運算子與運算式

甚麼是運算式？

- 在 C語言中, 大部分的敘述都是由運算 式 所構成, 至於運算式則是由一組一組的運算子 與運算元組成。其中, 運算子代表的是運算的種類(或者說是運算符號), 而運算元則是要運算的資料。舉例來說：

```
5 + 3
```

- 其中 + 就是運算子, 代表要進行加法運算, 而要相加的則是 5 與 3 這兩個資料, 所以 5 與 3 就是運算元。

甚麼是運算式？

- 不同的運算子所需的運算元數量不同, 像是剛剛所提的加法, 就需要二個運算元, 這種運算子稱為**二元**運算子如果運算子只需單一個運算元, 就稱為**單元**運算子(或稱為一元運算子)。

- 另外, 運算元除了可以是字面常數以外, 也可以是變數, 例如：

```
5 + i
```

甚麼是運算式？

- 運算元也可以是另外一個運算式, 例如：

```
5 + 3 * 4          // 可看成是 5 和 (3 * 4) 相加
```

- 實際在執行時, C會將 5 與 3 * 4 視為是加法的兩個運算元, 其中 3 * 4 本身就是一個運算式。
- 每一個運算式都有一個運算結果, 以加法運算來說, 兩個運算元相加的結果就是加法運算式的運算結果。

甚麼是運算式？

- <優先權>
  - 在 C 語言中, 四則運算的運算次序和我們從小所學的先乘除、後加減完全相同, 而且在運算式當中, 也可以任意使用配對的小括號 "()", 明確表示計算的方式, 舉例來說：

甚麼是運算式？

```
03 int main()
04 {
05    int i,j;
06    i = (1 + 3) * 5 + 6;     // -> 4 * 5 + 6
07    j = 1 + 3 * (5 + 6);     // -> 1 + 3 * 11
08    std::cout << "變數 i 等於:" << i << '\n'
09              << "變數 j 等於:" << j;
10 }
```

甚麼是運算式？

- 其中第 6 與第 7 行的運算式如果將括號去除, 兩個運算式將變得一模一樣, 可是因為加上了括號, 所以兩個運算式的運算順序並不相同, 最後的結果也不一樣。

C的算術

- 在數值運算中, 最直覺的就 是

  +、-、*、/　四則運算,

除了 * (乘) / (除) 符號要熟悉之外, 運算式的寫法, 都只要依我們習慣的方式來撰寫即可。不過 在 C++中的數值由於有**整數**與**浮點數**的差異, 也使得 C++ 的四則運算結果, 不一定會和我們所習慣的相同。舉例來說, 我們都知道 3 除以 2 的結果是 1.5, 但在 C++ 程式中可不一定如此：

C的算術

- 3/2　　//計算結果等於1
- 3.0/2　　//計算結果等於1.5
- 程式都是用 3 除以 2, 為什麼得到的商卻不同？這是因為在 C++ 中, 如果運算式中的運算元**都是**整數 (int、long、short 等), 則運算結果也會是整數, 若有小數部份, 將會被**捨去**, 只保留整數的部份。因此第 5 行 (3/2) 的結果就是 1.5 去掉小數部份, 只剩 下 1 。

C的算術

- 如果運算結果有可能產生小數, 則至少要有一個運算元是浮點數型別計算結果才會保留小數部份。如上第 6 行程式就是將 3 寫成 3.0, 對 C來說, 這個字面常數因為有小數點所以會是 double 型別, 因此計算結果也會保留小數部份, 得到正確的商。

C的算術

- 加減運算子也可以用來表示正負數, 這也和平常使用的方式相同, 例如：

```
-5              // 負 5
3 * (-6)        // 結果爲負的 18
```

餘數運算子

- C++ 除了提供基本的四則運算外, 還有個特別的求餘數運算子：%。語法如下：

```
a % b
```

- 這個運算式的意思是用 a 除以 b, 並取其餘數, 所以：

```
語法
8 % 3   → 8 除以 3 的餘數爲 2，所以計算結果爲 2
9 % 3   → 9 可以被 3 整除，所以計算結果爲 0（沒有餘數）
```

餘數運算子

- 請注意, % 只能用在整數的資料型別, 若用在浮點數將會產生編譯錯誤：

```
3.5 % 3    // 不合法， 編譯會失敗
```

遞增遞減運算子

- 在設計程式時, 經常會需要將變數的內容遞增或是遞減 (加一或減一), 例如：

```
i = i + 1;          // 遞增, 將 i 加 1 後設爲 i 的新數值
j = j - 1;          // 遞減, 將 j 減 1 後設爲 j 的新數值
```

- 為方便做這類運算, C 提供了專用的運算子, 可以用來代替加減法運算子來將變數加減 1 。如果您需要幫變數加 1, 可以使用 ++ 遞增運算子；如果需要幫變數減 1, 則可以使用 -- 遞減運算子

遞增遞減運算子

```
03 int main()
04 {
05    int i = 100;
06    i++;     // 遞增
07    std::cout << "變數 i 的值爲：" << i;
08
09   i--;     // 遞減
10    std::cout << "\n變數 i 的值爲：" << i;
11 }
```

執行結果

```
變數 i 的值爲：101
變數 i 的值爲：100
```

遞增遞減運算子

- 遞增及遞減運算子除了可以寫在變數的後面 (稱為後置), 也可以寫在變數的前面 (稱為前置),
- 這兩類運算子雖然同樣會將變數做遞增或遞減的運算, 但它們所代表的意義並不相同,
- 請看這個範例：

遞增遞減運算子

```
03 int main()
04 {
05    int i = 100, j;
06    j = (i++) + 5;     // 後置遞增
07    std::cout << "i = " << i << "\t\tj = " << j;
08
09    i = 100;
10    j = (++i) + 5;     // 前置遞增
11    std::cout << "\ni = " << i << "\t\tj = " << j;
12 }
```

執行結果

```
i = 101        j = 105
i = 101        j = 106
```

比較運算子

- C 語言共有 6 個做為比較用的比較運算子 ，其運算結果只傳回 true (整數 1) 或 false (整數 0)。其使用法如下：

運算子	使用例	說明
>	i>j	比較 i 是否大於 j
<	i<j	比較 i 是否小於 j
==	i==j	比較 i 是否等於 j
>=	i>=j	比較 i 是否大於或等於 j
<=	i<=j	比較 i 是否小於或等於 j
!=	i!=j	比較 i 是否不等於 j

比較運算子

- 由於只做比較而不做設定，所以執行後各運算元的值不變。

- 由於 << 的優先順序較高，所以比較運算子的運算要用小括弧括起來。注意，’==’ 和 ’=’ 不可搞混，前者是做為比較之用，而後者則是用來指定值之用。

比較運算子

- Int i=3, j=3;
- i==j;
- i>j;
- ++i>j;
- J--<3;
- i!=j;

邏輯運算子

- C 提供 3 個邏輯運算子 ,邏輯運算子是取運算元的布林值來參與運算, 運算結果也只傳回 true (整數 1) 或 false (整數 0)。其使用法如下表所示：

運算子	使用例	說明
!	!i	對 i 做邏輯的 NOT
&&	i && j	i 與 j 做邏輯的 AND
||	i || j	i 與 j 做邏輯的 OR

邏輯運算子

- 範例：
- i=3,　　b=0;
- i &&b
- i || b
- I &&!b
- I ||!b

位元運算子

- C也能對資料做位元層級的處理, 它總共提供了 6 種位元運算子, 大大的提高了低階處理能力。位元運算子會把資料看成是位元 (bit) 的集合, 其運算方式也是 1 個位元、1 個位元地處理：

位元運算子

運算子	使用例	說明
>>	i >> j	把 i 的位元右移 j 個位元
<<	i << j	把 i 的位元左移 j 個位元
~	~i	把 i 的每一位元反相 (0 變 1、1 變 0)
|	i|j	i 與 j 對應的位元做 OR
&	i&j	i 與 j 對應的位元做 AND
^	i^j	i 與 j 對應的位元做 XOR

位元運算子

程式 Ch04-10.cpp 各種位元運算的情形

```
01 #include<iostream>
02 using namespace std;
03
04 int main()
05 {
06    short i = 23, j = 14;
07
08    cout << "i >> 1 = " << (i >> 1) << endl;
09    cout << "i << 3 = " << (i << 3) << endl;
10    cout << "~i = " << (~i) << '\n' << endl;
11
12    cout << "i = " << i << "\tj =" << j << endl;
13
```

位元運算子

```
14      cout << "i & j = " << (i & j) << endl;
15      cout << "i | j = " << (i | j) << endl;
16      cout << "i ^ j = " << (i ^ j) << endl;
17  }
```

執行結果

```
i >> 1 = 11
i << 3 = 184
~i = -24

i = 23   j =14
i & j = 6
i | j = 31
i ^ j = 25
```