (char* strSrc)` 會 `_variant_t` 從字串中, `VT_BSTR` 類型的物件進行結構。會配置新的 `BSTR`。
- `_variant_t (bool bSrc)` 會 `_variant_t` 從值中 `VT_BOOL` 類型的物件進行結構 `bool`。
- `_variant_t (IUnknown* pIUnknownSrc, bool fAddRef = true)` `_variant_t` 從 COM 介面指標, 建立 `VT_UNKNOWN` 類型的物件。如果 `fAddRef` 為 `true`，則 `AddRef` 會在提供的介面指標上呼叫，以符合 `Release` 將在物件終結時發生的呼叫 `_variant_t`。您必須 `Release` 在提供的介面指標上呼叫。如果 `fAddRef` 為 `false`，則此函式會取得所提供的介面指標的擁有權；不會 `Release` 在提供的介面指標上呼叫。
- `_variant_t (IDispatch)* pDispSrc, bool fAddRef = true)` `_variant_t` 從 COM 介面指標, 建立 `VT_DISPATCH` 類型的物件。如果 `fAddRef` 為 `true`，則 `AddRef` 會在提供的介面指標上呼叫，以符合 `Release` 將在物件終結時發生的呼叫 `_variant_t`。您必須 `Release` 在提供的介面指標上呼叫。如果 `fAddRef` 為 `false`，則此函式會取得所提供的介面指標的擁有權；不會 `Release` 在提供的介面指標上呼叫。
- `_variant_t (DECIMAL& decSrc)` 會 `_variant_t` 從值 `VT_DECIMAL` 類型的物件 `DECIMAL`。
- `_variant_t (BYTE bSrc)` 會 `_variant_t` 從值來構造類型的物件 `VT_UI1` `BYTE`。

結束 Microsoft 專有

## 另請參閱

[\\_variant\\_t 類別](#)

# \_variant\_t::Attach

2020/4/23 • • [Edit Online](#)

Microsoft 特定的

將 `VARIANT` 物件附加到 `_variant_t` 物件。

## 語法

```
void Attach(VARIANT& varSrc);
```

參數

*varSrc*

要 `VARIANT` 附加到此 `_variant_t` 物件的物件。

## 備註

通過封裝獲取 `VARIANT` 的擁有權。此成員函數釋放任何現有的 `VARIANT` 封裝，然後複製 `VARIANT` 提供的 `VARTYPE` 並將其設置到 `VT_EMPTY` 以確保其資源只能由 `_variant_t` 析構函數釋放。

結束微軟的

## 另請參閱

[\\_variant\\_t類](#)

# \_variant\_t::Clear

2020/4/23 • • [Edit Online](#)

Microsoft 特定的

清除封裝 VARIANT 的物件。

## 語法

```
void Clear( );
```

## 備註

調用 VariantClear 封 VARIANT 裝 的物件。

結束微軟的

## 另請參閱

[\\_variant\\_t類](#)

# \_variant\_t::ChangeType

2020/4/23 • • [Edit Online](#)

Microsoft 特定的

將 `_variant_t` 物件的類型變更為 `VARTYPE` 指示的。

## 語法

```
void ChangeType(
    VARTYPE vartype,
    const _variant_t* pSrc = NULL
);
```

參數

*vartype*

此 `VARTYPE``_variant_t` 物件的。

*pSrc*

要轉換之 `_variant_t` 物件的指標。如果此值為 `NULL`, 則轉換將就地完成。

## 備註

此成員函數將 `_variant_t` 物件轉換為 `VARTYPE` 指示的。如果 *pSrc* 為 `NULL`, 則轉換就 `_variant_t` 位, 否則 此物件將從 *pSrc* 複製, 然後轉換。

結束微軟的

## 另請參閱

[\\_variant\\_t類](#)

# \_variant\_t::Detach

2020/3/25 • • [Edit Online](#)

## Microsoft 專屬

從這個 `_variant_t` 物件卸離封裝的 `VARIANT` 物件。

## 語法

```
VARIANT Detach( );
```

## 傳回值

封裝的 `VARIANT`。

## 備註

解壓縮並傳回封裝的 `VARIANT`，然後清除此 `_variant_t` 物件，而不會終結它。此成員函式會移除封裝中的 `VARIANT`，並將這個 `_variant_t` 物件的 `VARTYPE` 設定為 `VT_EMPTY`。您必須呼叫 [VariantClear](#) 函式，以釋放傳回的 `VARIANT`。

END Microsoft 特定的

## 另請參閱

[\\_variant\\_t 類別](#)

# \_variant\_t::SetString

2020/4/22 • • [Edit Online](#)

Microsoft 特定的

將字串指派給這個 `_variant_t` 物件。

## 語法

```
void SetString(const char* pSrc);
```

參數

*pSrc*

字元字串的指標。

## 備註

將 ANSI 字串轉換為 Unicode `BSTR` 字串，並將它指派給這個 `_variant_t` 物件。

結束微軟的

## 另請參閱

[\\_variant\\_t類](#)

# \_variant\_t 運算子

2020/3/25 • [Edit Online](#)

如需 `_variant_t` 運算子的詳細資訊，請參閱 [\\_variant\\_t 類別](#)。

## 另請參閱

[\\_variant\\_t 類別](#)

# \_variant\_t::operator =

2020/11/2 • • [Edit Online](#)

Microsoft 特定的

## 語法

```
_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 bSrc  

);

_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  

);

```

## 備註

將新值指派給 `_variant_t` 物件的運算子：

- `operator = ( varSrc)` 將現有的指派 `VARIANT` 級 `_variant_t` 物件。
- `operator = ( pVarSrc)` 將現有的指派 `VARIANT` 級 `_variant_t` 物件。
- `operator = ( var_t_Src)` 將現有的 `_variant_t` 物件指派給 `_variant_t` 物件。
- `operator = ( sSrc)` 將 `short` 整數值指派給 `_variant_t` 物件。
- `operator = ( lSrc )` 會將 `long` 整數值指派給 `_variant_t` 物件。
- `operator = ( fltSrc )` 將 `float` 數值指派給 `_variant_t` 物件。
- `operator = ( dblSrc )` 將 `double` 數值指派給 `_variant_t` 物件。
- `operator = ( cySrc )` 將 `cy` 物件指派給 `_variant_t` 物件。
- `operator = ( bstrSrc )` 將 `BSTR` 物件指派給 `_variant_t` 物件。
- `operator = ( wstrSrc )` 將 Unicode 字串指派給 `_variant_t` 物件。
- `operator = ( strSrc )` 會將多位元組字元串指派給 `_variant_t` 物件。
- `operator = ( bSrc )` 會將 `bool` 值指派給 `_variant_t` 物件。

- `operator = ( pDispSrc )` 將 `VT_DISPATCH` 物件指派給 `_variant_t` 物件。
- `operator = ( pIUnknownSrc )` 將 `VT_UNKNOWN` 物件指派給 `_variant_t` 物件。
- `operator = ( decSrc )` 將 `DECIMAL` 值指派給 `_variant_t` 物件。
- `operator = ( bSrc )` 會將 `BYTE` 值指派給 `_variant_t` 物件。

結束 Microsoft 專有

## 另請參閱

[\\_variant\\_t 類別](#)

# \_variant\_t 關係運算子

2020/11/2 • [Edit Online](#)

Microsoft 特定的

比較兩個 `_variant_t` 物件是否相等或不等。

## 語法

```
bool operator==(  
    const VARIANT& varSrc) const;  
bool operator==(  
    const VARIANT* pSrc) const;  
bool operator!=(  
    const VARIANT& varSrc) const;  
bool operator!=(  
    const VARIANT* pSrc) const;
```

參數

*varSrc*

要 `VARIANT` 要與物件比較的 `_variant_t`。

*pSrc*

要 `VARIANT` 與物件比較之的指標 `_variant_t`。

## 傳回值

`true` 如果比較保留，則傳回，`false` 否則傳回。

## 備註

比較 `_variant_t` 物件與 `VARIANT`，測試是否相等或不等。

結束 Microsoft 專有

## 另請參閱

[\\_variant\\_t 類別](#)

# \_variant\_t 擷取器

2020/11/2 • • [Edit Online](#)

Microsoft 特定的

從封裝的物件解壓縮資料 `VARIANT`。

## 語法

```
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;
```

## 備註

從封裝的中解壓縮原始資料 `VARIANT`。如果不 `VARIANT` 是正確的型別，`VariantChangeType` 就會用來嘗試轉換，並在失敗時產生錯誤：

- `operator short()` 將 `short` 整數值解壓縮。
- 運算子 `long()` 將 `long` 整數值解壓縮。
- 運算子 `float()` 解壓縮 `float` 數值。
- 運算子 `double()` 將 `double` 整數值解壓縮。
- 運算子 `CY()` 解壓縮 `CY` 物件。
- `operator bool()` 解壓縮 `bool` 值。
- 運算子 `DECIMAL()` 解壓縮 `DECIMAL` 值。
- `OPERATOR BYTE()` 解壓縮 `BYTE` 值。
- 運算子 `_bstr_t()` 解壓縮字串，其封裝在 `_bstr_t` 物件中。
- 運算子 `IDispatch *` () 會從封裝的中抽取一個分配介面指標 `VARIANT`。`AddRef` 會在產生的指標上呼叫，因此您可以由您呼叫 `Release` 來釋放它。
- 運算子 `IUnknown *` () 會從封裝的中，解壓縮 COM 介面指標 `VARIANT`。`AddRef` 會在產生的指標上呼叫，

因此您可以由您呼叫 `Release` 來釋放它。

結束 Microsoft 專有

另請參閱

[\\_variant\\_t 類別](#)

# Microsoft 擴充功能

2020/11/2 • [Edit Online](#)

```
asm-statement :  
* __asm *** assembly-instruction ; opt  
* __asm { *** assembly-instruction-list } ; opt  
  
assembly-instruction-list :  
assembly-instruction *** ; * opt  
assembly-instruction *** ; * assembly-instruction-list ; opt  
  
ms-modifier-list :  
ms-modifier ** ms-modifier-list opt  
  
ms-modifier :  
__cdecl  
__fastcall  
__stdcall  
__syscall (保留供未來的實施之用)  
__oldcall (保留供未來的實施之用)  
__unaligned (保留供未來的實施之用)  
based-modifier  
  
based-modifier :  
__based ( based-type )  
  
based-type :  
name
```

# 非標準行為

2020/11/2 • [Edit Online](#)

下列各節列出 Microsoft 的 C++ 執行不符合 C++ 標準的部分位置。下列章節編號是指 C++ 11 標準 (ISO/IEC 14882:2011(E)) 中的章節編號。

編譯器限制會提供與 C++ 標準中所定義不同的編譯器限制清單。

## Covariant 傳回類型

當虛擬函式具有可變數目的引數時，不支援虛擬基底類別做為 Covariant 傳回型別。不符合 C++ ISO 規格第 7 段的第 10.3 節。下列範例不會進行編譯，因而產生編譯器錯誤 [C2688](#)

```
// CovariantReturn.cpp
class A
{
    virtual A* f(int c, ...); // remove ...
};

class B : virtual A
{
    B* f(int c, ...); // C2688 remove ...
};
```

## 樣板中的繫結非相依名稱

Microsoft C++ 編譯器目前不支援在一開始剖析範本時系結非相依名稱。不符合 C++ ISO 規格的第 14.6.3 節。可能會在樣板出現後 (但在樣板具現化之前) 造成宣告多載。

```
#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)
```

## 函式例外狀況規範

會剖析但不使用 `throw()` 以外的函式例外狀況規範。不符合 ISO C++ 規格的第 15.4 節。例如：

```
void f() throw(int); // parsed but not used
void g() throw();    // parsed and used
```

如需例外狀況規格的詳細資訊，請參閱[例外狀況規格](#)。

## char\_traits::eof()

`Char_traits::eof` 的 C++ 標準狀態不能對應至有效的 `char_type` 值。Microsoft C++ 編譯器會對類型強制執行這個條件約束 `char`，而不是針對類型 `wchar_t`。這不符合 C++ ISO 規格第 12.1.1 節表 62 的要求。以下範例即為示範。

```
#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;
}
```

## 物件的儲存位置

C++ 標準 (第 6 段第 1.8 節) 要求完整的 C++ 物件必須具有唯一的儲存位置。不過，在 Microsoft C++ 中，有些情況下，沒有資料成員的類型會在物件的存留期內與其他類型共用儲存位置。

# 編譯器限制

2020/11/2 • [Edit Online](#)

C++ 標準會建議各種語言的建構限制。以下是 Microsoft C++ 編譯器不會執行建議限制的案例清單。第一個數位是 ISO C++ 11 標準(INCITS/ISO/IEC 14882-2011 [2012], 附錄 B)中所建立的限制，而第二個數字則是 Microsoft C++ 編譯器所執行的限制：

- 複合陳述式、反復專案控制結構和選取控制結構的嵌套層級-C++ 標準:256、Microsoft C++ 編譯器:取決於嵌套的語句組合，但通常是在100與110之間。
- 一個巨集定義中的參數-C++ 標準:256、Microsoft C++ 編譯器:127。
- 一個宏調用中的引數-C++ 標準:256、Microsoft C++ 編譯器:127。
- 字元字串常值或寬字元串常值中的字元(在串連之後)-C++ 標準:65536、Microsoft C++ 編譯器:65535單一位元組字元(包括 Null 結束字元)和32767雙位元組字元，包括 Null 結束字元。
- 單一 C++ 標準中的嵌套類別、結構或等位定義層級 `struct-declaration-list`:256、Microsoft C++ 編譯器:16。
- 在函式定義中的成員初始化運算式-C++ 標準:6144、Microsoft C++ 編譯器:至少6144。
- 一個識別碼的範圍限定-C++ 標準:256、Microsoft C++ 編譯器:127。
- 嵌套 `extern` 規格-C++ 標準:1024、Microsoft C++ 編譯器:9(`extern` 如果您在全域範圍中計算隱含規格，則不會計算全域範圍中的隱含規格，或10)`extern`。
- 範本宣告中的樣板引數-C++ 標準:1024、Microsoft C++ 編譯器:2046。

## 另請參閱

[非標準行為](#)

# C/C++ 前置處理器參考

2020/11/2 • [Edit Online](#)

*C/c++ 預處理器參考*會說明在 Microsoft C/c++ 中執行預處理器的程式。前置處理器會先對 C 和 C++ 檔案執行初步作業，再將檔案傳遞至編譯器。您可以使用前置處理器，有條件地編譯程式碼、插入檔案、指定編譯時間錯誤訊息，以及將電腦特定規則加入至程式碼區段。

在 Visual Studio 2019 中，[/zc:預處理器](#) 編譯器選項提供完全一致的 C11 和 C17 預處理器。這是使用編譯器旗標或時的預設 `/std:c11` 值 `/std:c17`。

## 本節內容

### 預處理

提供傳統和新的符合預處理器的總覽。

### 預處理器指示詞

描述指示詞，通常用來使原始程式易於變更，以及在不同的執行環境中易於編譯。

### 預處理器運算子

討論 `#define` 指示詞的內容中所使用的四個前置處理器特定運算子。

### 預先定義的宏

討論 C 和 C++ 標準和 Microsoft C++ 所指定的預先定義宏。

### Pragma

討論 pragma，它讓每個編譯器可以提供電腦和作業系統專屬功能，同時還能保留與 C 及 C++ 語言的整體相容性。

## 相關章節

### C++ 語言參考

為 C++ 語言的 Microsoft 實作提供參考資料。

### C 語言參考

為 C 語言的 Microsoft 實作提供參考資料。

### C/c++ 組建參考

提供討論編譯器和連結器選項的主題連結。

### Visual Studio 專案-C++

說明 Visual Studio 使用者介面，可讓您指定專案系統將搜尋的目錄，以尋找您的 C++ 專案的檔案。

# C++ 標準程式庫參考

2019/12/2 • [Edit Online](#)

C++ 程式可以透過 C++ 標準程式庫的這個合格實作呼叫大量函式。這些函式執行基本服務 (例如輸入和輸出)，並且有效率地實作常用作業。

如需 Visual C++ 執行階段程式庫的詳細資訊，請參閱 [CRT 程式庫功能](#)。

## 本節內容

### [C++ 標準程式庫概觀](#)

提供 C++ 標準程式庫之 Microsoft 實作的概觀。

### [iostream 程式設計](#)

提供 iostream 程式設計的概觀。

### [標頭檔參考](#)

提供 C++ 標準程式庫標頭檔之參考主題的說明連結與程式碼範例。