# ACPS C++ 教學講義

C++ 讀作「C 加加」，是「C Plus Plus」的簡稱。顧名思義，C++ 是在 C 語言的基礎上增加新特性，所以叫「C Plus Plus」，就像 iPhone 6S 和 iPhone 6 的關係。

從語法上看，C 語言是 C++的一部分，C 語言程式碼幾乎不用修改就能夠以 C++ 的方式編譯，這給很多初學者帶來了不小的困惑，學習 C++之前到底要不要先學習 C 語言呢？

我對這個問題保持中立，不過可以明確地說：學了 C 語言就相當於學了 C++的一半，從 C 語言轉向 C++時，不需要再從頭開始，接著 C 語言往下學就可以，但還是要花費更多的時間，因為你還要學習那些是 C++ 才有的指令或用法，也要背下不同的標頭引入檔。

學習程式是一個循序漸進的過程，不要期望一口吃個胖子。學習 C++，一來是學習它的語法，二來是同時培養寫程式的興趣，弄清程式語言的原理。

每個初學者都經歷過這樣的窘境：已經學習了語法，明白了程式語言都有什麼，也按照範例敲了不少程式碼，但是遇到實際問題就掛了，沒有思路，不知道從何下手。說白了就是只會學不會用。

究其原因，就是實踐少，沒有培養起撰寫程式的思維！學習知識容易，運用知識才是重點！

# C++ 基本語法

讓我們看一段簡單的程式碼，可以輸出單詞 Hello World。

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8      cout << "Hello World!" << endl;
9
10     cout<<endl;system("pause");
11     return 0;
12 }
13
```



接下來我們講解一下上面這段程式：

C++ 語言定義了一些標頭文件，這些標頭文件包含了程式中必需的或有用的訊息。上面這段程式中，包含了標頭文件

```
<stdio.h>
<stdlib.h>
<iostream>
```

using namespace std; 告訴編譯器使用 std 命名空間。

命名空間是 C++ 中一個相對新的概念。

int main() 是主函數，程式從這裡開始執行。

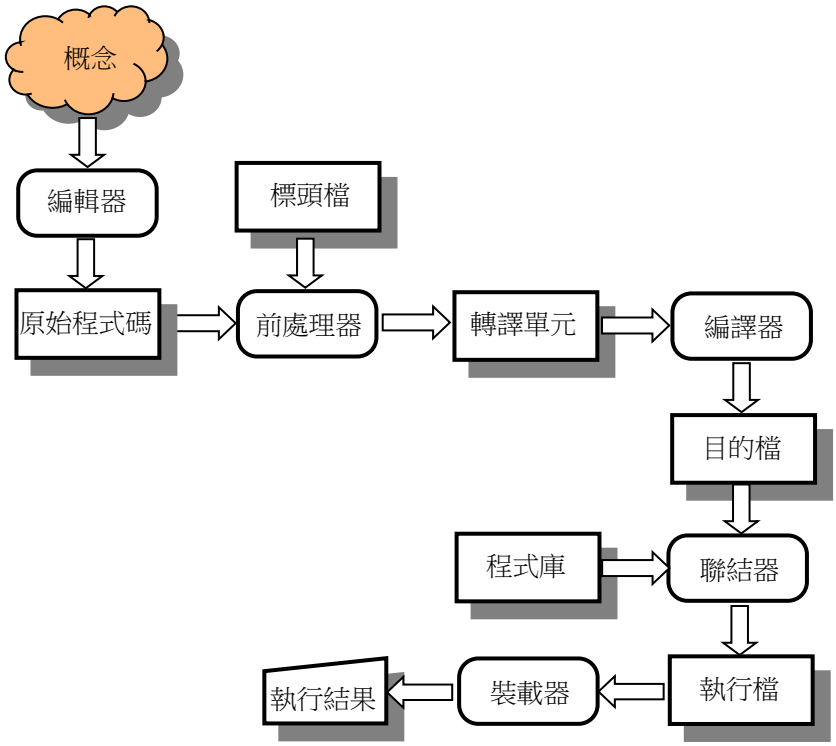
cout << "Hello World" << endl;

會在螢幕上顯示文字 "Hello World"，endl 是換行符(跳列)。

system("pause"); 是暫停。

return 0; 終止 main( )函數，並返回值 0。

# 基本程式開發步驟示意圖




# C++ 識別符號

C++ 識別符號是用來標識變數、函數、類別、區塊，或任何其他用戶自定義項目的名稱。一個識別符號以字母 A-Z 或 a-z 或下劃線 _ 開始，後跟零個或多個字母、下劃線和數字（0-9）。

C++ 識別符號內不允許出現標點字元，比如 @、$ 和 %。

C++ 是區分大小寫的程式語言。因此，在 C++ 中，**Man1** 和 **man1** 是兩個不同的識別符號。

下面列出幾個有效的識別符號：

**zara**
**retVal**
**abc**
**a_123**
**_temp**
**myname50**
**j**
**a23b9**

# C++ 關鍵字

下表列出了 C++ 中的保留字。這些保留字不能作為常數名、變數名或其他識別符號名稱。

| | | | |
|---|---|---|---|
| asm | else | new | this |
| auto | enum | operator | throw |
| bool | explicit | private | true |
| break | export | protected | try |
| case | extern | public | typedef |
| catch | false | register | typeid |
| char | float | reinterpret_cast | typename |
| class | for | return | union |
| const | friend | short | unsigned |
| const_cast | goto | signed | using |
| continue | if | sizeof | virtual |
| default | inline | static | void |
| delete | int | static_cast | volatile |
| do | long | struct | wchar_t |
| double | mutable | switch | while |
| dynamic_cast | namespace | template | |

# C++ 註釋

程式的註釋是解釋性語句，您可以在 C++ 程式碼中包含註釋，這將提高程式碼的可讀性。所有的程式語言都允許某種形式的註釋。

C++ 支持單行註釋和多行註釋。註釋中的所有字元會被 C++ 編譯器忽略。

C++ 註釋以 /* 開始，以 */ 終止。例如：

```
/* 這是註釋 */
/* C++ 註釋也可以
 * 跨行
 */
```

註釋也能以 // 開始，直到行末為止。例如：

```
#include
using namespace std;

main()
{
   cout << "Hello World"; // 輸出 Hello World
   return 0;
}
```

當上面的程式碼被編譯時，編譯器會忽略 // 輸出 Hello World

# C++ 資料類型

使用程式語言進行程式設計時，需要用到各種變數來儲存各種訊息。變數保留的是它所儲存的值的記憶體位置。這意味著，當您建立一個變數時，就會在記憶體中保留一些空間。

您可能需要儲存各種資料類型（比如字元型、寬字元型、整數型、浮點型、雙浮點型、布林型等）的訊息，操作系統會根據變數的資料類型，來分配記憶體和決定在保留記憶體中儲存什麼。

# 基本的內置類型

C++ 為程式設計人員提供了種類豐富的內置資料類型和用戶自定義的資料類型。下表列出了七種基本的 C++ 資料類型：

| 類型 | 關鍵字 | 描述 |
|---|---|---|
| 布林型 | bool | 儲存值 true 或 false。 |
| 字元型 | char | 通常是一個 8 bit（1 Byte）。是一個整數類型。 |
| 整數型 | int | 整數的最自然的大小。 |
| 浮點型 | float | 單精度浮點值。 |
| 雙浮點型 | double | 雙精度浮點值。 |
| 無類型 | void | 表示類型的缺失。 |
| 寬字元型 | wchar_t | 寬字元類型。 |

一些基本類型可以使用一個或多個類型修飾符號進行修飾：

- Signed 帶+-符號
- Unsigned 不帶+-符號
- Short 短
- Long 長

下表顯示了各種變數類型在記憶體中儲存值時需要佔用的記憶體，以及該類型的變數所能儲存的最大值和最小值。

注意：不同系統會有所差異。

| 類型 | 位元祖 | 範圍 |
|---|---|---|
| char | 1 個 Byte | -128 到 127 或者 0 到 255 |
| unsigned char | 1 個 Byte | 0 到 255 |
| signed char | 1 個 Byte | -128 到 127 |
| int | 4 個 Byte | -2147483648 到 2147483647 |
| unsigned int | 4 個 Byte | 0 到 4294967295 |
| signed int | 4 個 Byte | -2147483648 到 2147483647 |
| short int | 2 個 Byte | -32768 到 32767 |
| unsigned short int | 2 個 Byte | 0 到 65,535 |
| signed short int | 2 個 Byte | -32768 到 32767 |

| long int | 8 個 Byte | -9,223,372,036,854,775,808 到 9,223,372,036,854,775,807 |
|---|---|---|
| signed long int | 8 個 Byte | -9,223,372,036,854,775,808 到 9,223,372,036,854,775,807 |
| unsigned long int | 8 個 Byte | 0 to 18,446,744,073,709,551,615 |
| float | 4 個 Byte | +/- 3.4e +/- 38 (~7 個數字) |
| double | 8 個 Byte | +/- 1.7e +/- 308 (~15 個數字) |
| long double | 16 個 Byte | +/- 1.7e +/- 308 (~15 個數字) |
| wchar_t | 2 或 4 個 Byte | 1 個寬字元 |

從上表可得知，變數的大小會根據編譯器和所使用的電腦而有所不同。

下面實例會輸出您電腦上各種資料類型的大小。

```
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
using namespace std;

int main()
{
    cout << "Size of char : " << sizeof(char) << endl;
    cout << "Size of int : " << sizeof(int) << endl;
    cout << "Size of short int : " << sizeof(short int) << endl;
    cout << "Size of long int : " << sizeof(long int) << endl;
    cout << "Size of float : " << sizeof(float) << endl;
    cout << "Size of double : " << sizeof(double) << endl;
    cout << "Size of wchar_t : " << sizeof(wchar_t) << endl;

    cout<<endl;
    system("pause");
    return 0;
}
```

本實例使用 **sizeof()** 函數來獲取各種資料類型的大小。

當上面的程式碼被編譯和執行時，它會產生以下的結果，結果會根據所使用的計算機而有所不同：

```
Size of char : 1
Size of int : 4
Size of short int : 2
Size of long int : 4
Size of float : 4
Size of double : 8
Size of wchar_t : 4
```

# C++ 中的變數宣告

變數宣告向編譯器保證變數以給定的類型和名稱存在，這樣編譯器在不需要知道變數完整細節的情況下也能繼續進一步的編譯。

## 實例

```cpp
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
using namespace std;

int main()
{
    // 變數定義
    int a, b;
    int c;
    float f;

    // 實際初始化
    a = 10;
    b = 20;
    c = a + b;

    cout << c << endl ;
    f = 70.0/3.0;
    cout << f << endl ;

    cout<<endl;
    system("pause");
    return 0;
}
```



當上面的程式碼被編譯和執行時，它會產生下列結果：

```
30
23.3333
```

## C++ 中的左值（Lvalues）和右值（Rvalues）

**左值（lvalue）**：指向記憶體位置的表達式被稱為左值表達式。

**右值（rvalue）**：指的是儲存的數值。

變數是左值，因此可以出現在賦值號的左邊。數值型的字面值是右值。

下面是一個有效的語句：

```
int g = 20;
```

但是下面這個就不是一個有效的語句，會生成編譯時錯誤：

```
10 = 20;
```

# C++ 變數作用範圍

作用範圍是程式的一個區域，一般來說有三個地方可以宣告變數：

- 在函數或一個程式碼塊內部宣告的變數，稱為區域變數。
- 在函數參數的定義中宣告的變數，稱為形式參數。
- 在所有函數外部宣告的變數，稱為全域變數。

## 區域變數

在函數或一個程式碼塊內部宣告的變數，稱為區域變數。它們只能被函數內部或者程式碼塊內部的語句使用。

## 全域變數

在所有函數外部定義的變數（通常是在程式的開頭），稱為全域變數。全域變數的值在程式的整個生命週期內都是有效的。

全域變數可以被任何函數引用。也就是說，全域變數一旦宣告，在整個程式中都是可用的。下面的實例使用了全域變數和區域變數：

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

// 全域變數宣告
int g;

int main ()
{
  // 區域變數宣告
  int a, b;

  // 實際初始化
  a = 10;
  b = 20;
  g = a + b;

  cout << g;    return 0;
 }
```

在程式中，區域變數和全域變數的名稱可以相同，但是在函數內，區域變數的值會覆蓋全域變數的值。下面是一個實例：

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

// 全域變數宣告
int g = 20;

int main ()
{
  // 區域變數宣告
  int g = 10;

  cout << g;    return 0;
}
```

當上面的程式碼被編譯和執行時，它會產生下列結果：

```
10
```

# 初始化區域變數和全域變數

當區域變數被定義時，系統不會對其初始化，您必須自行對其初始化。定義全域變數時，系統會自動初始化為下列值：

| 資料類型 | 初始化默認值 |
|---|---|
| int | 0 |
| char | '\0' |
| float | 0 |
| double | 0 |
| pointer | NULL |

正確地初始化變數是一個良好的程式設計習慣，否則有時候程式可能會產生意想不到的結果。

# C++ 常數

常數是固定值，在程式執行期間不會改變。

常數就像是常規的變數，只不過常數的值在定義後不能進行修改。

## 整數常數

整數常數可以是十進制、八進制或十六進制的常數。前綴指定基數：0x 或 0X 表示十六進制，0 表示八進制，不帶前綴則默認表示十進制。

整數常數也可以帶一個後綴，後綴是 U 和 L 的組合，U 表示無符號整數（unsigned），L 表示長整數（long）。後綴可以是大寫，也可以是小寫，U 和 L 的順序任意。

下面列舉幾個整數常數的實例：

```
212         // 合法的
215u        // 合法的
0xFeeL      // 合法的
078         // 非法的：8 不是八進制的數字
```

```
032UU        // 非法的：不能重複後綴
```

以下是各種類型的整數常數的實例：

```
85          // 十進制
0213        // 八進制
0x4b        // 十六進制
30          // 整數
30u         // 無符號整數
30l         // 長整數
30ul        // 無符號長整數
```

## 浮點常數

浮點常數由整數部分、小數點、小數部分和指數部分組成。您可以使用小數形式或者指數形式來表示浮點常數。

下面列舉幾個浮點常數的實例：

```
3.14159       // 合法的
314159E-5L    // 合法的
510E          // 非法的：不完整的指數
210f          // 非法的：沒有小數或指數
.e55          // 非法的：缺少整數或分數
```

## 布林常數

布林常數共有兩個，它們都是標準的 C++ 關鍵字：

- `true` 值代表真。
- `false` 值代表假。

## 字元常數

字元常數是括在單引號中。如果常數以 L（僅當大寫時）開頭，則表示它

是一個寬字元常數（例如 L'x'），此時它必須儲存在 **wchar_t** 類型的變數中。否則，它就是一個窄字元常數（例如 'x'），此時它可以儲存在 **char** 類型的簡單變數中。

在 C++ 中，有一些特定的字元，當它們前面有反斜線時，它們就具有特殊的含義，被用來表示如換行符號（\n）或定位符號（\t）等。下表列出了一些這樣的跳脫字元：

| 跳脫序列 | 含義 |
|---|---|
| \\ | \ 字元 |
| \' | ' 字元 |
| \" | " 字元 |
| \? | ? 字元 |
| \a | 警報鈴聲 |
| \b | 倒退鍵 |
| \f | 換頁符號 |
| \n | 換行符號 |
| \r | Enter |
| \t | 水平定位符號 |
| \v | 垂直定位符號 |
| \ooo | 一到三位的八進制數 |
| \xhh . . . | 一個或多個數字的十六進制數 |

下面的實例顯示了一些跳脫序列字元：

```cpp
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
using namespace std;

int main()
{
    cout << "Helloo\b123\tWorld\n\nC++\n";

    cout<<endl;
    system("pause");
    return 0;
}

```



當上面的程式碼被編譯和執行時，它會產生下列結果：

```
Hello123        World

C++
```

## 字串常數

字串值或常數是括在雙引號 "" 中的。

包含普通的字元、跳脫序列的字元。

```
"hello,dear\n"
```

## 定義常數

在 C++ 中，有兩種簡單的定義常數的方式：

- 使用 **#define** 預處理器。
- 使用 **const** 關鍵字。

# #define 預處理器

下面是使用 #define 預處理器定義常數的形式：

```
#define identifier value
```

具體請看下面的實例：

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <iostream>
4  using namespace std;
5
6  #define LENGTH 10
7  #define WIDTH  5
8  #define NEWLINE '\n'
9
10 int main()
11 {
12     int area;
13     area = LENGTH * WIDTH;
14     cout << area;
15     cout << NEWLINE;
16
17     cout<<endl;
18     system("pause");
19     return 0;
20 }
21
```



當上面的程式碼被編譯和執行時，它會產生下列結果：

```
50
```

# const 關鍵字

您可以使用 const 前綴宣告指定類型的常數，如下所示：

```
const type variable = value;
```

具體請看下面的實例：

```
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
using namespace std;

int main()
{
    const int  LENGTH = 10;
    const int  WIDTH  = 5;
    const char NEWLINE = '\n';
    int area;

    area = LENGTH * WIDTH;
    cout << area;
    cout << NEWLINE;

    cout<<endl;
    system("pause");
    return 0;
}
```



當上面的程式碼被編譯和執行時，它會產生下列結果：

```
50
```

**請注意，把常數定義為大寫字母形式，是一個很好的程式設計方式。**

# C++ 儲存類型

儲存類型定義 C++ 程式中變數/函數的範圍（可見性）和生命週期。這些說明符放置在它們所修飾的類型之前。下面列出 C++ 程式中可用的儲存類型：

- auto
- register
- static
- extern
- mutable

## auto 儲存類型

**auto** 儲存類型是所有區域變數默認的儲存類型。

```
int mount;
auto int month;
```

auto 只能用在函數內，即 auto 只能修飾區域變數。

## register 儲存類型

**register** 儲存類型用於定義儲存在暫存器中而不是 RAM 中的區域變數。這意味著變數的最大尺寸等於暫存器的大小。

```
register int  miles;
```

暫存器只用於需要快速引用的變數，比如計數器。

## static 儲存類型

**static** 儲存類型指示編譯器在程式的生命週期內保持區域變數的存在，而不需要在每次它進入和離開作用範圍時進行建立和銷毀。

static 修飾符號也可以應用於全域變數。當 static 修飾全域變數時，

會使變數的作用範圍限制在宣告它的文件內。

```
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
using namespace std;

static int count = 10;  // 全域變數
void func( void )       // 函數
{
    static int i = 5;   // 局部靜態變數
    i++;
    cout << " i :" << i ;
    cout << "\tcount :" << count << endl;
}

int main()
{
    while(count--)
        func();

    cout<<endl;
    system("pause");
    return 0;
}
```



當上面的程式碼被編譯和執行時，它會產生下列結果：

```
i :6    count :9
i :7    count :8
i :8    count :7
i :9    count :6
i :10   count :5
i :11   count :4
i :12   count :3
i :13   count :2
i :14   count :1
i :15   count :0

請按任意鍵繼續 . . .
```

## extern 儲存類型

**extern** 修飾符號通常用於當有兩個或多個文件共享相同的全域變數或函數的時候，可以這麼理解，*extern* 是用來在另一個文件中宣告一個全域變數或函數。如下所示：

```
第一個文件：main.cpp
#include <iostream>

int count ;
extern void write_extern();

main()
{
   count = 5;
   write_extern();
}
第二個文件：support.cpp
#include <iostream>

extern int count;

void write_extern(void)
{
   std::cout << "Count is " << count << std::endl;
}
```

在這裡，第二個文件中的 *extern* 關鍵字用於宣告已經在第一個文件 main.cpp 中定義的 count。

## mutable 儲存類型

**mutable** 說明符號僅適用於類的對象，它允許對象的成員替代常數。也就是說，mutable 成員可以通過 const 成員函數修改。

# C++ 運算符號

運算符號是一種告訴編譯器執行特定的數學或邏輯操作的符號。C++ 內置了豐富的運算符號，並提供了以下類型的運算符號：

- 算術運算符號
- 關係運算符號
- 邏輯運算符號
- 位運算符號
- 賦值運算符號
- 雜項運算符號

## 算術運算符號

下表顯示了 C++ 支持的所有算術運算符號。

假設變數 A 的值為 10，變數 B 的值為 20，則：

| 運算符號 | 描述 | 實例 |
|---|---|---|
| + | 把兩個操作數相加 | A + B 將得到 30 |
| - | 從第一個操作數中減去第二個操作數 | A - B 將得到 -10 |
| * | 把兩個操作數相乘 | A * B 將得到 200 |
| / | 分子除以分母 | B / A 將得到 2 |
| % | 取模運算符號，整除後的餘數 | B % A 將得到 0 |
| ++ | 自增運算符號，整數值增加 1 | A++ 將得到 11 |
| -- | 自減運算符號，整數值減少 1 | A-- 將得到 9 |

## 關係運算符號

下表顯示了 C++ 支持的所有關係運算符號。

| 運算符號 | 描述 | 實例 |
|---|---|---|
| == | 檢查兩個操作數的值是否相等，如果相等則條件為真。 | (A == B) 不為真。 |
| != | 檢查兩個操作數的值是否相等，如果不相等則條件為真。 | (A != B) 為真。 |
| > | 檢查左操作數的值是否大於右操作數的值，如果是則條件為真。 | (A > B) 不為真。 |
| < | 檢查左操作數的值是否小於右操作數的值，如果是則條件為真。 | (A < B) 為真。 |
| >= | 檢查左操作數的值是否大於或等於右操作數的值，如果是則條件為真。 | (A >= B) 不為真。 |
| <= | 檢查左操作數的值是否小於或等於右操作數的值，如果是則條件為真。 | (A <= B) 為真。 |

## 邏輯運算符號

下表顯示了 C++ 支持的所有關係邏輯運算符號。

假設變數 A 的值為 1，變數 B 的值為 0，則：

| 運算符號 | 描述 | 實例 |
|---|---|---|
| && | 稱為邏輯與運算符號。如果兩個操作數都非零，則條件為真。 | (A && B) 為假。 |
| \|\| | 稱為邏輯或運算符號。如果兩個操作數中有任意一個非零，則條件為真。 | (A \|\| B) 為真。 |
| ! | 稱為邏輯非運算符號。用來逆轉操作數的邏輯狀態。如果條件為真則邏輯非運算符號將使其為假。 | !(A && B) 為真。 |

## 位運算符號

位運算符號作用於位，並逐位執行操作。&、 | 和 ^ 的真值表如下所示：

| A | B | A & B | A \| B | A ^ B |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 1 |

假設如果 A = 60，且 B = 13，現在以二進制格式表示，它們如下所示：

```
A = 0011 1100
B = 0000 1101
-----------------
A&B = 0000 1100
A|B = 0011 1101
A^B = 0011 0001
```

## 賦值運算符號

下表列出了 C++ 支持的賦值運算符號：

| 運算符號 | 描述 | 實例 |
|---|---|---|
| = | 簡單的賦值運算符號，把右邊操作數的值賦給左邊操作數 | C = A + B 將把 A + B 的值賦給 C |
| += | 加且賦值運算符號，把右邊操作數加上左邊操作數的結果賦值給左邊操作數 | C += A 相當於 C = C + A |
| -= | 減且賦值運算符號，把左邊操作數減去右邊操作數的結果賦值給左邊操作數 | C -= A 相當於 C = C - A |
| *= | 乘且賦值運算符號，把右邊操作數乘以左邊操作數的結果賦值給左邊操作數 | C *= A 相當於 C = C * A |
| /= | 除且賦值運算符號，把左邊操作數除以右邊操作數的結果賦值給左邊操作數 | C /= A 相當於 C = C / A |
| %= | 求模且賦值運算符號，求兩個操作數的模賦值給左邊操作數 | C %= A 相當於 C = C % A |
| <<= | 左移且賦值運算符號 | C <<= 2 等同於 C = C << 2 |
| >>= | 右移且賦值運算符號 | C >>= 2 等同於 C = C >> 2 |
| &= | 按位與且賦值運算符號 | C &= 2 等同於 C = C & 2 |
| ^= | 按位異或且賦值運算符號 | C ^= 2 等同於 C = C ^ 2 |
| \|= | 按位或且賦值運算符號 | C \|= 2 等同於 C = C \| 2 |

# 雜項運算符號

下表列出了 C++ 支持的其他一些重要的運算符號。

| 運算符號 | 描述 |
|---|---|
| sizeof | sizeof 運算符號。返回變數的大小。。 |
| Condition ? X : Y | 條件運算符號。如果 Condition 為真 ? 則值為 X : 否則值為 Y。 |
| , | 逗號運算符號會順序執行運算。 |
| .（點）和 ->（箭頭） | 成員運算符號用於引用類、結構和共用體的成員。 |
| & | 指標運算符號 & 返回變數的地址。例如 &a |
| * | 指標運算符號 * 指向一個變數。例如，*var |

# C++ 中的運算符號優先等級

運算符號的優先等級確定表達式中項的組合。這會影響到一個表達式如何計算。某些運算符號比其他運算符號有更高的優先等級，例如，乘除運算符號具有比加減運算符號更高的優先等級。

例如 x = 7 + 3 * 2，在這裡，x 被賦值為 13，而不是 20，因為運算符號 * 具有比 + 更高的優先等級，所以首先計算乘法 3*2，然後再加上 7。

下表將按運算符號優先等級從高到低列出各個運算符號，具有較高優先等級的運算符號出現在表格的上面，具有較低優先等級的運算符號出現在表格的下面。在表達式中，較高優先等級的運算符號會優先被計算。

| 類別 | 運算符號 | 結合性 |
|---|---|---|
| 後綴 | () [] -> . ++ - - | 從左到右 |
| 一元 | + - ! ~ ++ - - (type)* & sizeof | 從右到左 |
| 乘除 | * / % | 從左到右 |
| 加減 | + - | 從左到右 |
| 移位 | << >> | 從左到右 |

| | | |
|---|---|---|
| 關係 | < <= > >= | 從左到右 |
| 相等 | == != | 從左到右 |
| 位與 AND | & | 從左到右 |
| 位異或 XOR | ^ | 從左到右 |
| 位或 OR | \| | 從左到右 |
| 邏輯與 AND | && | 從左到右 |
| 邏輯或 OR | \|\| | 從左到右 |
| 條件 | ?: | 從右到左 |
| 賦值 | = += -= *= /= %=>>= <<= &= ^= \|= | 從右到左 |
| 逗號 | , | 從左到右 |

## 實例

我有一矩形場地，長為 100，寬為 80，但中央區域會興建圓形噴水池，噴水池直徑為 50，請問實際可用面積為多少？ Ans：6036.51

```
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
using namespace std;

int main()
{
    double area;
    int l=100,w=80,fi=50;
    const double PI=3.14159;

    area=(l*w)-(PI*(fi/2)*(fi/2));
    cout << "實際可用面積為：" << area << endl;

    cout<<endl;
    system("pause");
    return 0;
}
```