

目 錄

第一節 基礎 C++ 技巧	2	第二節 程式架構解析	34
一、程式架構	2	一、位址與指標	35
二、算術運算子	10	二、程式控制	46
三、比較和邏輯運算子	13	三、函數	50
四、位元運算子	16	四、程式技巧	50
五、指定運算子	28	五、C++ 物件導向	50
六、其他運算子	32	第三節 字串處理	50
七、結論	34	一、字元與字串	50
		二、輸入與輸出	57
		三、字串技巧	64
		四、字串應用	64

前言

寫程式依據要求的程度的不同，可以分成兩種：

1. 寫出一個完整的程式：只要照著講義、照著書、照著網路上大神的原始碼打一打，就可以動了。
2. 寫出一個好的程式：
 - (a) 要了解資料怎麼儲存在電腦中
 - (b) 程式怎麼開始執行，為什麼會執行
 - (c) 什麼時候會出現什麼狀況，怎麼判斷出來、怎麼修正 (也就是 debug)
 - (d) 用適當的工具解決問題
 - (e) ... 族繁不及被宰備載

以上就是此講義的培訓目標！目的就是要讓大家熟悉基本的 C++ 語法，以及學會一些 coding 技巧。

至於許胖講義，主要是講述演算法競賽，在演算法競賽中，不僅僅要寫出好的程式，更要具備以下能力：

1. 使用一個「有效」的方法解決問題
2. 不僅如此，還要知道不同工具使用上的優缺點
3. 手爆出很多 code，勇往直前
4. 進到 TOI 二階，保送大學

不過，寫程式不是只有演算法比賽，生命也不是只有一個出口，越往這個領域深入，就會看到更多無盡的事物，例如：

- 遊戲引擎
- 網頁設計
- 手機 App
- 韌體 coding

在資訊領域中有很多子領域，不僅這樣，在其他領域中多少也有許多寫程式的應用，不管最後是否在演算法競賽發光發熱，這裡都是一切的起點。希望各位在之後的內容都要動手快樂寫程式 XD！

本文適合剛認識 C++ 想要迅速熟悉語法，並且邁向演算法競賽的人閱讀。

第一節 基礎 C++ 技巧

本章目標

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

一、程式架構

(一) 基本 C++ 架構

最基礎的 C++ 架構如下：

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

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++ 的輸出符號寫為「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？

```
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

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 因為沒加上這行，程式就會直接執行結束，加上這行，程式會在這裡「等你」。

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？因為「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：

```
1 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。

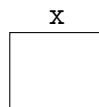


圖 1: 容器

變數的用途就是爲了把數字裝到變數中，

```
1 #include <iostream>
2 using namespace std;
3 int main() {
4     int x;                // 宣告變數 x
5     x = 5;                // 把整數 5 裝進 x 裡面
6     cout << x << endl;   // 印出變數 x 存的值
7 }
```

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真正了解問題出現時的現象，了解出問題的原因才有辦法 debug，為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。

<pre>1 int a; 2 int b;</pre>	<pre>1 int a, b;</pre>
(a) 兩個宣告	(b) 簡化版

程式碼 5: 宣告兩個變數

以此類推，宣告三個整數也是如法炮製，如程式碼 6：

```
1  int a, b, c;
```

程式碼 6: 宣告三個變數

變數初始化 剛剛我們說明變數就像是容器，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。

```
1  #include <iostream>  
2  using namespace std;  
3  int main() {  
4      int x;                // 宣告變數 x  
5      cout << x << endl;  // 印出變數 x 存的值  
6  }
```

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，一般來說，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。因為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。



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？

```
1  #include <iostream>
2  using namespace std;
3  int main() {
4      int x;
5      cin >> x;
6      cout << x << endl;
7  }
```

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：

```
1 int x, y;  
2 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為「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：

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」。

```
1 bool b;
```

程式碼 10: 布林值宣告

```
1 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，稱為**賦值**。例如，程式碼 12 把整數 5 裝進整數變數 x 中：

```
1 int x;  
2 x = 5;
```

程式碼 12: 賦值

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：

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

程式碼 13: 賦值簡化

練習 下面有一段程式碼：

```
1 bool b;  
2 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。

```
1 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：

```
1 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，可以簡化為 `.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：

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

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，但是程式為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： $1 + 2$ ：「+」稱為「運算子」，「1」和「2」稱為運算元（我們常稱為「被加數」和「加數」）。

(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 $3 + 3 = 6$
- 先算 $2 + 3 = 5$ ，再算 $1 + 5 = 6$

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：

$1 + 2 * 3 - 4$		* 的運算順序最高
$= 1 + 6 - 4$	加法和減法運算順序相同，依照結合性從左到右算	
$= 7 - 4$		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，稍後就會知道為什麼。

(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 被視為 `int`，因此 C++ 會做「整數除法」；而在 $8.0 / 5.0$ 中，8.0 和 5.0 被視為浮點數 `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

大部分的人會認為，% 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 $\text{mod } m$ 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 3：

比較運算子	意義	運算順序	結合性
<code>==</code>	等於	9	左→右
<code>!=</code>	不等於	9	左→右
<code>></code>	大於	8	左→右
<code><</code>	小於	8	左→右
<code>>=</code>	不小於	8	左→右
<code><=</code>	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，則為 `true`、否則就是 `false`。這種概念我們稱為「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，因為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。

	<code>n % m != 0</code>	<code>n % m</code>
當 <code>n % m</code> 不為零	<code>true</code>	<code>true</code>
當 <code>n % m</code> 為零	<code>false</code>	<code>false</code>

表 4: 真值表

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
<code>&&</code>	且	13	左→右
<code> </code>	或	14	左→右
<code>!</code>	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 $a \leq x \leq b$ ；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 5 可以知道， \leq 運算子在列出很多個時，會由左到右算，因此在左側的 $a \leq x$ 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 $a \leq x \leq b$ 。

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

反過來， $a \leq x$ 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： $a \leq x \ \&\& \ x \leq b$ 。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 $\|\|$ 運算子的行為：

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

```

1  int i, j;
2  i = j = 0;
3  if ((i++ < 0) && (j++ > 0))
4      cout << "XD" << endl; // 這行不會輸出
5  cout << i << "□" << j << endl; // i 為 1, j 為 0

```

程式碼 16: 範例

```

1  int i, j;
2  i = j = 0;
3  if ((i++ >= 0) || (j++ < 0))
4      cout << "XD" << endl; // 會輸出 XD
5  cout << i << "□" << j << endl; // i 為 1, j 為 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，但是上傳時因為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道問題。

✓ 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)：因為位元很多，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
<code>bool</code>	1 位元組
<code>int</code>	2 或 4 位元組
<code>long long</code>	4 或 8 位元組
<code>double</code>	8 位元組

表 7: 位元組長度

表 7 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 9， 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。

因為 `int` 的儲存方式很特別，要多花一些力氣說明。

$x_{31}x_{30} \cdots x_{24}$	$x_{23}x_{22} \cdots x_{16}$	$x_{15}x_{14} \cdots x_8$	$x_7x_6 \cdots x_0$
------------------------------	------------------------------	---------------------------	---------------------

表 9: 二進位 `int`

`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 11。

00000000	00000000	00000000	11111111
----------	----------	----------	----------

表 11: 255 的二進位表示法

`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;`：

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 因為超過 32 位元，所以被捨棄，稱為溢位。

這種表示法稱為二補數 (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 $(-x)+x$ 會因為溢位而等於 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。

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

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， $2 \ll 2 \Rightarrow 8$ 即是把 2 在二進位的位元往左移 2 格：

00000000	00000000	00000000	00000010
↓			
00000000	00000000	00000000	00001000

表 14: 左移的情況

類似的情況， $5 \gg 1 \Rightarrow 2$ 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
↓			
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，因為我們知道 x_{31} 決定正負號，以下例子讀者們可以試看，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：

	00000000	00000000	00000000	00000 101
&	00000000	00000000	00000000	00000 011
	00000000	00000000	00000000	00000 001

表 17: 5 & 3 的狀況

5 & 3 結果會是 1：

5 | 3 結果會是 7：

	00000000	00000000	00000000	00000 101
	00000000	00000000	00000000	00000 011
	00000000	00000000	00000000	00000 111

表 18: 5 | 3 的狀況

5 ^ 3 結果會是 6：

	00000000	00000000	00000000	00000 101
^	00000000	00000000	00000000	00000 011
	00000000	00000000	00000000	00000 110

表 19: 5 ^ 3 的狀況

補數運算子 對於兩個位元 x 和 y，遵守以下運算規則：

~	1	0
	0	1

表 20: 補數運算子

簡單來說就是「1 變 0，0 變 1」（相當於邏輯運算子的 !），又稱為 1's 補數。例如： $\sim 0 \Rightarrow -1$ 。

(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 $\sim\sim 3$ 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」，回傳結果為 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 $(1 \ll k) - 1$ ，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	11111000
----------	----------	----------	-----------------

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 $\sim x$ 的不同，就會發現，他們事實上只差 1。例如：

- $\sim 0 \Rightarrow -1$
- $\sim 123 \Rightarrow -124$

從上面的結論可以歸納出 $-x$ 恰好是 $(\sim x)+1$ 。

(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

下面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 16 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 16a 和 16c 的規則會如下：

表 22: 有未知數的位元運算

& x		x	
1	x	1	1
0	0	0	x
(a) and 運算子		(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：
 - $x \& 0$ 把該位元設為 0
 - $x | 1$ 把該位元設為 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：

	$x_{31}x_{30} \cdots x_{24}$	$x_{23}x_{22} \cdots x_{16}$	$x_{15}x_{14} \cdots x_8$	$x_7x_6 \cdots x_0$
<code>&</code>	00000000	00000000	00000000	0000000 1
	00000000	00000000	00000000	0000000 x_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 改為 1 同樣使用 `$x \mid 1$` ：

	$x_{31}x_{30} \cdots x_{24}$	$x_{23}x_{22} \cdots x_{16}$	$x_{15}x_{14} \cdots x_8$	$x_7x_6 \cdots x_0$
<code> </code>	00000000	00000000	00000000	0000000 1
	$x_{31}x_{30} \cdots x_{24}$	$x_{23}x_{22} \cdots x_{16}$	$x_{15}x_{14} \cdots x_8$	$x_7x_6 \cdots x_1$ 1

表 24: 將 x_0 改為 1

根據表 22b，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 強制設為 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、連續位元為 1，例如：將 x_2 改為 1。

	$x_{31}x_{30} \cdots x_{24}$	$x_{23}x_{22} \cdots x_{16}$	$x_{15}x_{14} \cdots x_8$	$x_7 \cdots x_3x_2x_1x_0$
<code> </code>	00000000	00000000	00000000	00000 1 00
	$x_{31}x_{30} \cdots x_{24}$	$x_{23}x_{22} \cdots x_{16}$	$x_{15}x_{14} \cdots x_8$	$x_7 \cdots x_3$ 1 x_1x_0

表 25: 將 x_2 改為 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 16a 的特性：任何位元和 0 and 起來恆為 0。

	$x_{31}x_{30} \cdots x_{24}$	$x_{23}x_{22} \cdots x_{16}$	$x_{15}x_{14} \cdots x_8$	$x_7x_6 \cdots x_0$
&	11111111	11111111	11111111	11111110
	$x_{31}x_{30} \cdots x_{24}$	$x_{23}x_{22} \cdots x_{16}$	$x_{15}x_{14} \cdots x_8$	$x_7x_6 \cdots x_1$ 0

表 26: 將 x_0 設為 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 26 可以寫爲程式碼 18。

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

程式碼 18: 將 x_0 設為 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 $x \& ((1 \ll 2) - 1)$ 。

以此類推，求 2^k 的餘數就可以寫成 $x \& ((1 \ll k) - 1)$ ，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：給你一個正整數 x ，問在二進位下有幾個 1？以下有幾個範例：

- $\text{PARITY}(5)$ 如表 27，可以看出二進位下有兩個 1，因此結果為 2。

00000000	00000000	00000000	00000101
----------	----------	----------	----------

表 27: 5 的 parity

- $\text{PARITY}(255)$ 如表 28，結果為 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：

```
1 for (cnt = 0; x; x /= 2) {  
2     if (x % 2 != 0)  
3         cnt++;  
4 }
```

程式碼 19: Parity 普通寫法

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：

```
1 for (cnt = 0; x; x >>= 1) { // 右移代替除法  
2     if (x & 1) // 省略「!= 0」，同時把除法改成位元運算  
3         cnt++;  
4 }
```

程式碼 20: Parity 位元運算寫法

以下是檢查 Parity 是否為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：

```

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

```

程式碼 21: Parity 究極寫法

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 16b 可以知道 xor 運算的性質是「同為 0 或同為 1 xor 起來就是 0」。

	$x_{31}x_{30} \cdots x_{24}$	$x_{23}x_{22} \cdots x_{16}$	$x_{15}x_{14} \cdots x_8$	$x_7x_6 \cdots x_0$
\wedge	$x_{31}x_{30} \cdots x_{24}$	$x_{23}x_{22} \cdots x_{16}$	$x_{15}x_{14} \cdots x_8$	$x_7x_6 \cdots x_0$
	00000000	00000000	00000000	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 ， $x \wedge x$ 恆為 0。表 29 清楚表示 xor 的過程，因為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：

```

1 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 23：

最後，我們來看看程式碼 24 怎麼運作的：

表 30 表示程式碼 24 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， $y \wedge x \wedge y$ 有兩個 y ，會抵消為 0，又 $0 \wedge x \Rightarrow x$ ，於是變數 y 最後的值為 x 。同樣地，變數 x 最後的值也為 y 。

```

1  int tmp = x;
2  x = y;
3  y = tmp;

```

程式碼 23: 變數版

```

1  x ^= y;
2  y ^= x;
3  x ^= y;

```

程式碼 24: 位元運算版

	變數 x	變數 y
原來的值	x	y
第一行後	$x \oplus y$	y
第二行後	$x \oplus y$	$y \oplus x \oplus y = x$
第三行後	$x \oplus y \oplus x = y$	x

表 30: 交換兩數

練習題

✓ UVa 10469: *To Carry or not to Carry*

這題算是位元運算的基本應用。

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

表 32 複合指定運算子代表的意義如下，不難理解：

- $x += a \Rightarrow x = x + a$
- $x -= a \Rightarrow x = x - a$

運算子	意義	運算順序	結合性
+=	加法賦值	16	右→左
-=	減法賦值	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

- $x *= a \Rightarrow x = x * a$
- $x /= a \Rightarrow x = x / a$
- $x %= a \Rightarrow x = x \% a$

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 33 複合指定運算子代表的意義如下，不難：

- $x <<= a \Rightarrow x = x << a$
- $x >>= a \Rightarrow x = x >> a$
- $x \&= a \Rightarrow x = x \& a$
- $x \^{}= a \Rightarrow x = x \^{} a$
- $x |= a \Rightarrow x = x | a$

++、-- 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。

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

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

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) 未定義行為

在講述未定義行為之前，我們先看例子 25，猜猜答案是什麼？

```
1  int i = 0;
2  cout << i++ + ++i << endl;
```

程式碼 25: 未定義行為

答案會是 2 嗎？根據運算順序 (表 34 和表 2)，我們先做 `++i`，回傳值為 1，接著做 `i++`，此時回傳值為 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` 的值為 2。但事實真有那麼簡單嗎？

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

大家可能會以為，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)，結果為 2

但因為現代電腦很多因素，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)，結果為 3

這個情況是因為我們做 $++i$ 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。

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，但是缺點就是因為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：

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

程式碼 26: 未定義行為

不難看出，C++ 雖然規定了運算順序，但程式碼 26 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，大多數的未定義行為都是在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為，因為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
<code>sizeof</code>	求記憶體大小	3	右→左
<code>(type)</code>	強制轉型	3	右→左
<code>,</code>	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 27 在筆者的機器上會是 1 位元組

```
1 bool b = true;
2 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 7，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` 轉為 `double` \Rightarrow `(double) x` 或者 `double(x)`
- `double` 常數轉為 `int` \Rightarrow `(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！（因為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：

```
1  int n;  
2  while (cin >> n, n) {  
3  }
```

程式碼 28: 迴圈輸入

程式碼 28 中，因為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。
- 注意未定義行為。

第二節 程式架構解析

這一節主要延續上一節的思維，但著重在了解程式如何執行，利用這些知識順利寫出架構簡潔、容易除錯程式。因此在這一節練習題較少，大多是重要的觀念。

一、位址與指標

(一) 溢位現象

以下程式碼會發生什麼現象？

```
1 int x = 2147483647;
2 cout << x + 1 << endl;
```

程式碼 29: 產生溢位的程式碼

若讀者的 `int` 也是 4 個位元組的話，那麼就會得到 -2147483648！這種現象我們稱為溢位現象 (Overflow)。我們從二進位下看這段程式，會比較了解：

01111111	11111111	11111111	11111111
----------	----------	----------	----------

表 36: 2147483647

表 36 表示 2147483647 在 C++ 當中如何儲存，讀者可以驗證 $2147483647 = 2^{31} - 1$ 。若我們此時對 2147483647 加 1，就會得到下面的結果：

10000000	00000000	00000000	00000000
----------	----------	----------	----------

表 37: 2147483647 + 1

我們用 `int` 的儲存方法驗證，這個數字就是 -2147483648，也就是 -2^{32} ！為什麼會這樣子呢？

我們可以建立一個有點抽象的概念，只要表示資料的方法所以用的記憶體大小是**有限**的，那麼就只能表示**有限多種**資料。例如：`int` 通常是 4 個位元組，也就是 $4 \times 8 = 32$ 個位元，每個位元只能表示 0 和 1 兩種可能性，則最多只能表示 2^{32} 種整數。

這 2^{32} 種整數表示法當中，我們切一半， 2^{31} 個表示負數， 2^{31} 表示非負整數，其中有一個 0，剩下 $2^{31} - 1$ 個是正整數，因此 `int` 的範圍就是介於 -2^{31} 到 $2^{31} - 1$ 。

表 38 表示每一種資料型態常見的上下界範圍，其中 `unsigned` 類型代表「不帶負號」，也就是說 `unsigned` 系列會把所有符號拿去表示非負整數。

資料型態	位元組	通常下界	通常上界
char	1	-128	127
short	2	-32768	32767
int	4	-2147483648	2147483647
long long	8	-9223372036854775808	9223372036854775807
float	4	-3.40282×10^{38}	3.40282×10^{38}
double	8	-1.79769×10^{308}	1.79769×10^{308}
unsigned char	1	0	255
unsigned short	2	0	65535
unsigned int	4	0	4294967295
unsigned long long	8	0	18446744073709551615

表 38: 資料型態上下界

此外，需要注意的有幾點：

- `int` 的上限是 2147483647，用十六進位表示為 0x7FFFFFFF
- `long long` 範圍大概是 -9×10^{18} 到 9×10^{18} 之間
- `double` 的範圍介於 -10^{308} 至 10^{308} 左右

這些範圍可以在標頭檔 `<climits>` 和 `<float>` 當中查詢到，實際範圍會依據不同計算機而有差異，使用方法自行 google，這裡不贅述。

(二) 記憶體

之前我們提到位元和位元組以及 `sizeof` 等觀念，接下來要進入有關記憶體的部份。首先，我們常常提到的記憶體有分廣義和狹義的分別，廣義的記憶體可以指稱所有儲存資料的設備，表 39 列出計算機中常用的儲存設備：

狹義的記憶體就是主記憶體，俗稱「記憶體」，程式在開始運行前，會將存在硬碟當中的程式資料移到記憶體當中，才會執行程式。

(三) 位址

主記憶體可以看做是一條很長、連續的位元組，程式執行時，會佔據其中一區塊，如圖 3：

種類	原文	存取速度	容量	用途
暫存器	Register	1 CPU 週期	數百 Bytes 內	CPU 內部暫存運算的資料
快取記憶體	Cache	數十 CPU 週期	數十 MB 內	協調 CPU 和主記憶體的速度
主記憶體	Main Memory	數百 CPU 週期	8 GB 左右	執行程式、暫存資料等
碟盤設備 (硬碟、光碟)	Disk Storage	數百萬 CPU 周期	數 TB	永久儲存程式、資料

表 39: 廣義的記憶體，又稱為記憶體階層 (2016年資料)

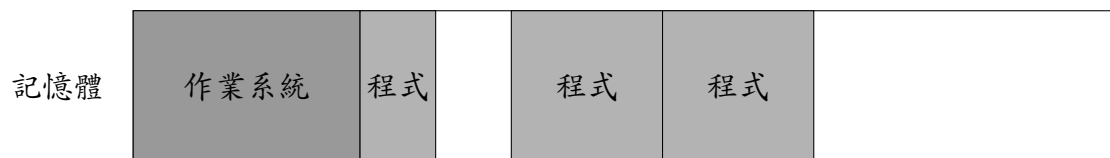


圖 3: 記憶體與程式

記憶體可以比作「土地」，一開始有一大片未經使用的土地，由作業系統和程式去分配用途，爲了方便管理記憶體，計算機會幫每個位元組標記「地址」，在此我們就稱爲「位址」。

此外，位元組是擁有地址的最小單位，單個位元並沒有位址。

取址運算子 位址通常是一個 16 進位的數字，如果我們要知道一個變數的位址，我們使用取址運算子 `&`，例如程式碼 30：

```

1 int a = 16;
2 cout << "Address of a = " << &a << endl;

```

程式碼 30: 印出 a 的位址

筆者的執行結果爲：Address of a = 003FF07C，代表變數 a 實際的位址是在 0x003FF07C 的位元組，相當於是他的「門牌號碼」，因爲 int 通常爲 4 個位元組，因此會佔據 0x003FF07C、0x003FF07D、0x003FF07E、0x003FF07F 這四個位元組。如圖 4：

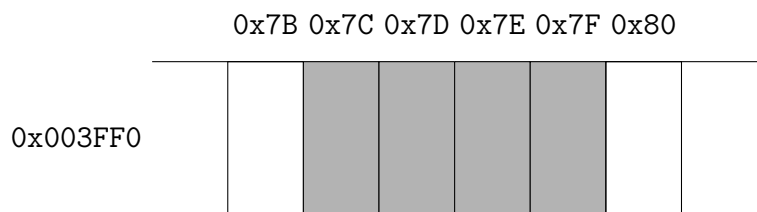


圖 4: 變數 a 實際的記憶體位置

這個結果會因為不同機器、每次程式執行分配的記憶體不同而不一樣 (總之就是不一定啦！(ノ。□。)ノ ㄣ ㄣ ㄣ)，但概念是相同的。

大字節序和小字節序 但是我們要怎麼知道變數 a 實際怎麼儲存在記憶體中呢？很多人會以為像是圖 5：

	0x7C	0x7D	0x7E	0x7F
0x003FF0	00000000	00000000	00000000	00001000

圖 5: 大字節序儲存方法

但這個說法不完全對，圖 5 的儲存方法被稱為「**大字節序 (Big Endian)**」，也就是 `int` 的高位數會儲存在位址比較小的地方。

另一種跟他相對的稱為「**小字節序 (Little Endian)**」，也就是數字的高位數儲存在位址比較大的地方。例如用整數 0x12345678 來表示這兩種儲存方法的差異如圖 6a 和 6b：

0x7C	0x7D	0x7E	0x7F	0x7C	0x7D	0x7E	0x7F
0x12	0x34	0x56	0x78	0x78	0x56	0x34	0x12

(a) 以大字節序儲存
(b) 以小字節序儲存

圖 6: 0x12345678 不同儲存方法

除此之外，還有一類是「**混合字節序 (Middle Endian)**」，是大字節序和小字節序混用或者是其他的狀況，這裡不贅述。

無論是大字節序還是小字節序，在程式當中都是表示「0x12345678」這個數字，這些儲存方法只是表示計算機實際儲存資料的差異，程式配置一塊記憶體用來儲存 0x12345678，實際怎麼儲存在很多狀況下其實並不重要，但偶爾要做一些操作時，就會牽扯到這個概念。

(四) 指標

指標是一個概念，他代表一個箭頭指向一塊記憶體。如圖 7。

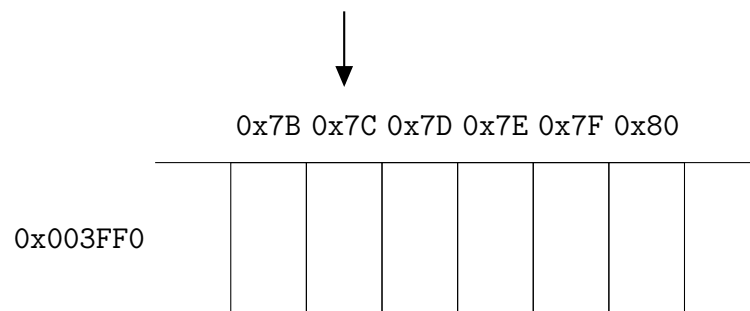


圖 7: 指標概念

C++ 是利用儲存記憶體位址的方式實做指標，如何實現一個指標我們慢慢細說。

宣告 首先，程式碼 31 宣告一個指標變數 ptr。

```
1 int *ptr;
```

程式碼 31: 宣告指標變數 ptr

在此，宣告指標變數的規則和之前宣告變數都是相同的原則：變數名稱和用途，此時 ptr 的資料型態為 int*，代表這是一個指向 int 的指標。因為資料型態是 int*，所以也可用程式碼 32 的方式來宣告。

```
1 int* ptr;
```

程式碼 32: 宣告指標變數 ptr

讀者要注意的一點是，當宣告多個指標變數時，不能寫成 int* ptr, ptr2，在這個情形下，C++ 會把 ptr2 宣告成 int，正確宣告多指標變數要像程式碼 33。

```
1 int *ptr, *ptr2;
```

程式碼 33: 宣告多個指標變數

賦值 剛剛說過，C++ 指標的運作是讓指標變數儲存位址，如果以之前變數 a 的例子來說，我們知道變數 a 的位址是 0x003FF07C，若我們要把 ptr 指向變數 a 所在的記憶體，我們可以用先前講過的取址運算子，得到 a 的位址，如程式碼 34。

```
1 int a = 16;
2 int *ptr = &a;
```

程式碼 34: 指標的賦值

程式碼 34 的實際狀況如圖 8。

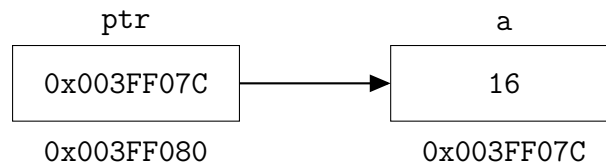


圖 8: 程式碼 34 的狀況

由圖 8 可以看出以下幾個重點：

- 雖然指標的概念是「箭頭」，但 C++ 實際上還是要用變數來代表。
- 既然 C++ 中指標也是一個變數，那麼就需要另外配置記憶體。
- C++ 實做指標，就是儲存位址。

取值運算子 指標最大的用處，就是可以知道指向位址的值，C++ 中取得指向位址的值使用取值運算子 *，以程式碼 35 為例。

程式碼 35 中，第 4 行的 * 是取值運算子，回傳指向記憶體的值，因此會印出「16」。

而在第 5 行中，因為 C++ 的指標也是一變數，因此會將 b 的位址儲存在 ptr 裡面，也就是指向變數 b，因此第 6 行會印出 b 的值，也就是「4」。


```

1  int a = 16;
2  int b = 4;
3  int *ptr = &a;
4  cout << *ptr << endl;
5  ptr = &b;
6  cout << *ptr << endl;

```

程式碼 35: 取值運算子

不同的資料型態，都有對應的指標型態，例如指向 `int` 的指標型態為 `int*`、指向 `double` 的指標型態為 `double*`，以此類推。以下情況由讀者做觀察，想想為什麼會有這些現象，有和預想中的不一樣嗎？

- `sizeof(int*)` 和 `sizeof(int)`
- `sizeof(long long*)` 和 `sizeof(long long)`
- `sizeof(double*)` 和 `sizeof(double)`

此外，讀者可以觀察一下程式碼 36，這一章節主要是讓大家能夠了解指標的概念，並非以熟練指標為主：

```

1  int a = 16;
2  int *ptr = &a;
3  cout << "Value_of_a=" << a << endl;
4  cout << "Address_of_a=" << &a << endl;
5  cout << "Value_of_ptr=" << *ptr << endl;
6  cout << "Value_of_ptr=" << ptr << endl;
7  cout << "Address_of_ptr=" << &ptr << endl;

```

程式碼 36: 指標小練習

除此之外，我們也可對指標所指的對象進行運算，如程式碼 37。

在程式碼 37 中，第 4 行會印出「17」，因為第 3 行的 `*ptr` 是先對 `ptr` 取值，得到變數 `a` 的值，接著對 `a` 做 `++`。要注意的是我們是對「`ptr` 所指的值得累加」，讀者可以比較 `ptr++` 與 `(*ptr)++` 的不同。

```

1  int a = 16;
2  int *ptr = &a;
3  (*ptr)++;
4  cout << a << endl;

```

程式碼 37: 指標操作

有了基本的指標概念之後，我們接下來看「指標的指標」，一個 `int` 指標的型態為 `int*`，如果是指向 `int*` 的指標，則型態為 `int**`，用法和普通的指標相同，程式碼 38 展示了指標的指標的用法。

```

1  int a = 16;
2  int *ptr, **tmp;
3  ptr = &a;
4  tmp = &ptr;
5  cout << "a:" << endl;
6  cout << "Value_of_a=" << a << endl;
7  cout << "Address_of_a=" << &a << endl;
8  cout << "ptr:" << endl;
9  cout << "Value_of_ptr=" << ptr << endl;
10 cout << "Value_of_*ptr=" << *ptr << endl;
11 cout << "Address_of_ptr=" << &ptr << endl;
12 cout << "tmp:" << endl;
13 cout << "Value_of_tmp=" << tmp << endl;
14 cout << "Value_of_*tmp=" << *tmp << endl;
15 cout << "Address_of_tmp=" << &tmp << endl;

```

程式碼 38: 指標的指標

讀者可以參考圖 9，第二行同時宣告 `int*` 和 `int**` 兩種指標，分別是變數 `ptr` 和 `tmp`，其中 `ptr` 指向變數 `a`，`tmp` 指向變數 `ptr`，其餘不贅述。

比較特別的是以下操作，如果程式碼 38 連續運用取址運算子和取值運算子，會有什麼結果呢？

- `**tmp` 的值為何？
- `*&ptr` 和 `&*ptr` 有什麼不同？

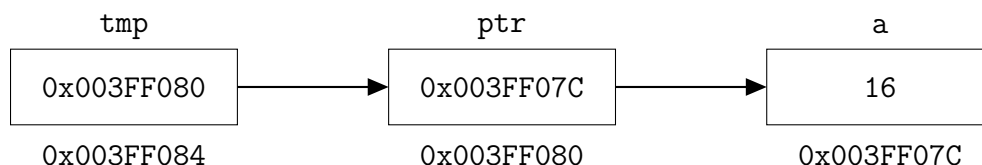


圖 9: 程式碼 38 的狀況

- `&&a` 可以運作嗎？

資料型態 程式碼 39 展示了指標型態的用途，這個例子比較複雜，在第 2 行時，型態為 `char*` 的指標 `ptr`「刻意」去接 `x` 的位址，但由於 `x` 位址的型態為 `int*`，因此得做型別轉換。

```

1 int x = 0x01020304;
2 char* ptr = (char*)&x;
3 cout << (int)*ptr << endl;

```

程式碼 39: 指標型態的用途

接著第三行我們把 `ptr` 指向的值轉換成 `int` 輸出，會得到 `0x01020304` 的十進位數字嗎？不會，否則就不會這樣問了。

我們回頭來探討記憶體和資料型態的關係，前面有提到記憶體就好比是「土地」，土地可以規劃為住宅用、工業用土地等等。

宣告一個變數，相當於程式會配給變數一塊記憶體，但是這個記憶體的「用途」，就是看宣告時的資料型態，例如 `int x` 的型態是 `int`，因此程式才會知道要配給變數 `x` 四個位元組。

同樣的情況也發生在指標身上，指標也需要知道他指向的記憶體用途為何，才能依照該有的格式去存取。程式碼 39 第 2 行，當我們利用 `char*` 指標去指向 `x` 的位址，`ptr` 實際上會把它所指向的記憶體當作 `char` 來存取，如圖 10。

為了簡化描述位址，我們將變數 `x` 第一個位元組的位址稱為 `X`，依序為 `X+1`、`X+2`、`X+3`。當我們用 `ptr` 指向位址 `X` 時，因為 `ptr` 會認定他指到的資料是 `char`，因此輸出時只會輸出一個位元組的資料，也就是位址為 `X` 所存的資料。

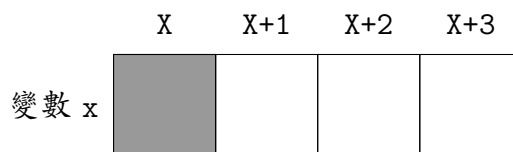


圖 10: 實際上 ptr 的有效範圍

然而最終答案會因為不同電腦而有差異，還記得大字節序和小字節序嗎？如果是大字節序的儲存方法，位址 X 儲存的數字為 0x01，而小字節序會儲存 0x04。

(五) 記憶體操作

這一段我們介紹 C++ 除了指標之外，一些對記憶體常見的操作方法，以下兩個函式在 `<cstring>` 中：

```
1 void* memset(void* ptr, int value, size_t num);
2 void* memcpy(void* destination, const void* source, size_t num);
```

程式碼 40: 兩個常用的函式

這兩個函式的回傳值是 `void*`，什麼意思呢？`void` 有兩層意義，當回傳值為 `void` 時，代表這個函式「**沒有回傳值**」，而當回傳值為一個 `void*` 指標時，這個指標會指向某一塊記憶體，此塊記憶體的用途是「無型態」，也就是單純當作記憶體來使用，不把他看做 `int`、`double` 等型態。

memset 函式 `memset` 函式傳三個參數：`ptr`、`value` 和 `num`，其中 `ptr` 會指向一塊記憶體，`memset` 函式的目的是將 `ptr` 指向的記憶體中，把前 `num` 個位元組的值改成 `value`。

```
1 int x;
2 memset(&x, 1, sizeof(x));
3 cout << x << endl;
```

程式碼 41: `memset` 的基本用法

舉例來說，假設我們有一個 `int` 變數 `x`，如程式碼 41，猜猜變數 `x` 會是多少呢？哈，答案並不是「1」！

首先，就像剛剛提過，ptr 只有單純指向記憶體，既然是視為記憶體。那它就是一個位元組一個位元組依序修改，因此程式碼 41 的結果會像圖 11。

00000000 1	00000000 1	00000000 1	00000000 1
-------------------	-------------------	-------------------	-------------------

圖 11: 程式碼 41 得到的結果

由此可知：

- 因為每次都是把每個位元組初始化，所以 value 的值會介於 0 到 255 之間
- 第三個參數是代表要初始化多少個位元組，往往我們都是初始化**所有**位元組，與其親自計算初始化的變數有多少位元組，不如取巧使用 `sizeof` 運算子

最後 `memset` 函式會回傳修改資料後的 ptr 指標。

memcpy 函式 這個函式與 `memset` 類似，只是差在 `memcpy` 就是把 source 指標的資料，複製前 num 個位元組到 destination 指標所指的記憶體。如程式碼 42 約略敘述它的用法，之後會介紹比較廣泛的用途。

```
1  int x, y;  
2  x = 5;  
3  y = 2;  
4  memcpy(&x, &y, sizeof(y));  
5  cout << x << endl;
```

程式碼 42: `memcpy` 用法

最後，`memcpy` 會回傳 destination 指標。

(六) 執行時期配置

前面有提到程式放到記憶體中才會執行，實際上一支程式在記憶體中會有五個主要的區塊，每個區塊會放置特定的資料，程式架構和配置會因作業系統不同而有差異。

- TEXT 區塊：編譯過後的二進位程式碼
- DATA 區塊：有初始化的全域變數、靜態變數等

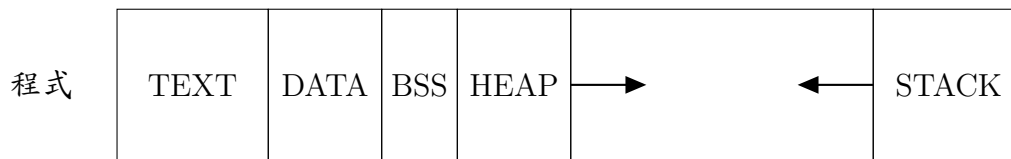


圖 12: 程式在記憶體執行的架構

- BSS 區塊：未初始化的全域變數、靜態變數等
- HEAP 區塊：動態配置的變數等，使用一種稱為堆積 (Heap) 的資料結構
- STACK 區塊：函數呼叫、區域變數等，使用稱為堆疊 (Stack) 的資料結構

這些區塊的概念在越後面就會越常使用，有興趣的讀者可以自行 google。

二、 程式控制

(一) 程式區塊

試試看程式碼 43 的兩個例子會發生什麼事？

```

1  int main() {
2      {
3          int x = 2;
4      }
5      cout << x << endl;
6  }
```

(a) 被大括號包住的 x

```

1  int main() {
2      int x = 2;
3      {
4          cout << x << endl;
5      }
6  }
```

(b) 另一個例子

程式碼 43: 程式區塊

C++ 中的變數有可視範圍 (Scope) 的觀念，通常變數會以函式、大括號做為區隔。例如程式碼 43a 中，變數 x 被包含在大括號中，狀況如圖 13a。

變數的可視範圍就像洋蔥一樣，外面一層的變數可以被裡面一層的變數看見，因此程式碼 43a 的 cout 沒辦法看見包在大括號的變數 x。

程式碼 43b 展示另外一個例子，結果如圖 13b，雖然變數 x 在外層，但裡面的 cout 會一層一層往外找變數 x，結果就是輸出 2。

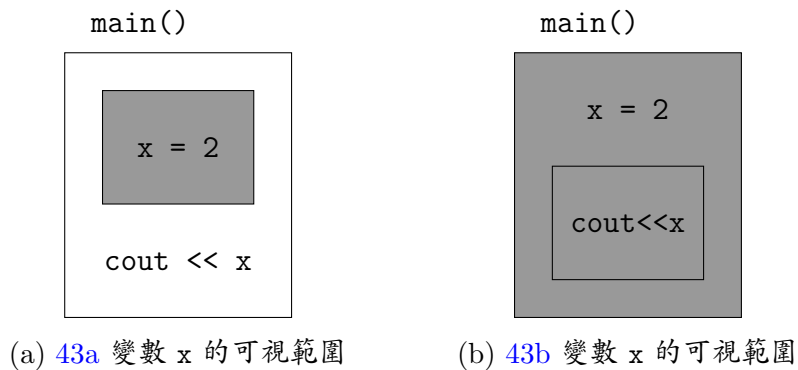


圖 13: 程式碼 43 的狀況

(二) 選擇結構

if 結構 選擇結構在 C++ 中就是 **if**。if 最簡單的語法如程式碼 44。

<pre> 1 int a = 4; 2 if (a < 10) 3 cout << a << endl; </pre> <p>(a) 單行指令</p>	<pre> 1 int a = 4; 2 if (a < 10) { 3 a += 5; 4 cout << a << endl; 5 } </pre> <p>(b) 程式區塊</p>
--	--

程式碼 44: if 的用法

綜觀兩種情況，在 **if** 後面的小括號放**邏輯運算式**，只要邏輯運算式為 **true**，就會執行後面的語句，若要執行多行語句，則要使用程式區塊用大括號括好。

這個結構可以幫助設計一個條件開關，若 **true** 執行某些程式；反之，若是 **false** 則否。因此程式碼 44a 第 3 行，和程式碼 44b 第 3 行至第 5 行會被執行。

if-else 結構 條件判斷可以更進化為 **if-else** 結構，語法如程式碼 45。**if-else** 做的是：當邏輯運算式的結果為 **true**，執行 **if** 的區塊；如果是 **false**，則執行 **else** 區塊。

同樣地，**else** 也可以改為程式區塊。當你要判斷的條件比較多時，**if-else** 可以連用，如程式碼 46。

```

1  int a = 4;
2  if (a < 3)
3      cout << "Yes!" << endl;
4  else
5      cout << "QQ" << endl;

```

程式碼 45: if-else 結構

```

1  int a = 4;
2  if (a < 3)
3      cout << "Case_1" << endl;
4  else if (3 <= a && a < 6)
5      cout << "Case_2" << endl;
6  else
7      cout << "Case_3" << endl;

```

程式碼 46: if 和 else 連用

程式碼 46 中 if-else 可以一直接續下去，除此之外，類似的結構也有 switch 等，這裡不贅述。

懸置 else 問題 將程式碼 47a 拔掉大括號，變成程式碼 47b，會有什麼差別？

<pre> 1 if (0) { 2 if (0) cout << "QQ" << endl; 3 } 4 else cout << "XD" << endl; </pre>	<pre> 1 if (0) 2 if (0) cout << "QQ" << endl; 3 else cout << "XD" << endl; </pre>
(a) 危險的 else	(b) 編譯器會不知道是哪一個 if 的 else

程式碼 47: 懸置的 else

通常沒括大括號的情況下，最後一個 else 會匹配到最近的 if，讀者可以驗證這兩段程式碼的不同。

應用：找極值

<pre> 1 for (int i = 0; i < 10; i++) { 2 cout << i << endl; 3 }</pre>	<pre> 1 { 2 int i = 0; 3 while (i < 10) { 4 cout << i << endl; 5 i++; 6 } 7 }</pre>
---	---

(a) `for` 語法

(b) 對應的 `while` 語法

程式碼 48: `for` 和 `while` 的對應關係

(三) 迴圈結構

比較迴圈結構

常見的輸入形式

(四) 陣列

接著來

三、 函數

- (一) 傳值呼叫
- (二) 傳址呼叫
- (三) 傳參考呼叫
- (四) 函數多載
- (五) 程式區塊

四、 程式技巧

- (一) 函式化
- (二) `#define` 與 `inline`

五、 C++ 物件導向

- (一) 物件與類別
- (二) 建構子與解構子
- (三) 運算子多載
- (四) 命名空間

第三節 字串處理

一、 字元與字串

(一) 字元與 ASCII

C++ 中的**字元** (Character) `char` 其實是儲存一個 0 到 255 的整數，在電腦中有一個**符號表**，每個符號都有他各自的編號。輸出字元時，計算機就會自動將 `char` 裡面的整數去查符號表，印出對應符號，這個表格我們稱為 **ASCII 碼**。雖然 `char` 印出來是符號，但實際上儲存的是整數。

ASCII 碼網路上都能查到，在這邊只提到幾個重要的觀念就好，各位讀者可以邊看網路上查到的 ASCII 碼邊看接下來的內容。

字元表示方法是用單引號包上一個符號，如：`'0'`。每個符號都有一個編號，例如 `'0'` 的編號是 48，其他常用的字元編號如表 40。

字元	'0'	'A'	'a'	'␣'	'\n'
編號	48	65	97	32	10
備註				空白	換行字元

表 40: 常用字元編號

特殊字元 有些字元因為沒辦法直接用單引號包住符號的方式表示，就會用「倒斜線+字元」來代表，表 41 是一些常見的特殊字元。

字元	意義	備註
'\n'	換行字元	
'\t'	Tab	
'\r'	迴車鍵	Windows 系統中以 \r\n 代表換行
'\''	單引號	
'\"'	雙引號	
'\0'	空字元	用來代表字串的結束
'\\'	倒斜線	倒斜線被用做跳脫字元，因此要用兩個倒斜線表示

表 41: 常用特殊字元

常用技巧：字元判斷 在 ASCII 碼中，有三個區塊是連續的：

- 數字區塊：'0' 到 '9'
- 大寫字母區塊：'A' 到 'Z'
- 小寫字母區塊：'a' 到 'z'

因為 `char` 實際上是存整數，所以可以用大於小於來判斷，程式碼 49 可以判斷一個字元是否是數字字元。

```

1 bool myIsDigit(char ch) {
2     return '0' <= ch && ch <= '9';
3 }
```

程式碼 49: 判斷數字字元

函式	範圍	意義
<code>int isalnum(int)</code>	A 到 Z、a 到 z、0 到 9	判斷是否為英文字母或數字字元
<code>int isalpha(int)</code>	A 到 Z、a 到 z	判斷是否為英文字母
<code>int isdigit(int)</code>	0 到 9	判斷是否為數字字元
<code>int islower(int)</code>	a 到 z	判斷是否為小寫字母
<code>int isupper(int)</code>	A 到 Z	判斷是否為大寫字母

表 42: `<cctype>` 常用函式

在 `<cctype>` 中有一些可以判斷字元類型的函式，常用的如表 42，留意這些函式傳進去的參數型態就直接是 `int`。

不過臨時記不得這些函式，筆者建議自己寫一個，不難寫。

常用技巧：計算數字 用同樣的概念，可以把一個數字字元「轉換」成 `int` 的 0。字元 `'0'` 的編號為 48，因此我們將 `'0' - 48` 就可以取得實際的 `int` 值，但這個方法稍嫌笨重，因為我們必須記得 `'0'` 的編號，在 C++ 中有提供字元相減的方法，於是我們可以寫為程式碼 50，將一個數字字元減去 `'0'`。

```

1 int charToNumber(char ch) {
2     return ch - '0';
3 }

```

程式碼 50: 數字字元轉換為 `int`

同樣的方法也可以用在具有連續區間的大寫字母、小寫字母。

(二) C++ 字串與 C 字串

C++ 的字串為 `string` 物件，需要引入標頭檔 `<string>`。這裡要講 C 字串，C 字串事實上就是 **字元陣列**，宣告如程式碼 51。

```

1 char str[110];

```

程式碼 51: C 字串宣告

使用 C 語言的字串時，就有很多東西要注意：第一、字串本身是用 `'\0'` 來當結尾，**注意**！不是阿拉伯數字的 `'0'` 而是 ASCII 碼為 0 的 `'\0'` (表 41)，代表「字串的結尾」。

```
1 char str[10] = "bird";
```

程式碼 52: C 字串

例如程式碼 52 的字串為 `"bird"`，事實上儲存情形如圖 14。

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
b	i	r	d	\0	?	?	?	?	?

圖 14: 字串就是字元陣列

因為 `'\0'` 被當作「字串的結尾」(很重要所以還要再說一次)，所有 C 字串函數的操作都會跟這個有關。

(三) C 字串函數

C 字串函數都在 `<cstring>` 底下，以下解說幾個常用的 C 字串函數，如程式碼 53。

```
1 size_t strlen(const char *str);
2 int      strcmp(const char *str1, const char *str2);
3 char*    strcpy(char* dest, const char* src);
4 char*    strncpy(char* dest, const char* src, size_t num);
5 char*    strcat(char* dest, const char* src);
6 char*    strncat(char* dest, const char* src, size_t num);
```

程式碼 53: 常用的字串函數

strlen 函數 `strlen` 函數可以知道一個 C 字串的長度，參數就是傳一個字串指標，這個函數就是從一開始的位址往後掃，直到碰到 `'\0'` 為止，並回傳長度值。

程式碼 54 回傳的長度值為 4。當然我們也可以對字串變數做 `strlen`，程式碼 55 中 `num` 字串的長度為 10。

```
1 strlen("bird");
```

程式碼 54: strlen 範例

```
1 char num[110] = "0123456789";
2 strlen(num);
```

程式碼 55: strlen 範例

此外，strlen 是**函式**，它和 C++ 字串的 str.size() 不同的地方在於：每次呼叫 strlen 就會重新計算一次字串長度，像程式碼 56a 的寫法非常浪費時間，每次迴圈都會重新呼叫一次 strlen。

<pre>1 for (i = 0; i < strlen(str); i++) 2 { 3 // do something 4 }</pre>	<pre>1 int len = strlen(str); 2 for (i = 0; i < len; i++) 3 { 4 // do something 5 }</pre>
---	--

(a) 直觀寫法

(b) 較好的寫法

程式碼 56: 注意 strlen 的用法

但很多時候字串的長度沒有改變，因此會浪費很多時間，比較好的做法是用一個變數儲存長度，再下去執行迴圈，如程式碼 56b。

補充，雖然現代的編譯器大多會去辨別這一情況加以優化，但盡量不要依賴編譯器，最好從一開始養成好習慣，才不會被坑。

strcmp 函數 strcmp 函數用來比較兩個 C 字串，參數為兩個字串指標。它會從兩個字串一開始的字元逐個比較，比較到**其中一個**為 '\0' 為止，strcmp 的功能和 C++ 字串之間用 ==、>、< 來比較的功能類似。回傳值分成三類：

- 等於 0：代表兩個字串相等。
- 大於 0：代表 str1 的字典順序大於 str2，通常是回傳 1。
- 小於 0：代表 str1 的字典順序小於 str2，通常是回傳 -1。

strcpy 函數 strcpy 函數就是把 src 字串複製到 dest 字串，複製的原理和上述函式類似：就是從第一個字元開始複製，直到遇到 '\0' 為止。接著對 dest 最尾端補 '\0'。strcpy 相當於在 C++ 中 dest = src。

使用 strcpy 時需要特別**注意**，strcpy 函數**不會檢查** dest 的長度，換句話說，如果 src 比 dest 長，strcpy 會持續複製直到複製完為止，這對程式而言是非常危險的，因為寫出字串的範圍很有可能會寫到很重要的記憶體，然後**電腦就炸了**。

其他較為安全的替代方案，就是使用 strncpy 函數，strncpy 的第三個參數代表最多複製幾個字元，不過要注意 strncpy 只負責複製字元，複製完後**不會補** '\0'。

strcat 函數 strcat 函數主要是把 src 字串接到 dest 字串後面，它是從 dest 字串的第一個 '\0' 開始串接，因此也和 strcpy 一樣可能會造成寫出記憶體的問題，對應的函數是 strncat。strcat 函數相當於 C++ 字串中，直接做 dest += src。

其它有關字串的函數還有：strstr、strchr、strrchr、strtok、strspn 等，這些函數在此不贅述。

(四) 字串轉換

實作上，C++ 字串操作上來得 C 字串**容易**，但因為 C++ 字串是物件，因此操作 C 字串會比 C++ 字串來得快。我們可以選擇混合使用這兩種字串，在不難處理又快速時我們選擇 C 字串；而在 C 字串比較難解決的事情時，我們可以利用方便的 C++ 字串來處理。

C++ 字串轉 C 字串 如果要將一個 C++ 字串轉成 C 字串，我們利用 <string> 中的一個成員函式 c_str()。

```
1 string str = "bird";  
2 cout << str.c_str() << endl;
```

程式碼 57: C++ 字串轉 C 字串

要注意的是，c_str() 產生一個**唯讀**的 C 字串。

C 字串轉 C++ 字串 正確將 C 字串轉成 C++ 字串的方法則是將 C 字串丟進 C++ 字串中，如程式碼 58。

```

1 char str1[110] = "bird";
2 string str2 = str1;
3 cout << str2.size() << endl;
4 cout << str2 << endl;

```

程式碼 58: C 字串轉 C++ 字串

(五) 字串練習

可能有些人看完上面的敘述後，可能還不是很理解這些函數的用途。要理解 C 字串的用法不難，但如果要完全掌握住這些函數，還需要配合指標的觀念，以下問題，筆者就不寫上解答，留給讀者思考、討論。

1. 假設現在有一個字串

```

1 char str1[110] = "abcdefghijklmnopqrstuvwxyz";

```

- (a) 它的長度為何？
- (b) 用 `sizeof` 和 `strlen` 有何不同呢？
- (c) 請問下面的語句和上面又有什麼差異呢？

```

1 strlen("abcdefghijklmnopqrstuvwxyz");
2 sizeof("abcdefghijklmnopqrstuvwxyz");

```

2. 有一字串

```

1 char str2[110] = "bird";
2 char str3[4];

```

- (a) 我們可以用 `strcpy(str3, str2)` 嘛？
- (b) 如果不行，我們使用 `strncpy(str3, str2, 4)`；來避免超出記憶體呢？試著印出來觀察看看。

3. 有一字串

- (a) 這個字串的長度為何？


```

1 char str4[110] = "cat\0bird";

1 cout << str4 << "□" << str4 + 4 << endl;

```

(b) `strlen(str4 + 4)` 的回傳值為何？`strlen(str4 + 8)` 呢？

(c) 若我們使用 `strcpy(str4, "dog")`，會得到什麼結果？此時執行下面語句會有什麼反應呢？

(d) 若我們對原先的 `str4` 執行 `strcat(str4, "dog")`，有得到你預期的結果嘛？

(e) 要是 `strcat(str4 + 4, "dog")` 呢？

4. 假設有一 C++ 字串和一 C 字串

```

1 string str5 = "bird";
2 char str6[110];

```

要怎樣將 `str5` 複製給 `str6` 呢？

5. 現有兩字串

```

1 char str7[110] = "cat\0bird";
2 char str8[110];

```

試比較 `strcpy(str8, str7);` 和 `memcpy(str8, str7, sizeof(str7));` 的不同。

二、輸入與輸出

(一) `scanf` 和 `printf`

C++ 中常使用的輸入輸出是 `cin` 和 `cout`，有時候 `cin` 和 `cout` 的速度並不能滿足我們的需求，這時候就需要使用 C 語言本身的輸入輸出。

C 語言的輸入輸出在 `<cstdio>` 內，輸入的函數為 `scanf`，而輸出的函數為 `printf`。

注意！C 語言輸入輸出是函式，這兩個函式的用法和原本 C++ 的相比較為繁瑣，但也有比較方便的地方。

```

1  int  scanf(const char *format, ...);
2  int  printf(const char *format, ...);

```

程式碼 59: scanf 和 printf

scanf 和 printf 的參數中，後面 ... 稱為不定參數，代表參數的數量是不固定的，決定參數的數量是靠前面的 format 字串決定，這個字串我們稱為**格式字串**。

格式字串可以像我們平常輸出字串一樣作法，如程式碼 60。

```

1  printf("Hello_world!\n");

```

程式碼 60: 輸出字串

記得 endl 屬於 C++ 當中的換行，在此須使用換行字元 '\n' 來換行。輸出整數變數，在格式字串中我們用 "%d" 來代表，每個 "%d" 都代表著一個整數，如程式碼 61。

```

1  printf("%d\n", 3);      1  int  x = 3;
                           2  printf("%d\n", x);
(a) 印出常數              (b) 印出 int 變數

```

程式碼 61: 印出整數

程式碼 61a 直接印出數字，也可以寫為 printf("3\n");，程式碼 61b 會印出變數 x 的值，印出的位置在 "%d" 處。

輸入的話，因為我們是呼叫函數，若要改到變數值就需要使用傳址呼叫，因此輸入一個整數寫為程式碼 62。

```

1  int  x;
2  scanf("%d", &x);

```

程式碼 62: 輸入一個整數

同樣地，unsigned int、long long、unsigned long long、float、double 都有對應的格式，如表 43。


```

1 char str[5];
2 scanf_s("%s", str, 4);

```

程式碼 65: scanf_s 的範例

到目前為止，可看出 scanf 和 printf 在用法上比 cin 和 cout 繁瑣，除了在效能上的優勢外，還有什麼其他優點呢？

(二) 格式字串

進制輸出 在 C++ 中整數提供八進位、十進位、十六進位的輸出方法，分別是使用 std::oct、std::dec、std::hex 這三種，如程式碼 66a。

<pre> 1 int a = 11; 2 cout << oct << a << endl; 3 cout << dec << a << endl; 4 cout << hex << a << endl; </pre>	<pre> 1 int a = 11; 2 printf("%o\n", a); // 13 3 printf("%d\n", a); // 11 4 printf("%x\n", a); // b </pre>
--	--

(a) cout 版本

(b) printf 版本

程式碼 66: 輸出進制比較

程式碼 66b 是對應的版本，分別使用 "%o"、"%d"、"%x"。如要將十六進位印為大寫，cout 須加上 std::uppercase，printf 則使用 "%X"。

iomanip 格式 在 C++ 中，標頭檔 <iomanip>，專門做輸入輸出的處理。表 44 列出 <iomanip> 中常用的串流操縱符 (Stream manipulator)，也就是用來串在 cin、cout 的東西。

操縱符	作用
std::setprecision	設定精準度
std::setw	設定輸出寬度
std::setfill	設定填充字元

表 44: <iomanip> 常用操縱符

std::setprecision 通常是用來限定輸出數字的精準度，例如程式碼 67，setprecision 的參數為 5，因此會保留 5 位有效位數。

```

1 double f = 3.14159;
2 cout << setprecision(5) << f << endl; // 3.1415
3 cout << fixed << setprecision(5) << f << endl; // 3.14159

```

程式碼 67: 設定有效位數

當 `setprecision` 加上 `std::fixed` 的話，會變作印出小數點後 `n` 位，也就是四捨五入，如程式碼 67 第 3 行。

`std::setw` 代表輸出的寬度，例如程式碼 68，輸出的結果為 `" 16"`。

```

1 int a = 16;
2 cout << setw(5) << a << endl;

```

程式碼 68: 設定輸出寬度

`std::setfill` 可以設定空白處的填充字元，傳入的參數是一個字元，通常和 `std::setw` 混用，如程式碼 69。

```

1 cout << setw(5) << setfill('x') << 16 << endl;

```

程式碼 69: 設定填充字元

可以試試看以下程式碼，在此不贅述。

- `cout << setw(3) << 55688 << endl;`
- `cout << setfill('x') << setw(5) << left << 16 << endl;`
- `cout << setfill('x') << setw(5) << right << 16 << endl;`
- `cout << setfill('x') << setw(5) << internal << -16 << endl;`

printf 格式 `printf` 本身就有內建和 `<iomanip>` 相似的功能，比如我們可以設定輸出寬度、印出小數點後 `n` 位、填充前導零等，如程式碼 70。

scanf 格式 學習 `scanf` 最有價值的是他可以對輸入的格式做設定，比如要讀入一個時間的格式「hh:mm」，`cin` 一般需要多使用一個變數來讀入 `':'`，如程式碼 71a。

```

1 printf("%5d\n", 16); // 設定寬度
2 printf("%.3f", 3.14159); // 小數點後 3 位
3 printf("%05d\n", 16); // 前導 0，結果為 00016

```

程式碼 70: printf 格式範例

<pre> 1 int hh, mm; 2 char ch; 3 cin >> hh >> ch >> mm; </pre>	<pre> 1 int hh, mm; 2 scanf("%d:%d", &hh, &mm); </pre>
(a) 用 cin 輸入	(b) 用 scanf 輸入

程式碼 71: 比較 cin 與 scanf 差異

scanf 可以直接設定格式，省去多使用一個變數的麻煩，如程式碼 71b。

回傳值 scanf 也可以使用在 0 尾版、n 行版等迴圈中，不難，在此集中講解 EOF 版。

當 scanf 讀到檔尾時，會回傳一個常數 EOF (數值通常是 -1)，因此 EOF 版就會寫為程式碼 72。

```

1 while (scanf("%d", &n) != EOF) {
2     // do something ...
3 }

```

程式碼 72: EOF 版

(三) 標準 I/O

(四) 檔案 I/O

fscanf 和 fprintf

(五) 字串 I/O

sscanf 和 sprintf

stringstream C++ 中也有提供字串 I/O，稱為 stringstream 類別，在 <sstream> 標頭檔裡面，用法與 cin、cout 差不多，如程式碼 73。

```

1  stringstream ss;
2  string str;
3  ss << "Hello_world!"; // 將東西塞進 stringstream
4  ss >> str; // 丟到字串
5  cout << str << endl; // Hello world!

```

程式碼 73: stringstream 基本用法

程式碼 73 會印出 "Hello_world"! 字串，再來看看 stringstream 怎麼解決 sscanf 遇到迴圈無法解決的事，如程式碼 74。

```

1  int a;
2  stringstream ss;
3  string str = "1_2_3_4_5";
4  ss << str;
5  for (int i = 0; i < 5; i++) {
6      ss >> a;
7      cout << a << endl;
8  }

```

程式碼 74: 可以用迴圈來讀取

```

1  while (ss >> a) {
2      // do something
3  }

```

程式碼 75: stringstream 碰到 EOF

三、字串技巧

(一) 善用 index

(二) 回文

(三) 二維問題

(四) 子字串

(五) 其他

四、字串應用

(一) 羅馬數字

對大多數的人而言，這並不是一個陌生的主題。羅馬數字用一些特別的字母來當做某個數字，如表 45。

數字	1	5	10	50	100	500	1000
符號	I	V	X	L	C	D	M

表 45: 羅馬數字

羅馬數字系統遵守兩個原則：

- **加法原則**：透過累加符號遞增數字。例如：數字 1 寫為「I」、數字 2 寫為「II」、3 表示為「III」，以此類推。
- **減法原則**：為簡化書寫，4 不寫作「IIII」，而是用「5 - 1」寫為「IV」。

總結來說，兩種規則的區分如下：如果較小的數寫在較大的數的右邊，則為加法；反之則為減法。例如：11 為 $10 + 1$ ，表示為 XI；9 被當作 $10 - 1$ ，表示為 IX。

除此之外，還有兩個小細節：

- **統一書寫規則**：羅馬人規定個位數由個位數決定、十位數由十位數決定，以此類推。例如，99 雖然可以視為 $100 - 1$ ，但羅馬數字會統一看做 $90 + 9$ ，也就是 XC (90) 和 IX (9)，寫為 XCIX。
- **符號不超過三個**：4 會寫為 IV 而非 IIII，9 被寫為 IX 而非 XIII。因此，不管怎麼湊，羅馬數字會在 3999 以內（在此不討論更大數的表示法）。

阿拉伯數字轉羅馬數字 羅馬數字之中，個位數的符號表示個位數、十位數用來表示十位數，這之間不會混用，因此我們用「**位數**」來觀察較合適。在此我們可以用表 46 去觀察出個位、十位、百位的規律：

數字	1	2	3	4	5	6	7	8	9
符號	I	II	III	IV	V	VI	VII	VIII	IX
數字	10	20	30	40	50	60	70	80	90
符號	X	XX	XXX	XL	L	LX	LXX	LXXX	XC
數字	100	200	300	400	500	600	700	800	900
符號	C	CC	CCC	CD	D	DC	DCC	DCCC	CM

表 46: 羅馬數字找規律

可以得到以下結果：

- 個位、十位、百位之間符號不同，但「**格式**」相同，也就是只差在 IXC 用 VLD、XCM 來代換。既然格式相同，我們就可以試圖用迴圈簡化其結果。
- 若只觀察個位數，可發現 1、2、3 的符號恰好是重複符號。除此之外，1、2、3 和 6、7、8 之間只差一個「5」的符號。這代表我們可以用 **if** 把「5」的符號判斷掉，扣掉後用 1、2、3 的方式處理。
- 4 和 9 如果要取巧可能會比較困難，平常練習可以思考如何修改，若思考時間不夠建議直接用特殊判斷處理掉。

根據上面的討論，我們利用迴圈判斷的同時，我們需要利用陣列來儲存我們想要的東西，程式碼 76 是一個很簡便的實現方法。

```

1  #define L 3
2  string Nine[L] = {"CM", "XC", "IX"};
3  string Four[L] = {"CD", "XL", "IV"};
4  string Five[L] = {"D", "L", "V"};
5  string One[L] = {"C", "X", "I"};
6  int Mod[L] = {100, 10, 1};
7  int Mod5[L] = {500, 50, 5};

```

程式碼 76: 儲存符號、餘數

程式碼 76 簡單紀錄每一次判斷的餘數 (Mod 和 Mod5 陣列)，並且會對應到 Five、One 這兩個陣列做處理。此外，Nine、Four 陣列是針對 4 和 9 直接例外處理。

```
1 string roman = "";
2 for (;n >= 1000 && n; n -= 1000)
3     roman += "M"; // 千位數例外判斷
4 for (int i = 0; i < L; i++) { // 百位、十位、個位數
5     if (n / Mod[i] == 9) // 特殊判斷 9
6         roman += Nine[i], n -= 9 * Mod[i];
7     if (n / Mod[i] == 4) // 特殊判斷 4
8         roman += Four[i], n -= 4 * Mod[i];
9     while (n >= Mod5[i]) // 判斷 5 以上
10        roman += Five[i], n -= Mod5[i];
11    while (n >= Mod[i]) // 剩下的部分
12        roman += One[i], n -= Mod[i];
13 }
```

程式碼 77: 阿拉伯數字轉羅馬數字

程式碼 77 是對應的處理方法，可以看到第 1 行針對千位數做判斷，在迴圈中第 5 行、第 7 行也是根據先前的討論，優先處理 4 和 9 的情形 (可以想一下這兩個判斷為什麼不能反過來)。

羅馬數字轉阿拉伯數字 至於如何把羅馬數字換回阿拉伯數字？由剛剛的表格知道，除了 4 和 9 的羅馬數字是兩個字元 (由於減法規則)，其他都可以拆成一個字元來看待，因此我們可以藉由**記錄目前**掃到的羅馬數字，來判斷下個羅馬數字該使用加法規則還是減法規則。

(二) 字串和數字轉換

字串轉數字 一開始有提到，把一個數字字元轉換成數字的方法，若現在有一個數字字串，例如 "123"，要怎麼做呢？

程式碼 50 的概念可以繼續延伸，我們可以對每個字元減掉 '0' 之後再乘上對應的值，最後加總就會轉換成對應數字。

$$123 = 1 \times 10^2 + 2 \times 10^1 + 3 \times 10^0$$

$$= ('1' - '0') \times 10^2 + ('2' - '0') \times 10^1 + ('3' - '0') \times 10^0$$

實作上我們會從高位的字元往後做，以節省計算 10^n 的時間。

```

1  int MyAtoi(string str)
2  {
3      int res = 0, i;
4      for (i = 0; i < str.size(); i++)
5      {
6          res *= 10;
7          res += str[i] - '0';
8      }
9      return res;
10 }
```

程式碼 78: 從高位數開始做

如程式碼 78，我們把剛剛的算式做轉換，可以得到如下算式：

$$123 = ((0 \times 10 + 1) \times 10 + 2) \times 10 + 3$$

除此之外，也有些人會從個位數開始做，只是這樣就要額外變數來紀錄 10 的次方。

數字轉字串

(三) 進位變換

(四) 習題

索引

1's 補數, 21

ASCII, 50

void, 44

一元運算子, 21

串流操縱符, 60

二元運算, 10

結合性, 11

運算元, 10

運算子, 10

運算順序, 11

位元, 16, 36

位元組, 17, 36

位元運算子, 16

位址, 37

取值運算子, 40

取址運算子, 37

可視範圍, 46

回傳值, 13

型別轉換, 33

型態, 6

堆疊, 46

堆積, 46

字元, 50

宣告, 4, 43

布林值, 7

整數除法, 11

未定義行為, 2, 30, 34

格式字串, 58

標頭檔

cctype, 52

cfloat, 36

climits, 36

cstdio, 57

cstring, 44, 53

iomanip, 60, 61

sstream, 62

string, 52, 55

浮點數除法, 11

溢位, 16, 18, 20, 35

科學記號, 9

編譯錯誤, 16

記憶體, 36

變數, 4

資料型態, 7

布林值, 7

賦值, 6

輸入

多變數輸入, 7

逗號運算子, 7, 33

運算, 32

邏輯運算子, 14

短路運算, 15

除零問題, 12