國立屏東大學 資訊工程學系 C++程式設計入門教材

5. 運算式

我們已經在過去的章節中提到：「絕大多數的電腦程式，都與資料的輸入、輸出與處理有關」，其中所謂的「處理」通常就是指對資料進行特定的運算，而這也就是本章的主題 – 運算式(Expression)。事實上，軟體的功能就是由一個個的運算式累積堆疊得來的，例如一個籃球遊戲軟體，包含比賽雙方的得分、選手個人投籃的命中率、籃板、助攻與犯規的次數等數據，都必須隨著遊戲的進行使用運算式來計算與更新；甚至就連選手投籃時，球的投射路徑、速度、力量，是否命中得分亦或是籃外空心，都必須經由運算才能得知。再者，遊戲進行時的所有動畫效果，也都是由一系列看似簡單的加、減、乘、除等運算來產生的。本章將從最基礎的運算式、運算子與運算元進行說明，再接續介紹C++語言所支援的各式運算，並提供相關的範例程式以幫助讀者學習。

5.1 運算式

運算式是由運算元(Operand)與運算子(Operator)所組成的，其作用是針對「特定的對象」進行「特定的運算操作」，並產生單一的運算結果。以一個簡單的運算式「x=6」為例，所謂的「特定的對象」就是運算元 – 變數「x」與整數值「6」，至於「特定的運算操作」則是代表要進行「賦值(Assigement)」[1)]操作的等號「=」 — 將等號右側的運算元之數值指派給左側的運算元，用白話文來講此運算式「x=6」就是要將變數x的數值設定為6。再舉另一個運算式「x + 3」為例，其運算元為變數「x」與整數值「3」，運算子的部份則是代表要進行「加法」運算的加號「+」，假設變數x的值為6，此運算式的運算結果將會是6與3的和，也就是整數值9。

要提醒讀者注意的是「C++語言的運算式可以單獨做為程式敘述，或是寫在其它的程式敘述中，例如下面的程式碼片段:

```
int main()
{
   x=6;
   cout << x + 3;
}
```

上面的程式碼，將「x=6」寫成了一行單獨存在的敘述，但是把「x+3」寫在用來輸出的cout敘述裡。要注意的是「C++的程式敘述必須以分號「;」結尾，所以不要忘了在第3行的「x=6]運算式後方加上結尾的「;」，否則編譯時將會發生錯誤。

我們將在以下的小節針對分別針對運算子與運算元做更清楚的說明：

5.2 運算子

運算子依其運算目的，需要搭配不同數目的運算元才能完成其運算，我們可其依所需的運算元數目，將運算子區分為以下三類：

- 一元運算子(Unary operator)「此種運算僅與一個運算元相關，例如用以表示數值的正或負的「符號」

即為此種一元運算子，這個正或負的符號必須放在其運算元的左邊，例如-3，+8其中的「-」與「+」號皆屬於此種一元運算子。
- 二元運算子(Binary operator)：此類運算與兩個運算元相關，例如常見的加法算術運算符號「+」，它必須完成將左右兩側的值進行加總的運算，例如3+8的運算結果為11。
- 三元運算子(Ternaryoperator)：此類運算與三個運算元相關，在C++語言中僅有「?:」為三元運算子，我們將在本書第6章加以介紹。

除了依照運算子所需搭配的運算元數目分類外，我們還可以根據運算性質的不同，將運算子區分為算術運算子(Arithmetic operator)、關係運算子(Relational operator)、邏輯運算子(logical operator)等不同類別，本章後續將會逐一地提供詳盡的說明，並且搭配相關的範例程式示範各種運算子的程式實作。

優先順序

當運算式擁有一個以上的運算子時，我們就必須考慮運算子的優先順序(Precedence)。首先，請考慮以下的運算式，它含有兩個運算子「+」與「×」：

```
x + 3 × 15
```

在此運算式裡有加法「+」與乘法「×」兩個運算子，它們會依照「先乘除、後加減」的原則，先進行「3 × 15」的運算，然後再將其結果加上x的數值 – 此處的「先乘除、後加減」就是運算子優先順序的觀念。

C++的運算式也可以如如數學式一樣，使用括號來提升運算的優先順序，但並沒有「大括號」、「中括號」與「小括號」之分，一律都是使用小括號( )來進行優先順序的提升，例如將x + 3 × 15使用括號改寫為(x + 3) × 15。如此一來，就會先進行x+3的運算，然後再將其結果與15進行乘法的運算。有時候，我們也會使用括號來把「隱含(Implicit)」的優先順序加以表明，例如將x + 3 × 15改寫為x + (3 × 15)。雖然根本不會改變其執行的順序與結果，但可以讓自己以及其他人更容易理解運算式的內容 – 這也屬於提升程式可讀性的一種方法。

還有一點要注意的是，由於電腦鍵盤上並沒有乘法符號(×)的按鍵，所以C++語言(以及絕大多數的程式語言)是以星號「*」來做為乘法的運算符號。因此，上述這個運算式，若使用C++語言則必須寫成下面的程式碼：

```
x + 3 * 15;
```

為了便利讀者學習，後續本書介紹到各個運算子時，都會為你說明其優先順序，並將所有常用的C++運算子的優先順序彙整於附錄 運算子的優先順序及關聯性供你查閱。

關聯性

在同一個運算式中，若有一個以上相同優先順序的運算子時，則必須依其關聯性(Associativity)方向逐一加以執行。所謂的關聯性可以分成左關聯與右關聯兩種：

- 左關聯(Left associativity)：意即在相同優先順序的情況下，由左往右加以計算，例如：i – j – k的執行順序應為(i – j) – k，也就是先執行左方的減法（也就是i – j），然後再將其結果與k進行相減。在C++語言中，大部份的二元運算子都是屬於左關聯。
- 右關聯(Right associativity)：與左關聯相反，右關聯是由右往左執行，例如：- + k(負的正k)會先執

行右方的「+」，然後才是左方的「－」，也就是說這個運算式會由右往左執行成為□- (+ k)□□
在C++語言中，大部份的一元運算子都是右關聯。

為了便利讀者學習，後續本書介紹到各個運算子時，都會為你說明其關聯性，並將所有常用的C++運算子的關聯性彙整於附錄 運算子的優先順序及關聯性供你查閱。

5.3 運算元

運算元在運算式中的角色就是「參與運算的對象」，本質上就是一個「數值」，包含數值(Value)□變數(Variable)或常數(Constant)等，都可以做為運算元。例如下面的運算式：

```
int a;
const int b;

123 + a;          // 使用一個數值123與變數a做為運算元
b * 10.2;         // 使用常數b與數值10.2做為運算元
```

除了數值、變數與常數之外，函式(Function)也可以做為運算元，因此以下的程式碼也是正確的：

```
123 + func(3); // 使用一個數值123與函式呼叫func(3)的結果做為運算元
```

函式是什麼？

請先思考以下的數學函數：

$f(x)=2x+5$□$f:\mathbb{Z}\rightarrow \mathbb{Z}$

這是一個名為f的函數定義，其定義域與對應域(值域)皆為整數(含正整數、負整數與0)，依其引數(Argument)x的數值，經計算$2x+5$後可得此函數的值，例如$f(1)=7$□$f(2)=9$□$f(3)=11$□餘依此類推。

C++語言也支援函數的定義，我們以前述的數學函數為例，定義一個C++語言的函數func如下(為了便利起見，我們也把對應的數學函數寫在其右側):

C++函數	數學函數
int func(int x) { return 2*x+5; }	$f(x)=2x+5$, $f:\mathbb{Z}\rightarrow \mathbb{Z}$

從上面的程式碼可以簡單地觀察到，就如同數學的函數一樣，此處的C++函數func在設計時，宣告了其定義域與對應域 — 都是int型態，以及定義了要如何使用引數x — 此處的用途就如同數學函數f一樣，計算2*x+5的運算結果，並做為函數的傳回值。當然，上述的說明還不夠清楚(尤其是完全沒有說明C++函數定義的語法)，不過對於現在來說，我們只要大致了解C++也能定義類似數學的函數即可，更詳細的說明請

在未來參考本書第X章 函式再為讀者詳細地加以說明。

有了上面的func定義後，我們在程式中就可以使用func(x)來得到此函數的計算結果，其中x就是每次使用時所給定的引數，可以是數值、常數、變數、甚至是其它的函式 — 只要它們的型態是int整數即可。例如：

```
int a, b;
a=func(1);
b=func(a);
cout << a << endl;
cout << b << endl;
```

不論是數學領域還是C++語言的程式設計，函數一詞都是來自同一個名詞Function，但本書為了與數學領域的函數區別，筆者“故意”將C++語言的Function改譯為函式，請讀者自行加以注意。此外，為了統一起見，本書後續將使用「函式名稱()」的型式，來表示C++語言的函式，例如前面的“func函式”就會寫成“func()函式” — 有沒有注意到，我們開始改稱為函式了。在程式碼中使用函式稱為「呼叫函式」 — 方法很簡單，只要寫下函式名稱與包含有引數的一組括號即可，例如上面例子中的func(1)與func(a)。

C++語言受到眾多程式設計師喜愛的原因之一，就是它提供了許多事先定義好的函式供我們使用(包含那些原本設計供C語言使用的函式，也可以在C++語言的程式裡使用)，這些函式的功能包羅萬象，善用這些事先寫好的函式來完成程式功能，就可以大幅縮短軟體開發的時間。

5.4 算術運算子

算術運算子(Arithmetic Operator)所進行的就是數學的算術運算，包含大家熟悉的正號、負號、加、減、乘、除，在此筆者將讀者該注意的重點列示如下：

- 正號與負號都是右關聯的一元運算子(Unary Operator)，其它的算術運算子都是左關聯的二元運算子(BinaryOperator)，在使用上只要在運算元前加上「+」或「-」即可用來表示正數或負數，例如+5」或-x。
- 乘法運算子使用的是星號字元「*」
- 除法運算子使用的是斜線字元「/」，就如同數學的分數一樣。
- 餘除(modulo)，可能是初學者比較不熟悉的運算，它的作用是進行除法的運算，但取的是餘數而不是商數。還要注意的是，餘除運算子使用的是百分比字元「%」。

關於C++所支援的算術運算子完整列表，請參考table 1。

運算子(Operator)	意義	一元/二元運算(Unary/Binary)	關聯性(Associativity)
+	正號	一元	右關聯
-	負號	一元	右關聯

運算子(Operator)	意義	一元/二元運算(Unary/Binary)	關聯性(Associativity)
+	加法	二元	左關聯
-	減法	二元	左關聯
*	乘法	二元	左關聯
/	除法	二元	左關聯
%	餘除	二元	左關聯

Tab. 1: <nowiki

至於在算術運算子的優先順序(Precedence)方面，請參考table 2。

優先順序(Precedence)	運算子(Operator)
高	+，- (一元的正負號)
中	*、/、%
低	+，- (二元的加減)

Tab. 2: 算術運算子的優先順序

如果需要更完整的運算子優先順序與關聯性資訊，可參考本書附錄B。

本節最後以一個範例說明算術運算子的使用方式：

Example 1

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

int main()
{
  int x=100;
  int y=30;

  cout << "x=" << x << " y=" << y << endl;
  cout << "x+y=" << x+y << endl;
  cout << "x-y=" << x-y << endl;
  cout << "x*y=" << x*y << endl;
  cout << "x/y=" << x/y << endl;
  cout << "x%y=" << x%y << endl;
}
```

在Example 1的arithmetic.c中，宣告了兩個int型態的變數x與y，其值分別為100與30，並使用cout將它們的

數值以及進行各個算術運算的結果加以輸出，其執行結果如下：

```
x=100 y=30
x+y=130
x-y=70
x*y=3000
x/y=3
x%y=10
```

請仔細看一下Example1的執行結果，有沒有發現除法的部份結果並不正確？因為x/y的結果應該是3.33333，但程式執行的結果卻是顯示3！到底下面這行程式碼到底出了什麼問題？

```
  cout << "x/y=" << x/y << endl;
```

這是因為在計算x/y的結果時，由於在除法「/」這個二元運算子的左右兩側都是int型態的數值，因此其計算結果也會「自動」地轉換為int型態 – 這種情況稱為「隱性轉換(Impilict Conversion)」，所以原本應該是3.33333的數值，就被自動地轉換成為了整數3。

既然問題是出在「隱性轉換」上，那麼解決的方法很簡單，我們就來做個「顯性轉換(ExplicitConversion)」就可以了！請參考下面的修改：

```
  cout << "x/y=" << (float)x/y << endl;
```

我們在這個除法左右兩側，任意選取一個運算元(當然也可以兩個都做)，在其前方加上一個「(float)」來強迫地把它轉換為float型態的數值即可。這個做法使得原本數值為100的變數x被編譯器視為是數值為100.0的float型態浮點數，如此一來，在除法的左右兩側現在至少有一個運算元是浮點數了，因此其運算結果也將會是浮點數的3.33333！我們又常把這種做法稱為「型態轉換(Type Casting)」，可以視需要將數值轉換為不同型態，但要注意不是每種轉換都是正確的，例如將一個浮點數轉換為整數時，其小數點後的數值將會消失，運算的結果也就會變得不再精確。現在，請自行修改Example 1的程式碼，讓它能輸出正確的結果。

使用型態轉換得到正確的計算結果

現在讓我們再回想出現在第三章3.2.3 門號違約金計算程式裡的Example 5，還記得它因為除法運算遇到型態的問題導致計算結果不正確嗎：

```
// 函式標頭檔區（Header File Inclusion Section）
#include <iostream>

// 命名空間區（Namespace Declaration Section）
using namespace std;

// 程式進入點（Entry Point）
int main()
{
```

```
 //變數宣告區（Variable Declaration Section）
   int contractDays;          // 合約總日數
   int contractRemainingDays; // 合約剩餘日數
   int monthlyFeeDiscount;    // 每月月租費優惠金額
   int subsidy;               // 手機補貼款
   int compensation;          // 違約金

 //輸入階段區（Input Section）
   cout << "請輸入合約總日數:";
   cin >> contractDays;
   cout << "請輸入合約剩餘日數:";
   cin >> contractRemainingDays;
   cout << "請輸入合約期間每月月租費優惠金額:";
   cin >> monthlyFeeDiscount;
   cout << "請輸入手機補貼款金額:";
   cin >> subsidy;

 //處理階段區（Process Section）
   compensation = ( (monthlyFeeDiscount/30) *
                  (contractDays-contractRemainingDays) +
subsidy ) *
                  (contractRemainingDays/contractDays);

 //輸出階段區（Output Section）
   cout << "您必須支付的違約金額為" << compensation << "元" <<
endl;
   return 0;
}
```

此程式的問題就出在其中的第28行與第30行 — 兩個整數值相除將只能得到整數的結果(儘管它並不精確)。除了將所有int整數變數都改宣告為浮點數以外，這個程式還可以使用「型態轉換」來得到同樣正確的結果。首先，由於我們把最終的計算結果存放到變數compensation裡，所以“ 必須 ”將它改宣告為浮點數型態：

```
   float compensation;          // 違約金改宣告為浮點數
```

接著，我們將第28行與第30行分別修改如下：

```
   compensation = ( (monthlyFeeDiscount/30.0) *
                  (contractDays-contractRemainingDays) +
subsidy ) *
((float)contractRemainingDays/contractDays);
```

我們在第三章3.2.3 門號違約金計算程式節已說明過，此程式的問題就出在「(monthlyFeeDiscount/30)」與「contractRemainingDays/contractDays」這兩個除法運算上；因此，我們可以使用顯性轉換，來將除法的左右兩側做為除數與被除數的整數擇一將其轉換為浮點數(當然也可以兩個都轉換)，如此一來這個除法的運算結果就不再會是整數。針對第28行，我們可以選擇將monthlyFeeDiscount變數轉換為浮點數，或是向上面的做法一樣，將整數值30改寫為浮點數30.0(儘管數值是一樣的，但其型態已經不同)。至於第30行，我們則將被除數轉換為float浮點數型態。如此一來，其計算的結果最會變得正確了！

養成檢查除法型態的習慣

為了確保計算結果正確，建議讀者養成檢查除法型態的習慣，遇到「int整數/int整數」時，若是整數值就直接在其後補上.0，讓它變成浮點數數值；若是整數變數，那就在其前面加上(float)或(double)強制將它轉換為浮點數型態。

5.5 賦值運算子

等號「=」在C語言中被稱為賦值運算子(Assignment Operator)，用以將等號右側的數值賦與(assign)給等號的左側。雖然它和數學裡的等號是同一個符號，但意義與作用並不相同！如果把「=」想像為「←」可能會更為貼切「賦值」的意涵，例如本章稍早前已使用過的「x=6」，將其想像成「x←6」將會更為貼切其意涵 — 將數值6賦與給變數x(也就是將變數x的數值設定為6)。但是想像一下就好，電腦鍵盤上並沒有「←」這個符號，大部份的高階程式語言和C++一樣都是使用「=」做為賦值之用[2)]。

依語法規定，賦值運算子(也就是等號＝)的左側只能有一個單一的變數(用來接收右側的運算結果)，至於右側的內容則可以是一個變數、數值、函式呼叫，甚至是另一個運算式，請參考以下的例子：

```
i = 5;    // 將數值5賦與給變數i，或是「將變數i設定為5」，也就是 i <-- 5
j = i;    // 將變數i的數值賦與給變數j，或是「將變數j設定為變數i的數值」，也就是 j <-- i
k = 10 * i + j;  // 將「10*i+j」的運算結果賦與給變數k，也就是 k <-- (10*i+j)
```

在上述的例子當中，前兩個運算式敘述非常單純，都僅擁有一個賦值運算子(也就是😀)，其結果也都相同的把等號右側單一運算元的數值賦與給等號左側的變數。但後兩個運算式就稍稍複雜一些，以「k = 10 * i + j」這個運算式為例，其中共有 = 、 * 與 + 三個運算子 — 當一個運算式存在有一個以上的運算子時，其執行就必須依照優先順序決定，C++語言規定賦值運算子 = 的優先順序比所有算術運算子都低，所以先執行的將會是 * 與 + 運算子，依據上一小節的說明，* 的優先順序高於 +，所以整個運算式可以被視為「k ← ((10*i) + j)」— 先進行等號右側的運算，再將其結果賦與等號左側的變數裡。

要注意的是，當賦值運算子左右兩側的資料型態不一致時，C語言將會進行自動的隱性型態轉換(implicit conversion)，假設我們宣告兩個變數i與j分別為int與float型態，以下的程式碼片段故意將不同型態的值指定給這兩個變數：

```
int i;
float j;

i = 3.1415f;
j = 100;
cout << "i=" << i << endl; // 將float型態的數值3.1415f賦與給int型態的i，其結果i的數值將會是3
cout << "j=" << j << endl; // 將整數值100指定給float型態的j，其結果j的數值將會是j=100.0
```

由於自動所進行的隱性型態轉換，i = 3.1415f;，等同於，i = (int)3.1415f;，— 要特別注意的是，當浮點數轉換為整數時，原本小數的部份將會被無條件捨去。另一方面，前面例子中的，j=136;，將等同於，j=(float)136;， — 將整數值轉換為浮點數時，小數部份為0，意即136將會變成136.0。上述程式碼片段的輸出將會是：

```
i=83
j=136
```

要注意的是，使用cout輸出136.0時，儘管它是一個浮點數，但因為其小數部份為0，所以只會輸出其整數的部份。

使用函式來完成四捨五入

在前面的例子裡，我們提到，i=3.1415;，會自動進行隱性型態轉換，並且小數的部份會被無條件捨去；但如果讀者所需要的是將小數的部份「四捨五入」的話，則可以考慮呼叫事先寫好的函式來完成，請參考以下的範例程式：

```
#include <iostream>
#include <cmath>
using namespace std;
int i, j;

i = round(3.1415f);
j = round(4.62f);

cout << i << endl;
cout << j << endl;
```

在使用C++語言事先定義好的函式時，有一點是要特別注意的，那就是必須要先使用#include來將定義函式的標頭檔(Header File)載入才行 — 如此一來，你的程式才會“ 認識 ”這個函式。此處的round()是其實C語言所提供的函式，定義於math.h標頭檔裡。在C++語言裡，我們仍然可以使用C語言的函式，只是在載入標題檔時要把原先的標頭檔名去掉副檔名，並在前面冠以一個c — 請參考上述程式的第2行，我們以#include <cmath>將C語言的math.h標頭檔加以載入。後續在第6行及第7行，我們呼叫了兩次round()函式，分別以3.1415f與4.62f做為引數。由於這個round()函式

會幫我們把引數的小數部份進行四捨五入到整數位，所以此程式的輸出結果將會如下：

```
3
5
```

好了，以後有機會的話，再和讀者多介紹一下C語言及C++語言所支援的函式；當你學會使用愈多的函式，以後寫起程式來就會愈來愈簡便了！

不使用函式來完成四捨五入

除了呼叫round()函式以外，將小數的部份「四捨五入」到整數位還有其它的做法，請參考以下的程式碼：

```
i= (int)(3.1415f + 0.5);
```

此處的做法是將要進行小數部份「四捨五入」到整數位的數值(當然也可以是變數)，先加上0.5之後，再使用強制性的「顯性型態轉換」將其小數部份無條件捨去，如此一來，原始的數值就完成了將小數部份「四捨五入」到整數位的操作了！

另外還要注意的是，賦值運算子的關聯性為右關聯，所以當一個運算式中有多個等號出現時，將會依照由右往左的方向加以執行，例如：

```
a = b = c = 0;
```

等同於

```
a = ( b = ( c = 0 ) );
```

上述的這種做法，就可以同時將多個變數的數值都設定為同一個數值。

5.6 複合賦值運算子

依句程式設計的需求，有些時候在賦值運算子(也就是 =)左側的變數，也會出現在運算式的右側。例如有一個變數score代表某位學生的成績，其原本的分數為58分，若執行了以下的運算式就可以幫他再加二分：

```
score = score + 2;
```

依據運算子 = 與 + 的優先順序，此運算式敘述將會先進行 score + 2 的運算，得到結果為60之後，再將數值60賦與給在等號左側的變數score。如此一來，這位同學就從不及格變成及格了！

針對這種將某個變數的原始數值進行運算後的數值，再做為其新的數值的情況，C++語言提供了複合賦值運算子(Compound Assignment Operator)，讓我們可以使用更為便捷的方式來完成。就好比「縮寫」一樣，score = score + 2;可以使用複合賦值運算 += 改寫如下：

```
score += 2; // 將score變數的數值加2
```

我們把這種將原本的數值進行加法運算後，做為變數新的數值的 += 運算子，稱為「以和賦值」的複合賦值運算子。除了「以和賦值」之外，C++語言針對算術運算子還有提供其它的複合賦值運算子，請參考table 3。

複合賦值運算子	意義	範例
+=	以和賦值	i+=n 等同於 i=i+n
-=	以差賦值	i-=n 等同於 i=i-n
*=	以積賦值	i*=n 等同於 i=i*n
/=	以商賦值	i/=n 等同於 i=i/n
%=	以餘賦值	i%=n 等同於 i=i%n

Tab. 3: <nowiki

在table 3中，所有的複合賦值運算子皆為二元運算子，而且都是右關聯，若是在一個運算式中出現多個複合賦值運算子時，則必須由右往左加以執行，請考慮下列的運算式：

```
a += b += c;
```

它將等同於以下的運算式：

```
a += ( b += c);
```

也就是先把c的數值加上b做為b的新數值，然後再把b的新數值加上a的數值做為a的新數值。

5.7 遞增與遞減運算子

前一小節所介紹的複合賦值運算子，可以視為是一種精簡的縮寫 — 例如將 x = x + y 縮寫為 x += y 。如果我們要使用複合賦值運算子來將變數x的數值加1或減1的話，除了可以寫做 x += 1 與 x -= 1 之外，C++還有提供另一種更精簡的運算子：

- ++：遞增運算子(Increment Operator)，讓變數值加1。
- --：遞減運算子(Decrement Operator)，讓變數值減1。

使用上述兩個運算子，我們可以把 x = x + 1 或 x += 1 改寫為 x++，這樣同樣可以讓i的數值加1，但卻

更為精簡；同理 x-- 則可以用來遞減x的數值。

不過這兩個遞增與遞減運算子的作用還不只有這樣而已，它還可以依據放置在運算元(也就是變數)的前面或後面，再區分為前序運算子(Prefix Operator)與後序運算子(Ppostfix Operator)當我們把 ++ 寫在變數的前面時，就稱為「前序遞增運算子(Prefix Increment Operator)寫在後面則稱為「後序遞增運算子(Postfix Increment Operator)同樣地， -- 寫在前面與後面則被稱為「前序遞減運算子(Prefix Decrement Operator)與「後序遞減運算子(Postfix Decrement Operator)以遞增為例 ++x 會先遞增x的數值，然後再傳回新的x的數值；但若是寫在後面(也就是 x++)的話，則會先傳回x現有的數值，然後才將x的數值遞增。請考慮以下的程式碼：

```
x=1;
cout << "x is" << x   << endl;  // 印出x的數值
cout << "x is" << ++x << endl;  // 先將x的數值遞增，然後才印出其數值
cout << "x is" << x++ << endl;  // 先印出x的數值，然後才將其數值遞增
cout << "x is" << x   << endl;  // 印出x的數值
```

其執行結果如下：

```
x is 1
x is 2
x is 2
x is 3
```

現在，讀者先在此花一點時間想一想，為什麼輸出的結果是這樣呢？首先，在第1個cout敘述裡，我們要印出的是x原本的數值，也就是1。接著在第2個cout敘述裡，由於 ++ 運算子是前序的，所以會先將x的數值遞增，然後才印出其數值，也就是2。至於在第3個cout敘述裡的 ++ 是後序的，所以會先印出x原本的數值，也就是2，然後才將其數值遞增為3。最後第4個cout敘述則幫我們把剛才遞增後的x數值印出，也就是3。

本節最後要提醒讀者，在C++語言中只有加法與減法有遞增與遞減的寫法，並沒有 ** 、 // 與 %% 運算子。試想，任何數字乘以1、除以1仍等於其本身，至於餘除(進行除法後取餘數)也是類似的結果，任何數字除以本身，其餘數一定是0。所以 ** 、 // 與 %% 運算子，根本沒有存在的必要。

5.8 逗號運算子

在C++語言的運算式中還有一種較為特殊的運算子 — 逗號運算子(Comma Operator)它可以在同一個運算式裡放入多個使用逗號「,」加以分隔的運算式，然後依序由左至右進行運算(所以逗號運算子是左關聯)，最終使用最右側的運算式的運算結果做為整個運算式的結果。請參考以下的Example 2

Example 2

```
#include <iostream>
```

```
using namespace std;

int main()
{
   int a, b, c, d;
   a=b=c=d=3;
   d = (a=b+c, b+=1, c=a+b);
   cout << "a=" << a << endl;
   cout << "b=" << b << endl;
   cout << "c=" << c << endl;
   cout << "d=" << d << endl;
}
```

此範例程式的執行結果如下：

```
a=6
b=4
c=10
d=10
```

現在讓我們來說明Example 2的結果是怎麼產生出來的？首先在comma.cpp的第6行，我們宣告了4個int整數變數a、b、c與d、並在第7行將它們的數值都設定為3。在接下來的第8行裡，由於使用了括號，所以會先進行 (a=b+c, b+=1, c=a+b) 的運算，然後再把其結果賦與給變數d、其實在 (a=b+c, b+=1, c=a+b) 裡包含了3個由2個逗號運算子所分隔開的3個運算式，因此它們將會由左至右(沒忘記我們剛說過逗號運算子是左關聯吧！？)進行運算，分別是 a=b+c 、 b+=1 以及 c=a+b 、我們將其依序運算的結果分述如下：

- 首先a=b+c會讓a的數值變成b+c的結果，也就是6
- 接著b+=1讓b的數值加1變成4
- 最後的c=a+b、則會讓c的數值變成a+b的結果，但要特別注意此時a的數值已經在前面的a=b+c運算式後變為了6、b的數值則在b+=1運算後變成了4，所以這個c=a+b的運算會讓c變成10。

讓我們再回到第8行的運算敘述、 d=(a=b+c, b+=1, c=a+b); 、、由於括號裡由逗號分隔的3個運算式，最終會以最右側的運算式的運算結果做為整個運算的結果，因此第8行的運算結果其實等同於、d=(10);、— 這是因為最後一個、最右側的運算結果是10的緣故。說明至此，相信讀者已經能夠理解Example 2的執行結果了。

> 在本節結束之前，有一個問題想讓讀者試著回答看看：「如果將Example 2的comma.cpp的第8行改為、 d= a=b+c, b+=1, c=a+b; 、、那麼其執行結果還會相同嗎？如果不同，結果是什麼呢？」

建議讀者可以自行修改程式以得到答案，或是參考本章末的解答 — 不過要記住，答案永遠不是那麼不重要，重要的是你知不知道答案的原理是什麼？ 先試著自己想一想吧！

5.9 取址運算子

我們在第4章已經說過，在程式中所宣告的變數，會在記憶體裡分配到一塊適合的空間供其存放數值。如

果你需要知道變數所配置到的記憶體位址在哪？你就可以使用本節所要介紹的「取址運算子(Address-Of Operator)」來達成。

為何需要知道變數所在的記憶體位址？ 到目前為止，我們所寫的程式大概都還沒有需要使用到變數所在的記憶體位址。我們只要能夠使用存放在變數裡面的數值即可，不需要知道它到底放在哪？但是，在本書後續的章節說明中，我們將會看到讓C++功能如此強大的原因 — 指標與參考！屆時我們將會有更清楚的說明，你就會理解為何需要知到變數在哪裡了！先別急，慢慢看下去，時間到了，你就會知道答案了！

取址運算子可以讓我們取得運算元所在的記憶體位址，其運算符號就是代表「And」的 & — 你可以直接將其唸做And」或是依其作用唸做Address Of(因為這個符號在C++程式的作用是取回變數的記憶體位址，所以唸做Address Of應該更為適當)。它是一個右關聯的一元運算子，直接寫在要取回記憶體位址的運算元前面(也就是左側)即可。取址運算子可以取回記憶體位址的運算元包含變數、常數、函式、陣列、指標等，但這已經超出本章範圍許多，所以本節將僅討論有關變數與常數的記憶體位址，至於其它的部份我們將留待未來使用到時再加以說明。

請參考以下的程式片段，它可以幫我們印出int整數變數x與常數y所配置到的記憶體位址：

```
int x;
const int y=0;
cout << "x is located at " << &x << endl;
cout << "y is located at " << &y << endl;
```

其「可能的」執行結果如下：

```
x is located at 0x16ef8f3e4
y is located at 0x16ef8f3d4
```

為什麼筆者在此要說是「可能的」執行結果呢？這是因為現代的作業系統使用ASLR的技術，來保護我們不被「駭客」攻擊的緣故，同一個程式每次執行時所配置得都會是不同的記憶體空間，所以筆者才會說上述的執行結果只是「可能的」結果，「真正的」會所印出的記憶體位址將依實際執行結果而有所不同。關於ASLR可參閱本書第somewhere in the book」

在上述的結果當中，讀者必須特別注意以下幾點：

- 記憶體位址是以16進制的數值呈現，所以你會看到它會標記為0x開頭(讀者可以參考本書somewhere關於16進制整數的說明)。
- & 所取回的記憶體位址，其實是指變數所配置到的記憶體空間的「起始位址」(也就是「開頭」的位址)，以一個佔4個byte的int整數變數x為例，若其起始位址為0x16ef8f3e4」那麼就表示它所配置到的是0x16ef8f3e4」0x16ef8f3e5」0x16ef8f3e6以及0x16ef8f3e7等連續的4個byte的空間。

5.10 sizeof運算子

sizeof運算子存在的目的是讓我們可以取得特定「變數」、「常數」或「資料型態」所佔的記憶體空間大小，其單位是位元組(byte)。它是右關聯的一元運算子，其使用語法如下：

sizeof使用語法

sizeof 識別字 | (識別字) | (型態名稱)

資訊補給站：表示 or（或者）的語法符號

在上面的語法定義中，我們使用「|」符號表示or（或者），意即在其左右兩側的語法構件中進行二擇一的選擇。後續本書將繼續使用此種表示法做為語法的說明。

做為一個一元運算子，sizeof必須寫在其運算元的左側。在上述的語法定義中，sizeof的運算元共有三個選擇：「識別字」、「(識別字)」與「(型態名稱)」，其中「識別字」是我們所想要取回其記憶體空間大小的「變數」或「常數」的名稱(還記得嗎？變數名稱或常數名稱都被稱為識別字，請參閱somewhere)，至於「型態名稱」則代表的是想要取回記憶體空間大小的資料型態名稱。由於在語法中使用了兩個 | 符號(or)，所以就是進行三選一的意思。若要使用sizeof來取得一個變數或常數所佔用的記憶體空間大小，那麼前兩個選擇都可以，例如以下的程式碼：

```
int x;
const int y=0;

cout << "The size of x is " << sizeof(x) << endl;
cout << "The size of y is " << sizeof y << endl;
```

其「可能的」執行結果如下：

```
The size of x is 4
The size of y is 4
```

為什麼又是「可能的」執行結果？

現代的電腦系統多半使用4個byte的int整數，但有些早期的電腦系統或是現代的小型嵌入式系統，其整數是2個byte，甚至現在有些大型的電腦系統使用8個byte的整數。因此，此處只能說是「可能的」結果，其「真正的」結果將視實際執行結果而定。

從上述的例子來看，使用sizeof取得變數或常數的記憶體空間大小時，只要後面接著其名稱即可，不論有

或沒有括號皆可。但是如果要使用sizeof來查詢資料型態的大小時，依語法就只能選第三種方法，一定要在型態名稱的前後加上括號才行，例如：

```
cout << "The size of int type is " << sizeof(int) << endl;
cout << "The size of double type is " << sizeof(double) << endl;
```

其「可能的」執行結果如下：

```
The size of int type is 4
The size of double type is 8
```

5.11 位元運算子

不同於我們日常生活中所使用的十進制，電腦系統所使用的是二進制的數字系統，C++語言也提供了相關的運算子，稱為「位元運算子(Bitwise Operator)」，讓我們可以進行二進制的數值運算。table 4是C++語言所支援位元運算子：

運算子(Operator)	意義	一元/二元運算(Unary/Binary)	關聯性(Associativity)
«	左移	二元	左關聯
»	右移	二元	左關聯
&	Bitwise AND	二元	左關聯
|	Bitwise OR	二元	左關聯
^	Bitwise XOR	二元	左關聯
~	Bitwise NOT(補數)	一元	右關聯

Tab. 4: <nowiki

在table 4中，位元位移(Bitwise Shift)是將二進制的數值進行左移(Left Shift)或右移(Right Shift)的運算，其因位移後所產生的空位則一律以0填補。例如十進制的數值36等同於二進制的100100，若將其進行一次左移的運算，其數值將變為1001000，也就是十進制的72；若再左移一次，則其數值將變為10010000，也就是十進制的144。另一方面，若是將十進制的數值36，進行一次右移的運算，則其數直將從100100變為10010，也就是十進制的18；若再右移一次，則從10010又變為1001，也就是十進制的9。從上述的說明可以看出，若將一個二進制的數值進行左移或右移的運算，就等同於該數值進行乘以2或除以2的運算，所以也常被用以代替乘法與除法的算術運算[3)]。

C++語言使用 « 與 » 做為左移與右移的運算子， 使用時在運算子左側接要進行位移的數值，並在右側接要位移的次數(也就是要位移的位數）。請參考以下的程式片段：

```
int a=36;
cout << (a<<1) << endl; // 將a左移一次(等同於乘2)，然後交由cout輸出
cout << (a<<2) << endl; // 將a左移二次(等同於乘4)，然後交由cout輸出
cout << (a>>2) << endl; // 將a右移二次(等同於除4)，然後交由cout輸出
```

上述程式片段的執行結果如下：

```
72
144
9
```

« 是左移還是流出？ » 是右移還是流入？

讀者可能會對本節所介紹的位元位移運算子(Bitwise Shift Operator)感到疑惑，沒錯，左移與右移的運算子符號與我們使用在cout與cin時的串流運算子完全相同！同樣都是 « 與 » ！那麼我們寫在程式裡面的 « 與 » 到底會被視為是左移、右移，還是流出、流入呢？答案是C++語言的編譯器會依據其所搭配的運算元來決定，若在 « 或 » 的兩側都是數值，那麼它們就會被視為是位元位移的運算子，若是左側是cout或cin這一類的串流物件，那麼就會被視為是流出與流入。因此，請特別注意像是□cout « (a«1) « endl;□這樣的一行敘述，如果少了其中的括號而變成了□cout « a«1 « endl;□□那麼它的輸出結果，也會從72變成了361了！ — 少了括號後□a«1就從先執行完位移運算後再流出給cout□變成了直接流出給cout的數值內容了(流出的a的數值36，以及流出整數值1)。

除了位移以外，table 5還列出了C++所支援的其它位元運算子 —「位元邏輯運算子(Bitwise Logical Operator)□包含了對數值進行的「位元AND□□□位元OR□□□位元SOR□與「位元NOT□等運算，我們將這些位元邏輯運算子的運算結果列示於table 5□

a	b	位元AND(a&b)	位元OR(a|b)	位元XOR(a^a)	位元NOT(~a)
0	0	0	0	0	1
0	1	0	1	1	
1	0	0	1	1	0
1	1	1	1	0	

Tab. 5: <nowiki

C++語言的位元邏輯運算子，都是所謂的「逐位元運算子」，意即其運算是針對二進制數值的每一個位元，進行相關的運算。從table 5可得知，位元AND□位元OR與位元XOR的運算子符號分別為 & 、| 與 ^ ，從表中可以得知當兩個位元a與b進行位元AND運算時，只有在a與b都為1的情況下，其運算結果才會為1，其餘情況皆為0；當a與b進行的是位元OR運算時，只要a或b兩者之中至少有一個為1，其運算結果就會為1，否則為0(也可以換句話說，只要a與b不是兩者皆為0，則其運算結果就會為1)。至於位元XOR運算，則是所謂的互斥OR運算(Exclusive OR)□只有在a與b兩者的值不相等時，其運算結果才會為1，否則為0。最後在<tab tab_BitwiseLogicalOperator>中還有一個位元NOT運算子，其符號為 ~，它的作用是將二進制數值裡的0變為1、1變為0。

本節所介紹的位元運算其實對於程式設計的初學者來說，通常會感到相當陌生 — 因位二進制並不是我們日常生活中慣用的數字系統。但是對於資訊領域來說，是非常重

要且普遍的運算，因為電腦系統所使用的就是二進制的數字系統，在許多應用問題上，位元運算可說是不可或缺的一部份。然而，位元運算的基礎通常涵蓋於資訊學系大一的計算機概論或數位系統導論等課程，已超出本書範圍，在此不予贅述。有興趣的讀者請自行參閱其它相關教材。

最後，要提醒讀者的是位元運算子也可以和賦值運算子共同使用，形成所謂的位元複合賦值運算子(Bitwise Compound Assignment Operator)，請參考table 6，這些位元複合賦值運算子都是右關聯的二元運算子，其作用都是將運算子左側的運算元與右側的運算元進行特定的位元運算後，將結果寫回到運算子左側的運算元裡。舉例來說，a&=b代表的是a與b進行位元AND的運算並將結果寫回至a，等同於a=a&b。

複合賦值運算子	意義	範例
&=	以位元AND賦值	i&=n 等同於 i=i&n
|=	以位元OR賦值	i|=n 等同於 i=i|n
^=	以位元XOR賦值	i^=n 等同於 i=i^n
<<=	以位元左移賦值	i<<=n 等同於 i=i<<n
>>=	以位元右移賦值	i>>=n 等同於 i=i>>n

Tab. 6: <nowiki

5.12 關係與邏輯運算子

所謂的關係運算子，relational operator，是用以比較兩個數值間的關係，例如大於 >、小於 < 等運算。至於邏輯運算子，logical operators，則是對數值進行Boolean值的運算，包含AND，OR，XOR，exclusive or，與NOT，不論是關係或是邏輯運算子，通常都和C語言的條件判斷敘述結合使用，因此本章將略過此部份，完整的說明及程式範例請讀者參考本書第7章的7-1節 邏輯運算式。

5.13 本章內容回顧

以下我們為讀者彙整了本章的學習重點：

- 運算式(Expression)是由運算元(Operand)與運算子(Operator)所組成的，其作用是針對「特定的對象」進行「特定的運算操作」，並產生單一的運算結果。
- 運算子可依其所搭配的運算元數目，將運算子區分為以下三類：
  - 一元運算子(Unary operator)，此種運算僅與一個運算元相關，例如用以表示數值的正或負的「符號」即為此種一元運算子。
  - 二元運算子(Binaryoperator)，此類運算與兩個運算元相關，例如常見的算術運算符號+、-、*、/、%等。
  - 三元運算子(Ternary operator)，此類運算與三個運算元相關，在C++語言中僅有 ?: 為三元運算子。
- 優先順序(Precedence)，每個C++的運算子都擁有事先定義的優先順序，當多個運算子出現在同一個運算式時，就會依照其優先順序(Precedence)加以執行。
- 關聯性(Associativity)，每個C++的運算子都擁有事先定義的關聯性，用以決定在同一個運算式中的多個相同優先順序的運算子的執行順序，可區分為：
  - 左關聯(Left associativity)，由左往右加以計算，例如，i - j - k，的執行順序應為，(i - j) - k，在C++語言中，大部份的二元運算子都是屬於左關聯。
  - 右關聯(Right associativity)，由右往左執行，例如，- + k，(負的正k)會先執行右方的「+」，然後才是左方的「 - 」。在C++語言中，大部份的一元運算子都是右關聯。

- 本書附錄 運算子的優先順序及關聯性彙整了C++各個運算子的關聯性與優先順序，請有需要的讀者自行參考。
- 運算元在運算式中的角色就是參與運算的對象，包含數值(Value)、變數(Variable)、常數(Constant)或函式(Function)等，都可以做為運算元。
- 算術運算子(Arithmetic Operator):就是數學的算術運算，包含正號+、負號-、加+、減-、乘*、除/與餘除%。
- 賦值運算子(Assignment Operator)、使用 = 做為其運算子符號，其作用是等號右側的數值賦與(assign)給等號的左側。
- 複合賦值運算子(Compound Assignment Operator):包含算術運算子與位元運算子都可以和賦值運算子共同使用，以類似「縮寫」的方式，將 a=a op b的運算式改寫為 a op=b、其中 op= 代表與賦值運算子共同結合的運算子，例如屬於算術運算的 +=、-=、*=、/=、%=，以及位元運算的<<=、>>=、&=、|=、^=。
- 遞增與遞減運算子: 像 a=a1 與 a=a-1 這種型式的運算式，可以改寫為 a++ 與 a-、代表將a的數值進行遞增與遞減的操作。這兩個運算子依其在運算元的前後，右可再區分為：
  - ++：遞增運算子、Increment Operator、、讓變數值加1。
  - –：遞減運算子、Decrement Operator、、讓變數值減1。
- 逗號運算子(Comma Operator):在同一個運算式裡，可以使用逗號 , 分隔多個運算式，並依序由左至右進行運算，最終使用最右側的運算式的運算結果做為整個運算式的結果。
- 取址運算子(Address-Of Operator):使用 & 符號讓我們取得運算元所在的記憶體空間的起始位址。
- sizeof運算子：取得特定變數、常數或資料型態所佔的記憶體空間大小，其單位是位元組(byte)、
- 位元運算子(BitwiseOperator)、針對電腦系統所使用的二進制數字系統、C++語言也提供了位元位移、位元邏輯等運算子：
  - 位元位移(Bitwise Shift Operator)、將二進制的數值進行 « 左移(Left Shift)或 » 右移(Right Shift)的運算，並將其因位移後所產生的空位以0填補。一個二進制的數值進行左移或右移的運算，就等同於該數值進行乘以2或除以2的運算。
  - 位元邏輯運算子(Bitwise Logical Operator)、是二進制數值的逐位元運算，以a與b兩個數值為例：
    - 位元AND、符號為&，逐一針對a與b的每個位元運算，只有a與b都為1的情況下，其運算結果才會為1，其餘情況皆為0。
    - 位元OR、符號為|，逐一針對a與b的每個位元運算，只有a與b至少有一個是1的情況下，其運算結果才會為1，其餘情況皆為0。
    - 位元XOR:符號為^，逐一針對a與b的每個位元運算，只有a與b的數值不相同時，其運算結果才會為1，其餘情況皆為0。
    - 位元NOT、符號為~，這是一個一元運算子，針對所給定的數值進行逐位元運算，若其值為0則變為1、其值為1則變為0。
- 關係運算子(Relational Operator)是用以比較兩個數值間的關係，請參考本書第6章的6-1節、

5.7節問題解答

---

[1] Assignment直譯應為「指定」、「指派」，但此處筆者取其意涵譯做「賦值」，代表其將等號右側的數值賦與給左側變數之意。
[2] 少部份的高階程式語言，例如Pascal語言，是使用「:=」做為賦值之用。
[3] 位移運算的效率比起算術運算的乘法與除法效率高得多，因此左移與右移常被用來進行乘2與除2的運算。

From:
https://junwu.nptu.edu.tw/dokuwiki/ - **Jun Wu的教學網頁**
國立屏東大學資訊工程學系
CSIE, NPTU
Total: 194932

Permanent link:
https://junwu.nptu.edu.tw/dokuwiki/doku.php?id=cppbook:ch-expression

Last update: **2024/02/29 07:09**