第一節　基礎 C++ 技巧

本章目標

- 了解 C++ 的語法皆爲「**運算**」。
- 熟悉各運算子的用法及特性。
- 注意**未定義行爲**。

一、 程式架構

(一) 基本 C++ 架構

最基礎的 C++ 架構如下：

```cpp
#include <iostream>
using namespace std;
int main() {
}
```

程式碼 1: C++ 基本架構

怎麼理解呢？不需要理解，我們先**記起來**。基本上程式的內容都寫在**大括號**中。裡面每個符號都要一樣 (分號也是)。

接下來要講一個程式最基本的兩個操作：**輸入**和**輸出**。

(二) 輸出

C++ 的**輸出符號**寫爲「cout」，你要輸出的東西用「<<」串連。試試看在剛剛的大括號中打上「cout << 1;」，會發生什麼事呢？

若程式只有一閃即逝的畫面，那麼可以在更下一行加上「system("PAUSE");」後觀察看看。在此，system("PAUSE"); 代表「暫停」的意思。程式 2 因爲沒加上這行，程式就會直接執行結束，加上這行，程式會在這裡「等你」。

```cpp
1 #include <iostream>
2 using namespace std;
3 int main() {
4   cout << 1;
5 }
```

程式碼 2: 還不清楚的人，這裡是剛剛操作的範例程式碼

接下來有一些 C++ 的特性你必須知道：

1. 如果將 2 中第 4 行改成「cout << 1」(去掉分號) 會發生什麼結果？因爲「分號」對 C++ 而言代表「**一個句子的結束**」，因此當一行**指令結束就要加分號**。
2. << 可以串很多東西一起輸出，試試看「cout << 1 << 2;」，和你所想的有何不同？
3. 那麼「cout << 1 << "␣"<< 2;」呢？**注意**！"␣" 是雙引號中間夾著一個「空白」。

換行符號 cout 中「endl」代表**換行**符號，輸出時很好用，以下情況可以練習看看：

1. 試試看「cout << 1 << 2 << endl;」，和「cout << 1 << 2;」有什麼不同呢？
2. 如果看不出來，試試看「cout << 1 << endl << 2;」。

(三) 變數

變數和數學「變數」的概念不太一樣，程式的變數像是「**容器**」，可以裝資料。C++ 裡，每個容器都要先講好兩件事：

- 名稱
- 用途

此時這個步驟叫做「**宣告**」，宣告變數的語法如下：

```cpp
int x;
```

程式碼 3: 宣告變數

程式碼 3 中，我們宣告一個變數，**名稱**叫做 x，用途叫做「int」，代表的意義是「**整數**」，規定變數 x **只能裝整數**，如圖 1。

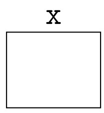

圖 1: 容器

變數的用途就是爲了把數字裝到變數中，

```
#include <iostream>
using namespace std;
int main() {
  int x;                   // 宣告變數 x
  x = 5;                   // 把整數 5 裝進 x 裡面
  cout << x << endl;  // 印出變數 x 存的值
}
```

程式碼 4: 變數的用途

宣告變數的種類除了int (即整數) 之外，還有其他不同的種類，以後會慢慢介紹。此外，程式碼 4 中 `x = 5;` 這行**不要**和數學中的「等於」搞混。

練習看看，若把程式碼 4 的「`x = 5;`」改成以下狀況，會出現什麼事？該怎麼解釋這些現象呢？

- `x = 5.0;`
- `x = 0.5;`
- `5 = x;`

這些練習目的是要讓你**眞正了解**問題出現時的現象，了解出問題的原因才有辦法 debug，爲什麼會出現這些現象我們繼續下去就知道了。

多變數宣告　宣告兩個整數可以寫成像程式碼 5a，更可以簡化成如程式碼 5b。

以此類推，宣告三個整數也是如法炮製，如程式碼 6：

```cpp
int a;
int b;
```

(a) 兩個宣告

```cpp
int a, b;
```

(b) 簡化版

程式碼 5: 宣告兩個變數

```cpp
int a, b, c;
```

程式碼 6: 宣告三個變數

變數初始化 剛剛我們説明變數就像是容器，作用就是**裝東西**，那如果容器不塞東西會發生什麼事呢？比如程式碼 7。

```cpp
#include <iostream>
using namespace std;
int main() {
  int x;                // 宣告變數 x
  cout << x << endl; // 印出變數 x 存的值
}
```

程式碼 7: 變數不初始化，會發生什麼事呢？

程式碼 7 可以多試幾次，一般來説，C++ 中，所有變數都要自己去**初始化**。例如：`x = 5;`，把整數 5 丟給 x 等等。因爲沒有初始化過的變數，裡面裝的資料是**不確定**的。

或許你很幸運看到 x 都是 0，但那只是**恰巧**而已，因此有時候程式有 bug 時，不妨檢查一下是否存在這個原因。

x

$\boxed{?}$

圖 2: 沒有被初始化的變數

要解決變數沒有初始化的問題，有兩個常用的方法，原則上都是**賦值**，方法會在稍後提到。

(四) 輸入

執行以下程式會發生什麼事呢？

```cpp
#include <iostream>
using namespace std;
int main() {
  int x;
  cin >> x;
  cout << x << endl;
}
```

程式碼 8: 輸入

大家可以試著執行看程式碼 8，如果沒發生什麼事，試著輸入「1」再按 enter 鍵，會發生什麼事呢？

和輸出相對，「cin」代表輸入符號，可以輸入後面變數的資料。輸入的資料和我們宣告變數的**型態**有關，在此例中，x 是整數，因此可以**輸入一個整數**。

注意！ cin 的 >> 不要和 cout 的 << 搞混。以下練習讀者們可以試試看會發生什麼事，重點在**觀察產生的現象**：

- 輸入「5.0」再按 enter 鍵呢？
- 輸入「0.5」再按 enter 鍵呢？
- 輸入「XD」再按 enter 鍵呢？

多變數輸入　多變數輸入和輸出類似：

```cpp
int x, y;
cin >> x >> y;
```

程式碼 9: 輸入多變數

唯一要注意的一點是，有些題目會要我們輸入「換行」相隔的數，我們不需要在輸入中加入「endl」，否則程式容易出錯。

(五) 資料型態

既然有裝整數的容器，那麼當然也可以宣告裝「小數點」的容器啦！這些不同用途的容器我們稱爲「**資料型態**」。表 1 代表 C++ 常用的資料型態，詳細內容之後再介紹，先來用看看這些東西。

關鍵字	意義	備註
`bool`	布林值	只有 `true` 和 `false`
`int`	整數	
`long long`	長整數	存比較大的整數，以後會介紹
`double`	**浮點數**	也就是小數點

表 1: 資料型態

布林值　布林值是一種資料型態，只來裝兩種數值：「`true`」和「`false`」，宣告方法和 `int` 類似，如程式碼 10：

```
bool b;
```

程式碼 10: 布林值宣告

值得注意的是，兩個不同資料型態不能同時宣告在同一行，如程式碼 11。其實在 C++ 當中，逗號有**特殊意義**，不要想成一般的「逗號」。

```
int a, bool b;
```

程式碼 11: 不同的宣告不能用「逗號」隔開

賦值　將一個「數值」裝進一個變數中，稱爲**賦值**。例如，程式碼 12 把整數 5 裝進整數變數 x 中：

```
int x;
x = 5;
```

程式碼 12: 賦值

```cpp
int x = 5;
```

程式碼 13: 賦值簡化

當然，我們每次如果一行宣告，一行賦值也太麻煩，因此有簡化的寫法，變數宣告和賦值可以寫在一起：

練習 下面有一段程式碼：

```cpp
bool b;
cout << b << endl;
```

對程式碼的 b 做以下賦值，會發生什麼事？

- b = true;
- b = false;
- b = 2;
- b = 0;
- b = -1;

布林值的重要觀念 C++ 中，「**非零整數**」會被當做「true」，印出時也會印出一個非零整數 (**通常是 1**)。「0」會被當做「false」，印出時會印出「**0**」。

這個特性在之後會**非常**常用！大家要注意！

整數 int 和 long long 都是存整數的資料型態，這裡先跳過 long long 和 int 的差別，先知道 long long 也是存整數就好。(謎之音：「那幹嘛現在說= =」)

整數常數有一些比較特別的賦值方法，可以試著執行程式碼 14，看看和預期的有什麼不同。

```cpp
cout << 012 + 1 << endl;
```

程式碼 14: 會印出多少？

整數賦值可以用八進位和十六進位等用法，可以看以下範例：

- 012 是八進位，開頭是 0
- 0xFF 是十六進位，開頭是 0x
- 有時候宣告常數也可以指定型態
  - 1234567U 在尾巴加上 U 代表 unsigned (之後說明)
  - 1234567LL 尾巴加上 LL 代表 long long

浮點數 接著來講一下浮點數，浮點數也就是存**小數點**的資料型態，宣告方法如下：

```
double d;
```

程式碼 15: 浮點數宣告

賦值和前面都一樣，不同的是浮點數有一些特別的表示法：

- 如果要把 1.0 賦值給 d $\Rightarrow$ d = 1.0;，這是最基本的賦值
- 稍微有變化一點，如果是 0.5 的話，可以簡化爲 .5 $\Rightarrow$ d = .5;
- 接著是**科學記號**
  - 18.23e5 代表 18.23×10^5
  - 5.14e-6 代表 5.14×10^{-6}

二、 算術運算子

(一) 運算性質

算術運算子有以下五個：

如果不管**運算順序**和**結合性**，一般來說可以用五則運算來理解，只不過程式跟數學還是有差距，舉個例子：$1+2+3$ 會是多少？

這個問題很顯然答案會是 6，但是程式爲什麼會計算出答案呢？我們先建立起**二元運算**的觀念：

定義 1.1. **二元運算由一個運算子和兩個運算元**構成，例如：$1+2$：「+」稱爲「運算子」，「1」和「2」稱爲運算元 *(我們常稱爲「被加數」和「加數」)*。

算術運算子	意義	運算順序	結合性
+	加法	6	左→右
-	減法	6	左→右
*	乘法	5	左→右
/	除法	5	左→右
%	取餘數	5	左→右

表 2: 算術運算子

(二) 結合性與運算順序

我們可以知道「加減乘除餘」都是二元運算，因此，我們回到原來的問題：$1+2+3$ 到底是先算 $1+2$、還是先算 $2+3$ 呢？

這時我們就會出現大麻煩了！儘管在這裡先算後算是沒有太大的問題，但是在 $1-2-3$ 的情況下，先算 $1-2$、還是 $2-3$ 這個問題就變成此時需要解決的問題。

計算機普遍採用的解法就是「決定運算的**方向**」。例如：

- 先算 $1+2=3$，再算 $\mathbf{3}+3=6$
- 先算 $2+3=5$，再算 $1+\mathbf{5}=6$

決定運算方向對「計算機」而言**意義重大**！同樣的想法可套進剛剛的 $1-2-3$ 中：

- 我們直觀上會先算 $1-2=-1$，再算 $-1-3=-4$。
- 因此 C++ 在設計上也會把加減乘除餘的**結合性**「設定」成**從左到右算**。

我們回頭看表 2，可以看出在結合性那一欄定義了每個運算子的運算順序。

接著我們處理更複雜的問題——四則運算：$1+2*3-4$。同樣地，我們的運算規則是「**先乘除餘，後加減**」，因此 C++ 發展出一套類似的規則，稱做**運算順序**。

- 運算順序小的優先運算，在表 2 中 C++ 定義了每個運算子的**優先權**
- 若運算順序相同，則依照運算方向做計算。

因此我們知道整個運算式的運算順序如下：

$1+\mathbf{2*3}-4$	$*$ 的運算順序最高
$=\mathbf{1+6}-4$	加法和減法運算順序相同，依照結合性從左到右算
$=\mathbf{7\text{-}4}$	依照結合性從左到右算
$=3$	

C++ 的四則運算用**優先順序**和**結合性**來處理，這件事情非常重要，稍後就會知道爲什麼。

(三) 整數除法與除零問題

整數除法　以下程式碼可能會讓你感到驚奇：

- `cout << 8 / 5 << endl;` 的結果？**Ans: 1**
- `cout << 8.0 / 5.0 << endl;` 的結果？**Ans: 1.6**

其原因出在於，在 `8 / 5` 中，8 和 5 被視爲 `int`，因此 C++ 會做「**整數除法**」；而在 `8.0 / 5.0` 中，8.0 和 5.0 被視爲浮點數 `double`，因此會做「**浮點數除法**」。

除以零　除法還有另外一個問題點，那就是**除以零**，我們知道數學上是不能除以零的，那程式呢？下面的狀況讀者們也請多做嘗試，看看會發生什麼結果。

- `cout << 1 / 0 << endl;`
- `cout << 0 / 0 << endl;`
- `cout << 1.0 / 0.0 << endl;`
- `cout << 0.0 / 0.0 << endl;`

註：有些 IDE 如 Visual C++ 會直接擋住除以零，不讓你編譯，如果無法編譯成功，那麼就嘗試「繞過」他，例如：宣告一個變數，把分母裝 0 進去再試試看。

注意：通常上面的程式碼在編譯時可以過，但是在執行時會出些狀況，各位知道出了哪些狀況就好，不用了解太詳細。

(四) 應用：取餘數

C++ 的 % 運算子會有跟我們想像中不太一樣的現象，首先我們可以觀察一下 C++ 怎麼做的：

- `cout << 5 % 3 << endl;` 會輸出什麼？**Ans:2**
- `cout << (-5) % 3 << endl;` 呢？**Ans:-2**

大部分的人會認爲，% 就是「取餘數」，但事實上並不完全是這樣子，如果在下面 -2 的例子，應該結果是要 1 才對，這也是 C++ 一個**奇怪的特性**。

解決方法？要怎麼做出取餘數的效果呢？以下提供一個解法：

1. 假設 n 要 mod m ...
2. 首先，我們取 `n % m`
   - 如果 $n \geq 0$，會得到介於 0 到 $m-1$ 的數字
   - 如果 $n < 0$，會得到介於 $-(m-1)$ 到 0 的數字
3. 接著加上 m，變成 `n % m + m`
   - 如果 $n \geq 0$，會得到介於 m 到 $2m-1$ 的數字
   - 如果 $n < 0$，會得到介於 $-(m-1)+m=1$ 到 m 的數字
   - 全都修成正值了！**但還差最後一步 ...**
4. 最後，再 mod m 一次，把所有數字修正回 0 到 $m-1$ 之間。
   - 大功告成啦！`(n % m + m) % m`

練習題

✓ ***UVa 10071: Back to High School Physics***
這題只要能夠讀懂題意就不難寫。如果不知道怎樣讀取多筆測資請先參考迴圈部分 (EOF 版)。

✓ ***UVa 10300: Ecological Premium***
一樣能讀懂題意就不難寫。

✓ ***UVa 11547: Automatic Answer***
計算 $(n \times 567 \div 9 + 7492) \times 235 \div 47 - 498$

三、 比較和邏輯運算子

(一) 比較運算子

比較運算子如表 ??：

比較運算子	意義	運算順序	結合性
==	等於	9	左→右
!=	不等於	9	左→右
>	大於	8	左→右
<	小於	8	左→右
>=	不小於	8	左→右
<=	不大於	8	左→右

表 3: 比較運算子

和數學的大於小於概念類似，只是要**注意**！C++ 的等於寫作「==」，不要和賦值的「=」搞混。

回傳值 C++ 程式當中有回傳值的概念，舉例來說：`cout << (3 < 5) << endl;` 這一行會發生什麼事呢？要解釋這一段程式有點複雜，我們慢慢講起。

比較運算子也算是一種**二元運算**，他會比較兩邊數字大小，如果是正確的，則爲 `true`、否則就是 `false`。這種概念我們稱爲「**回傳值**」。

回傳值也會有資料型態，由此可見比較運算子的回傳值是布林值 `bool`，例如 `3 < 5` 的回傳值就是 `true`。

但是還沒完，因爲我們發現剛剛那一行程式碼不是印出 `true`，怎麼回事呢？根據 C++ 的規則，`true` 通常會當作**非零**，因此會印出一個非零的數字 (通常是 1)；反之，如果是 `false`，就會當作是 0。以此出發，這會延伸到之後有很多技巧。

運算簡化 例如，判斷**不**整除直觀來想就是「檢查 n 取 m 的餘數是否非零」，我們利用前面學到的比較運算子和算術運算子可以得出 `n % m != 0`。

但是這一個判斷還可以進一步簡化，如果 `n % m` 結果不是零，如果在條件判斷時會被當作 `true`，否則就被當作 `false`，因此很多時候就只要簡寫成 `n % m` 就可以了。

	n % m != 0	n % m
當 n % m 不爲零	true	true
當 n % m 爲零	false	false

表 4: 眞值表

簡化的寫法大多時候可以取代原來一般寫法，且通常比較運算子要和 if、else 配合，之後會介紹這兩個東西。

(二) 邏輯運算子

邏輯運算子有以下三個：

邏輯運算子	意義	運算順序	結合性
&&	且	13	左→右
||	或	14	左→右
!	非	3	**右→左**

表 5: 邏輯運算子

邏輯運算子一般來說是連接比較運算子，例如：1 < x && x < 5。

舉個大家容易誤解的例子，如果要判斷 x 是否介於 a 和 b 之間能不能寫成 a <= x <= b; 呢？答案是不行。乍看之下似乎符合數學運算式，但是讀者必須注意，這裡是 C++，因此我們需要用 C++ 的觀念去切入這個問題。

我們可以採用回傳值的觀點，從表 ?? 可以知道，<= 運算子在列出很多個時，會**由左到右算**，因此在左側的 a <= x 會先算出 true 或者是 false。

假設 a=-4、b=-1、x=-2，我們**預期**結果是 true，接著分析 C++ 會怎麼處理 a <= x <= b。

- C++ 會先計算 a <= x 得到 true
- 接著計算 true <= b
- 我們知道 true 通常是 1
- a <= x <= b 的回傳值就會是 false

反過來，a <= x 是 false 的狀況也會有同樣的問題。

如果我們要解決此狀況，那麼就勢必要用邏輯運算子：a <= x && x <= b。這個觀念常常是剛上手 C++ 的人常常踩到的誤區，可以多注意。

我們先前是對「判斷不整除」進行簡化，那我們要怎麼簡化「判斷整除」呢？n % m == 0 可以用「! 運算子」變成 !(n % m != 0)，接著使用剛剛的簡化規則，最後變成 !(n % m)。

(三) 短路運算

C++ 的邏輯運算屬於「**短路運算**」，當我們在計算一個判斷式時，如果我們已經可以確認其結果，之後的判斷就**不會再進行**。以下講述 && 運算子和 || 運算子的行爲：

- A && B：實際上當 A 是 false，也就是確定整個運算式**必爲** false，則程式會**跳過** B，下面程式碼可以試試看：

```
int i, j;
i = j = 0;
if ((i++ < 0) && (j++ > 0))
  cout << "XD" << endl; // 這行不會輸出
cout << i << "␣" << j << endl; // i 爲 1，j 爲 0
```

程式碼 16: 範例

- A || B：只要 A 是 true，也就是確定整個運算式**必爲** true，則程式會**跳過** B

```
int i, j;
i = j = 0;
if ((i++ >= 0) || (j++ < 0))
  cout << "XD" << endl; // 會輸出 XD
cout << i << "␣" << j << endl; // i 爲 1，j 爲 0
```

程式碼 17: 範例

練習題

- ✓ ***UVa 10055: Hashmat the brave warrior***
  取絕對值有兩種做法，一種是用 `if` 判斷；另一種是呼叫函數 `abs()` 就好了。`abs()` 函數被定義在 `<cstdlib>` 中，雖然沒有 `#include` 在 Visual C++ 依然能編譯過，但是上傳時因爲編譯器的原因會導致**編譯錯誤** (Compilation Error, CE)。
  另外要注意這一題的整數型態需用 `long long`，用 `int` 會造成「溢位現象」，這個原因會在後面說明。

- ✓ ***UVa 11172: Relational Operators***
  能夠理解題意就不難解決此問題。

- ✓ ***UVa 11942: Lumberjack Sequencing***
  依序給你一些鬍子的長度，問你這些鬍子是不是由長到短，或是由短到長排列。

四、 位元運算子

(一) `int` 和 `long long` 的儲存形式

在此節我們要講位元運算子，但在一開始我們要先了解 `int` 在電腦當中怎麼儲存的，因此要先介紹一些觀念。

- **位元** (bit, b)：計算機儲存資料的基本單位，只儲存 **0 和 1**
- **位元組** (byte, B)：因爲位元很多，所以我們習慣上把 8 個位元「打包起來」，變成一個位元組。

01001010

表 6: 位元組

- 常見應用
  - KB、MB、GB、TB、PB：資料大小
  - Kbps、Mbps、Gbps：資料傳輸速度

以 `int` 來說，他至少使用 **2 個位元組**來紀錄資料，有些讀者可能會有疑問說：「不是都 4 個位元組嘛？」其實當初定義時，`int` 只有**定義**成「至少」2 個位元組，只是現在的電腦大多是 4 個位元組。

型態	長度
`bool`	1 位元組
`int`	2 或 **4** 位元組
`long long`	4 或 **8** 位元組
`double`	8 位元組

表 7: 位元組長度

表 ?? 標示每個資料型態使用多少位元組來紀錄資料，在 `int` 和 `long long` 部分，用粗體來表示現在大部分的機器所使用的位元組數。以下討論就使用 `int` 為 4 個位元組、`long long` 為 8 個位元組，不再贅述。

`int` 表示法 一般來說，`int` 由 4 個位元組組成

10100010	00110011	00100111	10101101

表 8: `int` 的位元組

可以視為一個長度是 32 的二進位數字，我們將位數依照高低編號，如表 ??，x_{31} 表示正負號，若 x_{31} 為 0 代表此 `int` 是正數，反之則為負數。

$x_{31}x_{30}\cdots x_{24}$	$x_{23}x_{22}\cdots x_{16}$	$x_{15}x_{14}\cdots x_{8}$	$x_{7}x_{6}\cdots x_{0}$

表 9: 二進位 `int`

因為 `int` 的儲存方式很特別，要多花一些力氣說明。

`int` 存正數的情況 當 `int` 儲存正數時，是依照一般二進位方式儲存。例如 `int x = 1;`

00000000	00000000	00000000	00000001

表 10: 1 的二進位表示法

當 `int x = 255;` 時如表 ??。

`int` 存負數的情況 上面情況都不難，比較有意思的是當它存負數時，怎麼表示呢？例如下面的 `int x = -1;`：

00000000	00000000	00000000	11111111

表 11: 255 的二進位表示法

11111111	11111111	11111111	11111111

表 12: 存 -1 的情況

要理解負數的儲存方法 (謎之音：「根本黑魔法！」)，我們嘗試看看 (-1)+1，我們知道 $(-1)+1=0$，那麼以這種表示法相加的結果是：

	11111111	11111111	11111111	11111111
+	00000000	00000000	00000000	00000001
	100000000	00000000	00000000	00000000

紅色的 1 因爲超過 32 位元，所以被捨棄，稱爲**溢位**。

這種表示法稱爲**二補數 (2's complement)**，好處是減法和加法只需要用溢位的方式就可以處理掉，這在底層硬體實作上帶來許多方便，缺點當然是不好理解負數的儲存方法。要想像負數 -x 的表示法，訣竅是 (-x)+x 會因爲溢位而等於 0。

大致上來說，最特別的兩個數，一個是 0，在此表示法中會是**全 0**；而 -1 會是**全 1**，這兩個數字在很多時候會很好用，可以稍微記得這個結論。

以下練習看看：

- `int x = -2;`
- `int x = -256;`

(二) 位元運算子

位元運算子是大家比較難理解的運算子，但是在效能優化上，或是在一些特殊的題目時是很有用的，以下分別講述這些運算子的功用與概念。

左移和右移運算子 左移運算子和右移運算子代表在位元操作上左移和右移 k 個位元，但注意不要和 cin 與 cout 的 <<、>> 混淆。

舉例來說，2 << 2 $\Rightarrow$ 8 即是把 2 在二進位的位元往左移 2 格：

位元運算子	意義	運算順序	結合性
`<<`	左移運算子	7	左→右
`>>`	右移運算子	7	左→右
&	位元 and	10	左→右
^	位元 xor	11	左→右
|	位元 or	12	左→右
~	1's 補數	3	**右→左**

表 13: 位元運算子

00000000	00000000	00000000	00000010

$\Downarrow$

00000000	00000000	00000000	00001000

表 14: 左移的情況

類似的情況，`5 >> 1` $\Rightarrow$ `2` 把 5 在二進位的位元往右移一格，最右邊多餘的 1 會被捨棄：

00000000	00000000	00000000	00000101

$\Downarrow$

00000000	00000000	00000000	00000010

表 15: 右移的情況

不管是左移還是右移，移出去的位元會被捨棄，這也是溢位的一種，但在左移右移會影響到 x_{31} 時會比較複雜，因爲我們知道 x_{31} 決定正負號，以下例子讀者們可以試試看，應該會出乎意料之外：

- `2147483647 << 1`
- `-5 >> 1`
- `(2147483647 << 1) >> 1`

那左移和右移運算子有什麼應用呢？我們觀察一下 `a << k` 會得到什麼數字呢？ 那 `a >> k` 呢？

一般來說 a << k 會得到 $a \times 2^k$，a >> k 會得到 $a/2^k$，有些情況比較複雜，大家看看就好，起碼對這些運算「有感覺」。

and、xor、or **運算子** 對於兩個位元 x 和 y，遵守以下運算規則：

&	**1**	**0**
1	1	0
0	0	0

(a) and 運算子

^	**1**	**0**
1	0	1
0	1	0

(b) xor 運算子

|	**1**	**0**
1	1	1
0	1	0

(c) or 運算子

表 16: 三種運算子

and、or 運算子類似之前的邏輯運算子，不同在於這是**位元**運算，是對每一個位元做運算。另外，xor 運算很特別，規則簡單來說就是不同數字為 1，相同為 0。

以下是 int 做位元運算，很多人容易將位元運算子與邏輯運算子搞混，於是我們來看看 5 和 3 做位元運算會發生什麼事：

5 & 3 結果會是 1：

	00000000	00000000	00000000	00000101
&	00000000	00000000	00000000	00000011
	00000000	00000000	00000000	00000001

表 17: 5 & 3 的狀況

5 | 3 結果會是 7：

	00000000	00000000	00000000	00000101
|	00000000	00000000	00000000	00000011
	00000000	00000000	00000000	00000111

表 18: 5 | 3 的狀況

5 ^ 3 結果會是 6：

	00000000	00000000	00000000	00000101
^	00000000	00000000	00000000	00000011
	00000000	00000000	00000000	00000110

表 19: 5 ^ 3 的狀況

~	**1**	**0**
	0	1

表 20: 補數運算子

補數運算子 對於兩個位元 x 和 y，遵守以下運算規則：

簡單來説就是「1 變 0，0 變 1」 (相當於邏輯運算子的 !)，又稱爲 **1's 補數**。例如： ~0 ⇒ -1。

(三) 一元運算子

和二元運算子類似，**一元運算子**就是只有一個運算元的運算子。

運算子	意義	運算順序	結合性
+	正號	3	右→左
-	負號	3	右→左

表 21: 一元運算子

他們的運算順序都是**從右到左**，例如 ~~3 會先算右邊的 ~3，得到 -4，接著 -4 再和左邊的補數運算子「運算」，回傳結果爲 3。

(四) 常用技巧：連續的 1

位元運算最常見的問題之一，那就是：要怎樣產生 2 進位下連續 k 個 1？例如：

- 3 個 1

00000000	00000000	00000000	00000**111**

- 5 個 1

00000000	00000000	00000000	000**11111**

可以很容易發現，k 個 1 恰好是 2^k-1。**只要不牽扯到正負號** x_{31} 的情況下，可以很容易地寫成 `(1 << k) - 1`，但要注意減號和左移運算子的優先順序。

加強版　當然，這個結論可以繼續推廣：要怎樣產生 2 進位下 x_a 到 x_b 都是 1？(假設 $a<b$) 例如：

- x_0 到 x_2，此時恰好是 3 個 1 的情形

00000000	00000000	00000000	00000**111**

- x_3 到 x_7

00000000	00000000	00000000	**11111**000

觀察之後，可以發現是 $2^{b+1}-2^a$。該怎麼實作就從之前取 k 個 1 的方法去擴展就可以得到。

取負數　另外來講一個特別的例子，它可以幫助你判斷負數的儲存方法給。你一個正數 `x`，問如何**不用負號**的情況下求出 `-x` 呢？比較 `-x` 和 `~x` 的不同，就會發現，他們事實上只差 1。例如：

- `~0` ⇒ `-1`
- `~123` ⇒ `-124`

從上面的結論可以歸納出 `-x` 恰好是 `(~x)+1`。

(五) 常用技巧：遮罩與指定位元

這裡要講述我們先前學會產生連續 1 的用途，有時候我們想要對位元做一些事情，例如：

- 知道某些位元的值
- 改變某些位元

以下就分別講述位元運算要怎樣做到這些技巧。

位元運算的性質 從表 ?? 可以看到這些位元運算的規則，但我們可以換個角度來發掘他更多的特性，假設其中一個位元是未知的，叫做 x (可能是 0 或 1)，那麼根據 ?? 和 ?? 的規則會如下：

表 22: 有未知數的位元運算

&	**x**
1	x
0	0

(a) and 運算子

|	**x**
1	1
0	x

(b) or 運算子

可以看出，當 x 是變數時，x & 0 永遠是 0，x & 1 永遠是 x；同樣地，x | 1 永遠是 1，x | 0 永遠是 x。根據這些性質可以得到對於**一個位元**，我們如何利用位元運算來操作：

- 知道一個位元的值：使用 x & 1 或是 x | 0
- 改變一個位元的值：
  - x & 0 把該位元設爲 0
  - x | 1 把該位元設爲 1

常用技巧：遮罩 根據剛剛位元運算的性質，我們拓展到 int 上，可以知道 x_0 是 1 還是 0：

	$x_{31}x_{30}\cdots x_{24}$	$x_{23}x_{22}\cdots x_{16}$	$x_{15}x_{14}\cdots x_8$	$x_7x_6\cdots x_0$
&	00000000	00000000	00000000	00000001
	00000000	00000000	00000000	0000000x_0

表 23: 取得 x_0

如果我們要

- 知道 x_i 是 1 還是 0 要怎麼做？
- 取出 x_a 到 x_b 的位元，要怎麼做呢？

常用技巧：指定位元 要如何把一個整數 x 當中，x_a 的位元「變成」1？我們可以從剛剛的概念繼續推廣，發現將 x_0 改爲 1 同樣使用 x | 1：

	$x_{31}x_{30}\cdots x_{24}$	$x_{23}x_{22}\cdots x_{16}$	$x_{15}x_{14}\cdots x_{8}$	$x_{7}x_{6}\cdots x_{0}$
|	00000000	00000000	00000000	00000001
	$x_{31}x_{30}\cdots x_{24}$	$x_{23}x_{22}\cdots x_{16}$	$x_{15}x_{14}\cdots x_{8}$	$x_{7}x_{6}\cdots x_{1}1$

表 24: 將 x_0 改爲 1

根據表 ??，可以發現 x_1 到 x_{31} or 0 都會是原來的值，但是 x_0 和 1 or 起來會是 1，如此一來就可以將 x_0 **強制** 設爲 1 而不改變其他位元。

同樣的狀況，利用我們在產生連續 1 的技巧，我們可以設定特定位元、連續位元爲 1，例如：將 x_2 改爲 1。

	$x_{31}x_{30}\cdots x_{24}$	$x_{23}x_{22}\cdots x_{16}$	$x_{15}x_{14}\cdots x_{8}$	$x_{7}\cdots x_{3}x_{2}x_{1}x_{0}$
|	00000000	00000000	00000000	00000100
	$x_{31}x_{30}\cdots x_{24}$	$x_{23}x_{22}\cdots x_{16}$	$x_{15}x_{14}\cdots x_{8}$	$x_{7}\cdots x_{3}1x_{1}x_{0}$

表 25: 將 x_2 改爲 1

另一個問題，要如何把一個整數 x 當中，x_a 的位元「變成」0？同樣也是利用表 ?? 的特性：任何位元和 0 and 起來恆爲 0。

	$x_{31}x_{30}\cdots x_{24}$	$x_{23}x_{22}\cdots x_{16}$	$x_{15}x_{14}\cdots x_{8}$	$x_{7}x_{6}\cdots x_{0}$
&	11111111	11111111	11111111	11111110
	$x_{31}x_{30}\cdots x_{24}$	$x_{23}x_{22}\cdots x_{16}$	$x_{15}x_{14}\cdots x_{8}$	$x_{7}x_{6}\cdots x_{1}0$

表 26: 將 x_0 設爲 0

但是比較不同的是，用 & 運算子時，其他的位元爲了要保持不變，需要用 1 來 and，此時常數會變得比較難以直接求出，建議就是以「補數」的觀念來做出此常數，表 ?? 可以寫爲程式碼 ??。

```
x = x & (~1);
```

程式碼 18: 將 x_0 設爲 0

位元技巧：取 2^k 餘數 當我們取 2 的餘數時，我們可以發現一個規律，因爲餘數只有 0、1 兩種，恰好是看 x_0，我們就可以把 x % 2 換成 x & 1。

取 4 的餘數時，餘數只有 0 (00)、1 (01)、2 (10)、3 (11) 四種，恰好是看 x_1x_0。因此可以知道 `x % 4` 可轉寫爲 `x & 3`，更一般性來說，我們利用連續 1 的寫法寫成 `x & ((1 << 2) - 1)`。

以此類推，求 2^k 的餘數就可以寫成 `x & ((1 << k) - 1)`，這種寫法有許多優點，如：

- 和「%」相比速度較快，% 運算子需要實際做除法，比較消耗時間。位元運算通常比較快，因此可以快速取餘數。
- 在負數下也沒有問題，例如：`(-1) & 3` 可以得到餘數爲 3，沒有 % 運算子的問題。

當然，也有一些缺點：

- 不易閱讀。
- 只能取特定餘數。
- 要注意運算順序！

(六) 應用：Parity

Parity 問題：給妳一個正整數 x，問在二進位下有幾個 1？以下有幾個範例：

- PARITY(5) 如表 ??，可以看出二進位下有兩個 1，因此結果爲 2。

00000000	00000000	00000000	00000101

表 27: 5 的 parity

- PARITY(255) 如表 ??，結果爲 8。

00000000	00000000	00000000	11111111

表 28: 255 的 parity

普通的 Parity 算法，就是利用他的定義，一個位元一個位元慢慢算：

如果我們仔細觀察，可以看出有些東西我們可以用剛剛的概念來替換：

```c
for (cnt = 0; x; x /= 2) {
  if (x % 2 != 0)
    cnt++;
}
```

程式碼 19: Parity 普通寫法

```c
for (cnt = 0; x; x >>= 1) { // 右移代替除法
  if (x & 1) // 省略「!= 0」，同時把除法改成位元運算
    cnt++;
}
```

程式碼 20: Parity 位元運算寫法

```c
unsigned int v; // 32-bit word
v ^= v >> 1;
v ^= v >> 2;
v = (v & 0x11111111U) * 0x11111111U;
(v >> 28) & 1;
```

程式碼 21: Parity 究極寫法

以下是檢查 Parity 是否爲奇數的程式碼看看就好，至於其中的細節讀者們可以從位元的觀念下去思考得到：

看看就好，不要刻意去記這些炫砲技能。

(七) 應用：xor 性質

還記得 xor 嗎？這個運算是這幾個當中最讓人陌生的一個，回顧表 ?? 可以知道 xor 運算的性質是「同爲 0 或同爲 1 xor 起來就是 0」。

	$x_{31}x_{30}\cdots x_{24}$	$x_{23}x_{22}\cdots x_{16}$	$x_{15}x_{14}\cdots x_{8}$	$x_{7}x_{6}\cdots x_{0}$
^	$x_{31}x_{30}\cdots x_{24}$	$x_{23}x_{22}\cdots x_{16}$	$x_{15}x_{14}\cdots x_{8}$	$x_{7}x_{6}\cdots x_{0}$
	00000000	00000000	00000000	00000000

表 29: 相同的數 xor 會等於 0

但是，xor 有一個很好用的性質：給一個整數 x，x ^ x 恆爲 0。表 ?? 清楚表示 xor 的過程，因爲每個位元都是一模一樣的，所以 xor 起來會是 0。

位元技巧：交換兩數 交換兩個 int x 和 y 的值。一般來說 C++ 提供 swap 函數：

```cpp
swap(x, y);
```

程式碼 22: swap 版

不用 swap 的話，我們可以再加開一個變數，先把一個變數裝起來，再把另外一個變數的值丟過去，如程式碼 ??：

```cpp
int tmp = x;
x = y;
y = tmp;
```

程式碼 23: 變數版

最後，我們來看看程式碼 ?? 怎麼運作的：

```cpp
x ^= y;
y ^= x;
x ^= y;
```

程式碼 24: 位元運算版

表 ?? 表示程式碼 ?? 的執行過程，我們可以看到，變數 x 和變數 y 再執行每一行後，實際值的變化，我們可以看到在第二行之後，y ^ x ^ y 有兩個 y，會抵消爲 0，又 0 ^ x $\Rightarrow$ x，於是變數 y 最後的值爲 x。同樣地，變數 x 最後的值也爲 y。

練習題

✓ ***UVa 10469: To Carry or not to Carry***
這題算是位元運算的基本應用。

	變數 x	變數 y
原來的值	x	y
第一行後	x ^ y	y
第二行後	x ^ y	y ^ x ^ y = x
第三行後	x ^ y ^ x = y	x

表 30: 交換兩數

五、 指定運算子

(一) 運算性質

運算子	意義	運算順序	結合性
=	賦值	16	右→左

表 31: 指定運算子

運算子	意義	運算順序	結合性
+=	加法賦值	16	右→左
-=	加法賦值	16	右→左
*=	乘法賦值	16	右→左
/=	除法賦值	16	右→左
%=	取餘賦值	16	右→左

表 32: 複合指定運算子——算術運算子

表 ?? 複合指定運算子代表的意義如下，不難理解：

- `x += a` ⇒ `x = x + a`
- `x -= a` ⇒ `x = x - a`
- `x *= a` ⇒ `x = x * a`
- `x /= a` ⇒ `x = x / a`
- `x %= a` ⇒ `x = x % a`

運算子	意義	運算順序	結合性
<<=	左移賦值	16	**右→左**
>>=	右移賦值	16	**右→左**
&=	位元 AND 賦值	16	**右→左**
^=	位元 XOR 賦值	16	**右→左**
|=	位元 OR 賦值	16	**右→左**

表 33: 複合指定運算子——位元運算子

同樣地，表 ?? 複合指定運算子代表的意義如下，不難：

- `x <<= a` ⇒ `x = x << a`
- `x >>= a` ⇒ `x = x >> a`
- `x &= a` ⇒ `x = x & a`
- `x ^= a` ⇒ `x = x ^ a`
- `x |= a` ⇒ `x = x | a`

運算子	意義	運算順序	結合性
++	字尾遞增	2	左→右
--	字尾遞減	2	左→右
++	字首遞增	3	左→右
--	字首遞減	3	左→右

表 34: 複合指定運算子——遞增遞減

++、-- 是從 +=、-= 簡化而得來，代表的意義都是 `i = i + 1` 和 `j = j - 1`，又個別分兩種，字首系列與字尾系列。

- 字尾系列用法爲「`i++`」、「`j--`」。
- 字首系列用法爲「`++i`」、「`--j`」。

想知道他們的差別，就試試看下面的程式碼有什麼不同吧！

- cout << i++ << endl;
- cout << ++i << endl;
- i++; cout << i << endl;
- ++i; cout << i << endl;

字首系列會先做運算，再回傳，回傳值是運算**後**的值；而**字尾系列**會先回傳，再做運算，回傳值是運算**前**的值。

(二) 未定義行爲

在講述**未定義行爲**之前，我們先看例子 ??，猜猜答案是什麼？

```cpp
int i = 0;
cout << i++ + ++i << endl;
```

程式碼 25: 未定義行爲

答案會是 2 嗎？根據運算順序 (表 ?? 和表 2)，我們先做 ++i，回傳值爲 1，接著做 i++，此時回傳值爲 1 但還沒 ++，最後做 i+i，把兩邊的回傳值相加，變成 2，印出答案再 i++，最終 i 的值爲 2。但事實眞有那麼簡單嗎？

假如：i++ 可以拆成 4 個步驟：

1. 複製 i 值到暫存區 R
2. 回傳 i 值
3. R = R + 1
4. 把 R 值寫回 i

++i 也可以拆成 4 個步驟：

1. 複製 i 值到暫存區 R2
2. R2 = R2 + 1
3. 把 R2 值寫回 i
4. 回傳 i 值

大家可能會以爲，程式執行會像這個樣子：

1. 複製 i 值到暫存區 R2
2. R2 = R2 + 1
3. 把 R2 值寫回 i
4. 回傳 i 值 (此時回傳 1)
5. 複製 i 值到暫存區 R
6. 回傳 i 值 (此時回傳 1)
7. R = R + 1
8. 把 R 值寫回 i
9. 執行 i 的回傳值 (1) + i 的回傳值 (1)，結果爲 2

但因爲現代電腦很多因素，會導致程式依然遵守運算順序，但實際執行會有不同結果，例如：

1. 複製 i 值到暫存區 R2
2. R2 = R2 + 1
3. 把 R2 值寫回 i
4. 複製 i 值到暫存區 R
5. 回傳 i 值 (此時回傳 1)
6. R = R + 1
7. 把 R 值寫回 i
8. 回傳 i 值 (此時回傳 2)
9. 執行 i 的回傳值 (1) + i 的回傳值 (2)，結果爲 3

這個情況是因爲我們做 ++i 或 i++ 雖然看起來像是一步到位，但是**實際上**是很多步驟串起來的結果，C++ **並沒有規定**何種做法才是正確，只要 i++ 能夠正確加一就好。

```
cout << i++ << i++ << i++ << endl;
```

程式碼 26: 未定義行爲

這種規定方法有它的好處，也就是各家編譯器在實作上比較靈活，但是缺點就是因爲每個廠商做出來編譯器功能差異，而導致不同的結果。上面的例子稍微複雜，我們再看一個淺顯的例子：

不難看出，C++ 雖然規定了運算順序，但程式碼 ?? 沒辦法知道我們要先做哪一個 i++，因此這一題的答案也是：**沒有人知道！**在不同的編譯器會有不同的結果，簡單來說，大多數的未定義行爲都是在同一行之內改同一變數**一次**以上。當然，還有各種不同的例子：

- `i = ++i + 1;`
- `i++*++i+i--*--i`
- `a ^= b ^= a ^= b;`

寫程式的時候要避免未定義行爲，因爲這種寫法會導致千百種答案，歸咎於這種寫法本身就不是正確的寫法。

六、 其他運算子

總結來說，萬物對計算機而言皆是「**運算**」，既然是運算，就有「結合性」和「運算順序」。接下來還有一些特別的運算子，將會介紹其功能和用途。

運算子	意義	運算順序	結合性
`sizeof`	求記憶體大小	3	**右→左**
(type)	強制轉型	3	**右→左**
,	逗號	18	左→右

表 35: 其他運算子

`sizeof` 運算子 `sizeof` 可以知道某個**資料型態**或**變數**所使用的位元組數。例如：

- `sizeof(int)` 在**筆者的機器**上會是 4 位元組
- `sizeof(double)` 在**筆者的機器**上會是 8 位元組
- 程式碼 ?? 在**筆者的機器**上會是 1 位元組

```
bool b = true;
cout << sizeof b << endl;
```

程式碼 27: 布林變數的位元組數

注意：每個人的機器會出現不同的結果，參考表 ??，像是前面提到有些機器的 `int` 會是 2 個位元組。

(type) **運算子** C++ 有資料型態，若型態間需要**強制轉換**就要使用這個運算子。例如：

- `int` 變數 x 轉爲 `double` ⇒ (`double`) x 或者 `double`(x)
- `double` 常數轉爲 `int` ⇒ (`int`) 5.14 或者 `int`(5.14)

註：我們說過資料型態代表容器可以裝的資料類型不同，因此我們之後會遇到需要「**改變資料類型**」的狀況，那時需要做型別轉換。

逗號運算子 最後，我們要講一下「逗號運算子」，他是最常被人誤解的**運算子**、**運算子**、**運算子**！(因爲很重要所以要說三次) 逗號運算子可以**分隔**兩個運算式，回傳值是**右邊**運算式的回傳值。

例如：用迴圈讀入 n，直到 $n = 0$ 停止：

```
int n;
while (cin >> n, n) {
}
```

程式碼 28: 迴圈輸入

程式碼 ?? 中，因爲迴圈內的判斷式是回傳 n 的值，只要 n 非零，就會被當成 `true`。

七、 結論

- 句子結尾是分號「;」。
- 初始化的重要性。

- C++ 運算子依照運算順序和結合性做運算，大約了解運算的優先順序。
- 運算優先順序：一元運算子 → 算術運算子 → 比較運算子 → 邏輯運算子 → 位元運算子 → 指定運算子、複合指定運算子 → 逗號運算子
  - 萬一忘記順序怎麼辦呢？
  - 當然是把**括號括好**啦！運算順序只要知道大概，這不是必背的東西，我們的目的是「寫出好程式」而非在運算順序上多作著墨！
- 除以零會遇到的現象。
- 「零」代表 `false`，「非零」代表 `true`。
- 邏輯運算子是短路運算。
- `int` 和 `long long` 如何儲存，以及位元運算技巧。
- 注意未定義行爲。

索 引