基礎程式設計技巧

許胖

2017 年 4 月 11 日

目 錄

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，爲什麼會執行
   (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、嵌入式系統

資訊領域中有很多子領域，此外，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 章

程式與計算

本節目標

- 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**。

第一節　程式架構

一、 C++ 基本架構

最基礎的 C++ 架構如下：

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

程式碼 1.1: C++ 基本架構

怎麼理解呢？先不要理解去**記**起來，之後會慢慢帶出這些字的意義。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？

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

程式碼 1.2: 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 1.2 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」。

接下來有一些 C++ 的特性你必須知道：

1. 如果將程式碼 1.2 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：

```
1 int x;
```

程式碼 1.3: 宣告變數

程式碼 1.3 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 1.1。

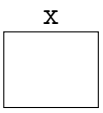


圖 1.1: 容器

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 1.4: 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 1.4 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 1.4 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 1.5a，更可以簡化成如程式碼 1.5b。

```
1 int a;
2 int b;
```

(a) 兩個宣告

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

(b) 簡化版

程式碼 1.5: 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 1.6：

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

程式碼 1.6: 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 1.7。

```
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 1.7: 變數不初始化，會發生什麼事呢？

程式碼 1.7 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。

圖 1.2: 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 1.8: 輸入

大家可以試著執行看程式碼 1.8，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 1.9: 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 1.1 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 1.1: 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 1.10：

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 1.11。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 1.10: 布林值宣告

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

程式碼 1.11: 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 1.12 把整數 5 裝進整數變數 x 中：

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

程式碼 1.12: 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 1.13: 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 1.14，看看和預期的有什麼不同。

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

程式碼 1.14: 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 1.15: 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 ⇒ 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 ⇒ 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}

第二節　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 1.2: 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 2.1. 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」*)*。

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 1.2，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 1.2 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
$=\mathbf{7-4}$	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 $(n \times 567 \div 9 + 7492) \times 235 \div 47 - 498$

第三節　比較和邏輯運算子

一、 比較運算子

比較運算子如表 1.3：

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 1.3: 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 1.4: 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。

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 1.5: 邏輯運算子

我們可以採用回傳值的觀點，從表 1.5 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：
- 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

```cpp
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 1.16: 範例

```cpp
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 1.17: 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。

第四節　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 1.6: 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 1.7: 位元組長度

表 1.7 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` 爲 4 個位元組、`long long` 爲 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 1.8: `int` 的位元組

可以視爲一個長度是 32 的二進位數字，我們將位數依照高低編號，如表 1.9，x_{31} 表示正負號，若 x_{31} 爲 0 代表此 `int` 是正數，反之則爲負數。

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

表 1.9: 二進位 int

因爲 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 1.10: 1 的二進位表示法

當 int x = 255; 時如表 1.11。

00000000	00000000	00000000	11111111

表 1.11: 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 1.12: 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。

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 1.13: 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 << 2 ⇒ 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 1.14: 左移的情況

類似的情況，5 >> 1 ⇒ 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 1.15: 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 1.16: 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爲 1，相同爲 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	00000101
&	00000000	00000000	00000000	00000011
	00000000	00000000	00000000	00000001

表 1.17: 5 & 3 的狀況

5 & 3 結果會是 1：

5 | 3 結果會是 7：

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

表 1.18: 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 1.19: 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 1.20: 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 1.21: 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 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 1.16 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 1.16a 和 1.16c 的規則會如下：

表 1.22: 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	0000000**1**
	00000000	00000000	00000000	0000000x_0

表 1.23: 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`：

	$x_{31}x_{30}\cdots x_{24}$	$x_{23}x_{22}\cdots x_{16}$	$x_{15}x_{14}\cdots x_8$	$x_7x_6\cdots x_0$
|	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 1.24: 將 x_0 改爲 `1`

根據表 1.22b，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。

	$x_{31}x_{30}\cdots x_{24}$	$x_{23}x_{22}\cdots x_{16}$	$x_{15}x_{14}\cdots x_8$	$x_7\cdots x_3x_2x_1x_0$
|	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 1.25: 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 1.16a 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	1111111**0**
	$x_{31}x_{30}\cdots x_{24}$	$x_{23}x_{22}\cdots x_{16}$	$x_{15}x_{14}\cdots x_{8}$	$x_{7}x_{6}\cdots x_{1}$**0**

表 1.26: 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 1.26 可以寫爲程式碼 1.18。

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

程式碼 1.18: 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 1.27，可以看出二進位下有兩個 1，因此結果爲 2。

00000000	00000000	00000000	00000101

表 1.27: 5 的 parity

- Parity(255) 如表 1.28，結果爲 8。

00000000	00000000	00000000	11111111

表 1.28: 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 1.19: 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 1.20: Parity 位元運算寫法

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：

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

程式碼 1.21: 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 1.16b 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 1.29: 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 1.29 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：

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

程式碼 1.22: 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 1.23：

最後，我們來看看程式碼 1.24 怎麼運作的：

表 1.30 表示程式碼 1.24 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 ^ x ^ y 有兩個 y，會抵消爲 0，又 0 ^ x ⇒ 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。

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

程式碼 1.23: 變數版

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

程式碼 1.24: 位元運算版

	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 1.30: 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。

第五節　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 1.31: 指定運算子

表 1.32 複合指定運算子代表的意義如下，不難理解：

- x += a ⇒ x = x + a

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 1.32: 複合指定運算子——算術運算子

- `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 1.33: 複合指定運算子——位元運算子

同樣地，表 1.33 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 1.34: 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 1.25，猜猜答案是什麼？

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

程式碼 1.25: 未定義行爲

答案會是 2 嗎？根據運算順序 (表 1.34 和表 1.2)，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。

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：

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

程式碼 1.26: 未定義行爲

不難看出，C++ 雖然規定了運算順序，但程式碼 1.26 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。

第六節　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 1.35: 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 1.27 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 1.27: 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 1.7，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：

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

程式碼 1.28: 迴圈輸入

程式碼 1.28 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。

第七節　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。

第 二 章

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**。

本章目標

- 了解記憶體與指標的概念
- 利用選擇結構和迴圈結構
- 物件導向的觀念

第一節　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 2.1: 產生溢位的程式碼

上一節提到 int 的定義爲「至少 2 個位元組」，若讀者的 int 也是 4 個位元組的話，那麼就會得到 -2147483648！這種現象我們稱爲**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 2.1: 2147483647

表 2.1 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 2.2: 2147483647 + 1

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

大體上，表示資料的記憶體大小是**有限**的，那麼就會有以下兩種事情發生：

- 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。這些可能性切一半，2^{31} 個表示負整數，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 $2^{31} - 1$。
- **精確度**的限制。資料型態 double 是以 IEEE 754 爲標準，有 8 個位元組，最多只能表示 2^{64} 種小數。因爲在標準中，以 53 個位元儲存尾數，故有 52 位有效數字，精確度爲 $\log 2^{52} \approx 15.95$ 位十進位有效數字 (float 則爲 7 位)。

表 2.3 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數。

此外，需要注意的有幾點：

- int 的上限是 2147483647，用十六進位表示爲 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 <cfloat> 當中查詢到，實際範圍會依據不同計算機而有差異，查詢的方法自行 google，這裡不贅述。

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 2.3: 資料型態上下界

二、 位址

(一) 記憶體

上一節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**的部分。首先，我們常常提到的記憶體有分廣義和狹義，廣義的記憶體可以指稱所有儲存資料的設備，表 2.4 列出計算機中常用的儲存設備：

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 2.4: 廣義的記憶體，又稱爲**記憶體階層** (2016 年資料)

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。

(二) 位址概念

主記憶體可以看做是一條很長、連續的位元組，程式執行時，會佔據其中一個區塊，如圖 2.1：

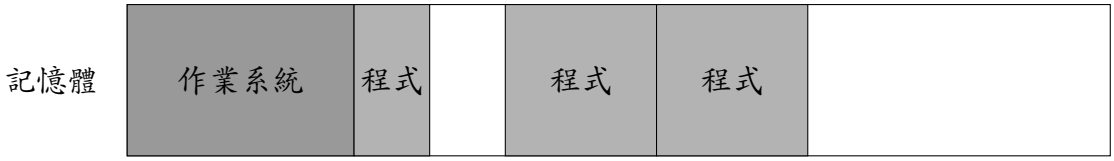


圖 2.1: 記憶體與程式

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 (當成 int、double 等)。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**」。

位元組是擁有地址的**最小單位**，單個位元並沒有位址。

(三) 取址運算子

位址是一個數字，通常以十六進位表示，如果要知道一個變數的位址，可以使用取址運算子 &，例如程式碼 2.2：

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

程式碼 2.2: 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 2.2：

這個結果會因爲不同機器、每次程式執行分配的記憶體而不同 (總之就是**不一定**啦！(ノ ° □°) ノ ⁀ ┻┻)，但概念是相同的。

(四) 大字節序和小字節序

但是要怎麼知道變數 a 實際怎麼儲存 16 呢？很多人會以爲像是圖 2.3：

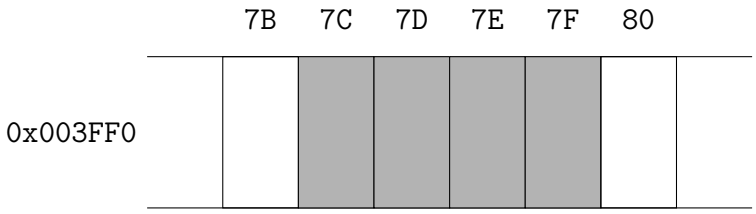


圖 2.2: 變數 a 實際的記憶體位置

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

圖 2.3: 大字節序儲存方法

但這個說法不完全對，圖 2.3 的儲存方法被稱爲「**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爲「**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。用整數 0x12345678 來表示這兩種儲存方法，差異如圖 2.4a 及 2.4b：

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

(a) 以大字節序儲存

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

(b) 以小字節序儲存

圖 2.4: 0x12345678 不同儲存方法

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。

三、 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 2.5。

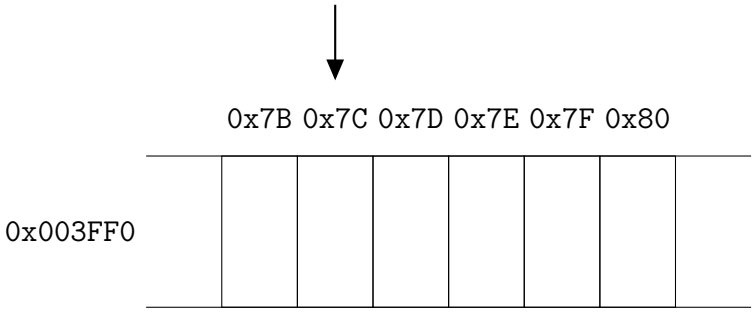


圖 2.5: 指標概念

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。

(一) 宣告

首先，程式碼 2.3 宣告一個**指標變數** ptr。

```
1 int *ptr;
```

程式碼 2.3: 宣告指標變數 ptr

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 的資料型態爲 int*，代表這是一個指向 int 的指標。因爲資料型態是 int*，所以也可用程式碼 2.4 的方式來宣告。

```
1 int* ptr;
```

程式碼 2.4: 宣告指標變數 ptr

值得注意的點是，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 2.5。

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

程式碼 2.5: 宣告多個指標變數

(二) 賦值

剛剛說過，指標的運作是讓指標變數儲存**位址**，如果以之前變數 a 的例子來講，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 2.6。

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

程式碼 2.6: 指標的賦值

程式碼 2.6 的實際狀況如圖 2.6。

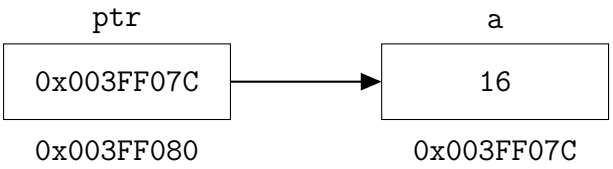


圖 2.6: 程式碼 2.6 的狀況

由圖 2.6 可以看出以下幾個重點：

- 指標只是一個概念，實際上用變數來代表，既然 C++ 的指標也是一個變數，那麼就需要另外配置記憶體。
- C++ 實做指標，就是**儲存位址**。

(三) 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 2.7 爲例。

程式碼 2.7 中，第 4 行的 * 是取值運算子 (這符號容易和指標宣告搞混)，回傳指向記憶體的值，因此會印出「16」。

在第 5 行中，因爲 C++ 的指標也是一變數，因此會將 b 的位址儲存在 ptr 裡面，意義是指向變數 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 2.7: 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 的指標型態爲 int*、指向 double 的指標型態爲 double*，以此類推。以下情況由讀者做觀察，想想爲什麼會有這些現象，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 2.8，這一章節主要是讓大家能夠了解指標的概念，並非以熟練指標爲主：

```
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 2.8: 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 2.9。

在程式碼 2.9 中，第 4 行會印出「17」，因爲第 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 所指的值做累加」，讀者可以比較 ptr++、*ptr++ 與 (*ptr)++ 的不同。

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

程式碼 2.9: 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 指標的型態爲 int*，如果是指向 int* 的指標，則型態爲 int**，用法和普通的指標相似，程式碼 2.10 展示它的用法。

```
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 2.10: 指標的指標

讀者可以參考圖 2.7，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 2.10 連續運用取址運算子和取值運算子，會有什麼結果呢？

- **tmp 的值爲何？
- *&ptr 和 &*ptr 有什麼不同？

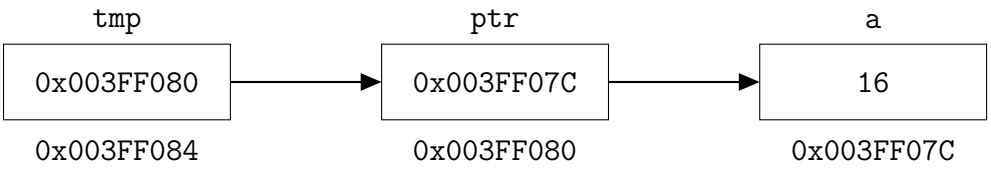


圖 2.7: 程式碼 2.10 的狀況

- &&a 可以運作嗎？

(四) 資料型態

程式碼 2.11 展示了指標型態的用途，這個例子比較複雜，在第 2 行時，型態爲 char* 的指標 ptr「刻意」去接 x 的位址，但由於 x 位址的型態爲 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 2.11: 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**」，土地可以規劃爲住宅用、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爲何，才能依照該有的格式去存取。程式碼 2.11 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 2.8。

爲了簡化描述，我們將變數 x 第一個位元組的位址稱爲 X，依序爲 X+1、X+2、X+3。當我們用 ptr 指向位址 X 時，因爲 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，也就是位址爲 X 所存的資料。

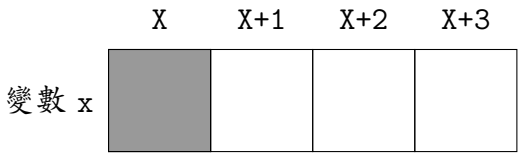


圖 2.8: 實際上 ptr 的有效範圍

然而最終答案會因爲不同電腦而有差異，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 儲存的數字爲 0x01，而小字節序會儲存 0x04。

(五) 空指標

在 C++ 中，爲了避免未初始化的指標指向未知記憶體，我們會用空指標**常數** NULL 來操作，嚴格來說，NULL 不是空指標，而要經過型別轉換爲指標型態後，也就是 (int*)NULL 之類的操作，才會變爲空指標。

```
1 #include <cstddef>
2
3 int* x = NULL;
4 int* y = (int*)NULL;
5 int* z = nullptr;
```

程式碼 2.12: 空指標用法

程式碼 2.12，示範了清空指標的方法。在 C++11 中，標頭檔 <cstddef> 提供了空指標 nullptr 可供使用。

四、 記憶體操作

這一段介紹 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 2.13: 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：

- 當回傳型態爲 void 時，代表這個函式「**沒有回傳值**」；
- 當回傳型態爲一個 void* 指標型態時，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。

(一) 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 2.14: 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 2.14，猜猜變數 x 會是多少呢？哈，答案並不是「1」！

剛剛提到，ptr 只有單純指向記憶體，既然是視爲記憶體。那它就是**一個位元組一個位元組**依序修改，因此程式碼 2.14 的結果會像圖 2.9。

00000001	00000001	00000001	00000001

圖 2.9: 程式碼 2.14 得到的結果

由此可知：

- 因爲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。常用的值有兩個：0 和 -1 (255)，因爲這兩個數字在二進位下代表全 0 和全 1，因此可以把數字都設爲 0 和 -1，讀者不妨驗證一下。
- 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。

(二) 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 2.15 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 2.15: memcpy 用法

最後，memcpy 會回傳 destination 指標。

第二節　程式控制

程式語言中，在語法上會設計一些用來方便表達我們思想的工具，這些工具經過幾十年的演變後，歸納出大部分程式語言會有的特性，以下介紹幾個 C++ 當中基本但重要的工具。

一、 程式區塊

C++ 中大括號被稱爲程式區塊，用以包住多個程式敘述。試試看程式碼 2.16 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 }
```

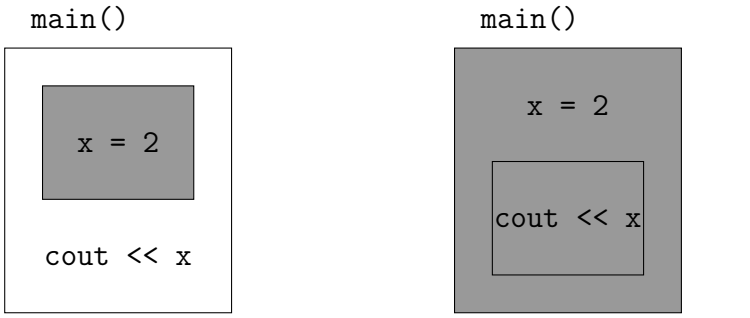
(a) 被大括號包住的 x

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

(b) 另一個例子

程式碼 2.16: 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**做爲區隔。例如程式碼 2.16a 中，變數 x 被包含在大括號中，狀況如圖 2.10a。



(a) 2.16a 變數 x 的可視範圍　　(b) 2.16b 變數 x 的可視範圍

圖 2.10: 程式碼 2.16 的狀況

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 2.16a 的 cout 沒辦法看見包在大括號的變數 x。

程式碼 2.16b 展示另外一個例子，結果如圖 2.10b，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。

詳細的可視範圍觀念留在之後說明。

二、 選擇結構

(一) if 結構

選擇結構在 C++ 中就是 if。if 最簡單的語法如程式碼 2.17。

```
1 int a = 4;
2 if (a < 10)
3   cout << a << endl;
```

(a) 單行指令

```
1 int a = 4;
2 if (a < 10) {
3   a += 5;
4   cout << a << endl;
5 }
```

(b) 程式區塊

程式碼 2.17: if 的用法

綜觀兩種情況，在 if 後面的小括號放**邏輯運算式**，只要邏輯運算式爲 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 2.17a 第 3 行，和程式碼 2.17b 第 3 行至第 5 行會被執行。

(二) if-else 結構

條件判斷更可進階爲 if-else 結構，語法如程式碼 2.18。if-else 做的是：當邏輯運算式的結果爲 true，執行 if 的區塊；如果是 false，則執行 else 區塊。

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

程式碼 2.18: if-else 結構

同樣地，else 也可以改爲程式區塊。當你要判斷的條件比較多時，if-else 可以連用，如程式碼 2.19。

```
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 2.19: if 和 else 連用

程式碼 2.19 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。

(三) 懸置 else 問題

將程式碼 2.20a 拔掉大括號，變成程式碼 2.20b，會有什麼差別？

```cpp
if (0) {
  if (0) cout << "QQ" << endl;
}
else cout << "XD" << endl;
```

(a) 危險的 else

```cpp
if (0)
  if (0) cout << "QQ" << endl;
else cout << "XD" << endl;
```

(b) 編譯器會不知道是哪一個 if 的 else

程式碼 2.20: 懸置的 else

不僅僅是程式敘述以分號結尾，實際上編譯器也無法得知 else 是和哪一個 if 匹配，因此在沒括大括號的情況下，最後一個 else 通常會匹配到最近的 if，讀者可以驗證這兩段程式碼的不同。

三、 迴圈結構

(一) 簡介

C++ 中常用的迴圈結構有三種：while、do ... while 和 for 迴圈，原理都是反覆檢查一個判斷式，如果結果爲 true 則執行對應程式區塊；直到判斷式爲 false，則跳出迴圈結構。

新手常見的錯誤是無法使判斷式變爲 false 導致無法跳出迴圈，或是寫出一個判斷式爲 false 的迴圈，使程式不會執行到迴圈內部。以下比較這些迴圈結構來說明。

```cpp
for (int i = 0; i < 10; i++) {
  cout << i << endl;
}
```

(a) for 語法

```cpp
{
  int i = 0;
  while (i < 10) {
    cout << i << endl;
    i++;
  }
}
```

(b) 對應的 while 語法

程式碼 2.21: for 和 while 的對應關係

程式碼 2.21 是 for 和 while 語法間的對應。從 while 的寫法可以看到小括號間是判斷式，除非 i < 10 爲 false，就會反覆執行程式區塊。在 while 迴圈中最後一

行，i++ 決定了是否能跳出迴圈，因爲在每次執行完迴圈時，i 值會遞增，直到不小於 10，判斷式變爲 false 因而跳出迴圈。

for 迴圈與 while 迴圈結構的對應中，要注意在 for 迴圈中宣告變數，效力相當於一個區塊變數，離開 for 迴圈後變數就會被回收。

do ... while 的特色是**後置判斷**，在至少須執行一次迴圈的場合可以使用，基本用法如程式碼 2.22。

```
1 int i = 0;
2 do {
3   cout << i << endl;
4 } while (i < 0);
```

程式碼 2.22: if 和 else 連用

(二) 應用：找極值

極值問題，通常是從 n 個元素間求最大值和最小值。

我們先從最簡單的狀況來考慮：兩個數字。給你兩個數字 a、b，找最大值就接用 if 下去判斷，如程式碼 2.23。

```
1 if (a < b)
2   cout << b << endl;
3 else
4   cout << a << endl;
```

程式碼 2.23: if 求最大值

內建函數的話，<iostream> 中有 max 函數和 min 函數，使用方法如程式碼 2.24。

```
1 cout << max(a, b) << endl;
```

程式碼 2.24: max 求最大值

現在考慮原本的問題，n 個數字下的極值，因爲 max、min 等函數只能對兩個數字做比較，若是用 if 去把所有可能做分類，如程式碼 2.25，除非是來不及寫正解的前提下，這類程式碼一來龐大、難以維護，一來如果出錯也不好 debug。

```
if (a < b < c)
  cout << c << endl;
else if (a < c < b)
  cout << b << endl;
else if (b < a < c)
  cout << c << endl;
else if (b < c < a)
  cout << a << endl;
else if (c < a < b)
  cout << b << endl;
else
  cout << a << endl;
```

程式碼 2.25: if 求 3 個數字最大值

如果我們善用迴圈和變數，迴圈可以走過所有數字，再利用一個變數 mx 去儲存前 i 個比較過後的最大值，迴圈開始前需要初始化這個變數，通常是第一個數字。

程式碼 2.26 實作了這個想法，筆者在迴圈當中使用 if 作爲判斷，如果新的數字比之前比較過的最大值還要大，就可以替換成更大的數字，迴圈最後會使得 mx 是 n 個數字中的最大值。

```
int mx = a[0]; // 陣列 a 是 n 個數字
for (int i = 1; i < n; i++)
  if (mx < a[i])
    mx = a[i];
cout << mx << endl;
```

程式碼 2.26: 求 n 個數字最大值

如果沒辦法初始化爲第一個數字時，那麼求最大值的初始值就讓他盡可能的小，使得往後讀到的第一個數字一定能取代他，求出來的最大值一定是在 n 個數字中，例

如：求極值的範圍是 int 的範圍時，可令最大值 mx = -2147483648，亦即 int 中最小的數；如果求極值的範圍是正浮點數，則令最大值爲隨便一個負數即可 mx = -1.0。類似的道理，求最小值的時候，令初始值盡可能的大即可。

(三) 應用：輸入測資

競賽中常見的輸入形式有三種：**EOF 版**、**0 尾版**、**n 行版**，其他少部分的形式在了解以下基本的輸入法和一些 IO 應用後都不難構造，因此要好好理解。

0 尾版　典型的 0 尾版是題目要求：「輸入以 $n = 0$ 結束。」我們可以這樣做：

```
while (cin >> n, n) {
  cout << n << endl;
}
```

程式碼 2.27: 0 尾版

程式碼 2.27 利用逗號運算子和比較運算子的簡化，當輸入爲零時，逗號右側的 n 會判定爲 false 而跳出迴圈。

類似 0 尾版的輸入可能是以特定的數字做爲結束 (常見的是 -1)，有時會以多個數字是否全零、是否爲特定字串作爲結尾，讀者可以自行練習。

n 行版　典型 n 行版的輸入爲：「第一行有一個整數 n，代表接下來的測試資料筆數。」因此在設計上，我們要優先讀入此整數，再處理各組測試資料。一種寫法如程式碼 2.28：

```
for (cin >> n; n; n--) {
  cout << n << endl;
}
```

程式碼 2.28: n 行版

程式碼 2.28 也是利用邏輯運算子簡化，每次執行完迴圈 n 就會遞減，直到第 n 次做完後，回頭檢查 n 值爲零而退出迴圈。

```cpp
for (cin >> n, cnt = 1; cnt <= n; cnt++) {
  cout << "Case␣" << cnt << ":" << endl;
}
```

程式碼 2.29: n 行版的另一種形式

另外一種常見形式是題目要求形如「"Case␣#:"」的輸出，這時就要使用一個變數去計算目前執行到的資料筆數，如程式碼 2.29。

EOF 版 當題目敘述提到：「以 EOF 結束」，或是沒有特別提到結束的方式，且在輸入中不是前面兩種形式結尾通常都是 EOF 版。

EOF 是 end-of-file 的簡寫，意思是輸入到檔案結尾，沒有測試資料就可結束。當 cin 讀取到檔案結尾時，會「知道」讀取到檔案結尾，因此設計上可寫爲程式碼 2.30。

```cpp
while (cin >> n) {
  cout << n << endl;
}
```

程式碼 2.30: EOF 版

在測試執行當中，如果要用鍵盤輸入檔案結尾，依據作業系統的不同而有差異，大致上是 Ctrl+Z 或是 Ctrl+D。

四、 陣列

(一) 簡介

注意！C++ 陣列 index 從 0 開始！

(二) 指標運算

試試看下面語句，解釋輸出結果：

- `*(&a[0] + 1)`
- `&a[1] - &a[0]`
- `(char *)&a[1] - (char *)&a[0]`

```
int a[5] = { 1, 4, 9, 16, 25 };
int *ptr = &a[1];
cout << ptr << endl;
cout << *ptr << endl;
cout << ptr + 1 << endl;
cout << *(ptr + 1) << endl;
```

程式碼 2.31: 指標加法

- `(long long *)&a[1] - (long long *)&a[0]`
- `(void *)&a[1] - (void *)&a[0]`

(三) 陣列指標

事實上，陣列本身是一種指標，例如程式碼 2.32，會印出陣列本身的位址，此種指標型態爲 `int(*)[5]`，可以把 `int[5]` 看成是一種資料型態，代表五個 `int`，`int(*)[5]` 是指向 `int[5]` 的指標，`[5]` 清楚表示指向的記憶體有五個 `int` 的長度，而與此對應的 `int*` 則沒有標示有多少個 `int` 的長度。

```
int a[5] = { 1, 4, 9, 16, 25 };
int *p = a; // int* 指向陣列 a
cout << a << endl;
cout << a + 1 << endl;  // a[1] 的位址
cout << &a + 1 << endl; // 指標運算跳過整個陣列
```

程式碼 2.32: 陣列指標

配合上面的指標運算，陣列的下標運算子 `a[x]` 實際上就等同於 `*(a + x)`。程式碼 2.32。

五、 函數

將陣列傳入函數，事實上是傳入陣列指標。

(一) 傳值呼叫

(二) 傳址呼叫

C 語言一開始的設計是用指標。

```cpp
void c8763(int x) {
  x++;
  return;
}
int main() {
  int a = 0;
  c8763(a);
  cout << a << endl; // 0
}
```

程式碼 2.33: 傳值呼叫

```cpp
void c8763(int *x) {
  (*x)++;
  return;
}
int main() {
  int a = 0;
  c8763(&a);
  cout << a << endl; // 1
}
```

程式碼 2.34: 傳址呼叫

(三) 傳參考呼叫

C++ 中，有另外一種傳參考呼叫，相當於是「**別名**」。

(四) 函數多載

(五) 函數指標

函數也有指標，例如對於傳兩個 `int` 參數、回傳值爲 `int` 的函數而言，其指標型態爲 `int(*)(int,int)`，在此以程式碼 2.36 簡單介紹，不贅述。

六、 C++ 物件導向

這一節主要提供一些 C++ 物件導向的方法，來增加競賽寫程式的速度及方便性，並不會提到太多物件導向的觀念，想要知道更多有關這方面的讀者可以在網路上搜尋。

```cpp
void c8763(int &x) { // 參考
  x++;
  return;
}
int main() {
  int a = 0;
  c8763(a);
  cout << a << endl; // 1
}
```

程式碼 2.35: 傳參考呼叫

```cpp
int f(int a, int b) { return a + b; }
int g(int a, int b) { return a - b; }
int h(int (*func)(int, int), int a, int b) {
  return func(a, b);
}
int main() {
  cout << h(f, 1, 2) << endl; // 3
  cout << h(g, 2, 1) << endl; // 1
}
```

程式碼 2.36: 函數指標

(一) 物件與類別

在物件導向的世界裡，我們把所有東西都視爲一個又一個的**物件 (Object)**，這些物件都有各自的**內部狀態**——也就是物件本身會儲存各式各樣的資料，而這些物件之間會相互影響，進而改變內部的狀態，這就是物件導向的核心概念。

物件有其**生命週期**，當物件被產生時，物件會呼叫**建構子 (Constructor)** 初始化，當物件的生命結束要被消滅時，物件會呼叫**解構子 (Destructor)**，在物件內的函式可以表達物件的行爲模式，以及如何影響其他物件，此時這些函式稱爲**方法 (Method)**。

物件導向中的物件，是由**類別 (Class)** 所產生，若類別 A 產生了一個物件 B，我們稱 B 是 A 的一個**實例 (Instance)**。類別好比是一張製作物件的**藍圖**，裡面記載創造的物件需要能夠儲存什麼資料，提供什麼方法等等。

當程式開始執行時，程式會先創造一塊記憶體來存放藍圖。創造實例時，創造出的物件會依據類別所包含的變數、方法來創造。在 C++ 中，struct 和 class 都被當成類別的一種，當我們要寫一個複數的類別時，可以寫如程式碼 2.37。

```
struct Complex {
  double real, imag;
};
```

(a) Complex 結構

```
class Complex {
public:
  double real, imag;
};
```

(b) Complex 類別

程式碼 2.37: 結構和類別的對應

要創造一個實例的話，其實就是**宣告變數** Complex c;。

(二) 建構子與解構子

上一節提到，物件被創造時會呼叫**建構子**，結束時會呼叫解構子，在此節我們把注意力放在建構子上。

通常變數宣告時，計算機並不會幫我們初始化變數，而變數的內容是**未知數**，因此要避免使用未經初始化的變數。在物件導向的世界中，創造一個物件時會呼叫建構子——它可以看做是類別的一種方法，一個很特殊的方法。

例如，我們想讓每個複數一開始都能被歸零，一種直觀的寫法如程式碼 2.38。

```
Complex c;
c.real = c.imag = 0.0;
```

程式碼 2.38: 初始化 Complex 物件

同樣功能如果要用建構子實作的話，一來每次宣告變數時會自動呼叫建構子初始化，二來在程式風格上同樣都是寫於類別內，比較不容易出現漏網之魚。

建構子特別之處在於，

- 建構子不需要寫回傳值
- 建構子名稱必爲類別名稱，因此 Complex() 即爲 Complex 類別的建構子

```
1 struct Complex {
2   double real, imag;
3   Complex() {
4     real = imag = 0.0;
5   }
6 };
```

程式碼 2.39: Complex 建構子

此外，建構子也可以多載，如果想要初始化成特定數字，我們可以多寫一個建構子，如程式碼 2.40。

```
1  struct Complex {
2    double real, imag;
3    Complex() {
4      real = imag = 0.0;
5    }
6    Complex(double r, double i) {
7      real = r;
8      imag = i;
9    }
10 };
11
12 Complex c1, c2(4.0,5.0);
```

程式碼 2.40: Complex 建構子多載

程式碼 2.40 會將 c1 的實部虛部初始化爲 0，而 c2 會呼叫另外一個建構子，將實部初始化爲 4.0，而虛部爲 5.0。

(三) 運算子多載

假設現在有兩個 Complex a,b，我們要把這兩個複數相加，可以寫作程式碼 2.41。

程式碼 2.41 第 1 行中，const 代表變數在此不會被改值，可以避免掉誤改的狀況。第 2 行呼叫建構子 2.40，創造一個新的 Complex 物件並賦值。這種方法跟以往我們習慣處理這類問題時大不相同，如果不用建構子的話我們通常要先宣告一個複數，再把值放進去，如程式碼 2.42。

```
Complex compAdd(const Complex &left, const Complex &right) {
  return Complex(left.real + right.real, left.imag + right.imag);
}

Complex result = compAdd(a, b);
```

程式碼 2.41: Complex 相加

```
Complex compAdd(const Complex &left, const Complex &right) {
  Complex c;
  c.real = left.real + right.real;
  c.imag = left.imag + right.imag;
  return c;
}
```

程式碼 2.42: Complex 相加

此外，原先的函式也可以放入 struct 中，成爲 struct 的**成員函式 (Member Function)**——也就是 Complex 的一個方法。成員函式和 real、imag 等成員資料 (Member Data) 的用法相同，使用「.」運算子來存取：

```
struct Complex {
  double real, imag;
  Complex() {
    real = imag = 0.0;
  }
  Complex(double r, double i) {
    real = r;
    imag = i;
  }
  Complex compAdd(Complex right) {
    return Complex(real + right.real, imag + right.imag);
  }
};

Complex res = a.compAdd(b);
```

程式碼 2.43: Complex 相加

程式碼 2.43 可發現複數 a 呼叫了 compAdd 函數，並傳入複數 b。在 compAdd 中，因爲此方法是 a 的成員，因此可以直接使用 a 的 real、imag。

若要寫得更自然，像是 int 做四則運算 a + b 的寫法時，我們就要用到**運算子多載 (Operator Overloading)**。在 C++ 中，運算子被當作成員函式一般，因此 a + b 可視爲

```
1 a.operator+(b);
```

程式碼 2.44: 運算子多載

程式碼 2.44 的 operator+ 是函式名稱，代表加法運算。既然「加法」可被當作一種成員函式，因此也可以對 Complex 類別多載「加法」，如程式碼 2.45。

```
1  struct Complex {
2    double real, imag;
3    Complex() {
4      real = imag = 0.0;
5    }
6    Complex(double r, double i) {
7      real = r;
8      imag = i;
9    }
10   Complex operator+ (Complex right) {
11     return Complex(real + right.real, imag + right.imag);
12   }
13 };
14
15 Complex result = a + b;
```

程式碼 2.45: 運算子多載

若要寫得更精準的話，我們知加法運算會把 a、b 兩者相加，但 a、b 本身值不變，這時就可以使用 const 修飾，如程式碼 2.46，其中後面的 const 代表呼叫 operator+ 本身物件不會被修改，回傳值亦爲一個新的 Complex 物件。

其他運算子大多可以多載，在此不贅述。

```
Complex operator+ (const Complex &right) const {
  return Complex(real + right.real, imag + right.imag);
}
```

程式碼 2.46: 運算子多載

(四) 類別與物件

本節一開始有提到物件和類別之間的關係，依據種類，我們可以把類別內記載的資料分爲以下四類：

- 實例屬性
- 實例方法
- 類別屬性
- 類別方法

實例屬性和方法即是物件所擁有的屬性和方法，例如 Complex 類別中，每個實例擁有不一樣的 real、imag 等。

```
struct Circle {
  double r;
  Circle() {}
  Circle(double c) { r = c; }
  static double PI;
};

double Circle::PI = 3.14;

int main() {
  cout << Circle::PI << endl;
}
```

程式碼 2.47: 類別屬性

類別屬性和方法可以看做是整個類別所**共有**的，也就是在同個類別下所有物件，看到的類別屬性值都會是相同的，以程式碼 2.47 爲例，我們要建立 Circle 類別，其中 Circle 擁有一個實例屬性 r，代表圓形的半徑。

在 Circle 類別中有一個類別屬性 PI，以一個浮點數代表圓周率，一般來說圓周率是一個通用值，因此可以設爲類別屬性，亦即往後創造的 Circle 物件，所見的圓周率皆是相同的。

設定一個屬性爲類別屬性是在類別中用 static 修飾，要**注意**的是，類別屬性在類別內只能宣告，賦值需要在類別外賦值，可以參考程式碼 2.47 第 8 行。若要使用類別屬性，需要使用範圍解析運算子「::」，使用方法如程式碼 2.47 第 11 行。

```
1  struct Circle {
2    double r;
3    Circle() {}
4    Circle(double c) { r = c; }
5    static double PI;
6    static double area(const Circle &c) {
7      return c.r * c.r * PI;
8    }
9  };
10
11 double Circle::PI = 3.14;
12
13 int main() {
14   Circle c(1);
15   cout << Circle::area(c) << endl;
16 }
```

程式碼 2.48: 類別方法

類別方法用法也和類別屬性類似，程式碼 2.48 展示類別方法 Circle::area。

(五) 命名空間

命名空間 (Namespace) 可以看做是管理程式碼的一種方法，在競賽之外往往會使用外部的標頭檔、第三方程式碼等，這些程式碼在變數、類別命名上會有同名的可能，爲了避免同名導致編譯錯誤，我們會將程式碼裝入一個命名空間內 (類似一個很大的類別)。

最直接的例子就是命名空間 std，在第一章我們提到程式的基本架構中有一行：

```cpp
using namespace std;
```

程式碼 2.49: 預設 std 命名空間

程式碼 2.49 代表我們在沒有指定使用何命名空間的東西時，預設就是使用 std 內的物件、函數等，如 std::cin、std::cout、std::sort 等等。

第三節　程式技巧

一、 函式化與結構化

當輸入、解題、輸出較爲複雜時，可以做以下的函式化：

```cpp
while ( input() ) {
  sol();
  output();
}
```

程式碼 2.50: 函式化程式

函式化雖然理論上會慢一些，但是可以換來以下好處，使得我們可以方便除錯：

- 易於閱讀。程式碼 2.50 可以清楚看出此程式區塊大致上在處理多筆輸入，針對每筆輸入解出對應解答並輸出。
- 易於撰寫。當你將解題所需大部分的工作都規劃成函式，那麼只要將每個函式個功能寫出來後，只要想法正確、沒有 bug，基本上都能一次 AC。

程式碼 2.51 是一個函式化的範例程式碼片段，除去 #define 等用法之外，雖然我們不知道題目是什麼，但可以清楚看出 input() 可以是每次讀完測資之後，讓 sol() 函數處理並回傳答案。

接著看 input() 部分，程式碼 2.52 可以看出是 0 尾版的輸入，只是 input() 是放在 while 的判斷條件內，因此無論是 0 尾、n 行、EOF 等，只要讀到一筆測資就回傳 true，反之則回傳 false。

最後，看看 sol()，可以一眼看出這個題目使用了求最大公因數 gcd (除非寫程式的人亂取名字)、sol 用到 gcd 求出答案。

```
#define MAX 1010
int n, a[MAX];

int main() {
  while (input()) printf("%d\n", sol());
}
```

程式碼 2.51: 函式化範例 (main 部分)

```
int input() {
  for (n = 0; scanf("%d", &a[n]), a[n]; n++)
    if (n) a[n - 1] -= a[n];
  return n;
}
```

程式碼 2.52: 函式化範例 (input 部分)

```
int gcd(int a, int b) {
  if (b) return gcd(b, a % b);
    return a;
}
int sol() {
  int i, g = abs(a[0]);
  for (i = 1; i < n - 1; i++)
    g = gcd(g, abs(a[i]));
  return g;
}
```

程式碼 2.53: 函式化範例 (sol 部分)

函式化容易辨識出哪個部分負責什麼樣的工作，因此就會比較好除錯。函式化的缺點就是程式碼容易變長、程式可能會較慢等等，但事實上寫程式的速度主要是思考、除錯的時間，除非在寫秒殺題，否則都是思考的時間居多，因此程式碼變長沒什麼關係，重點在於讓你除錯的時間變快，容易理解的程式碼也會加快除錯的速度。

二、 #define

三、 標準模板函式庫

(一) 模板

之前有學過函式多載，例如我們自己要寫一個取最小值的函數 myMin，我們可以自己寫一個判斷 int 最小值的函數：

```
1 int myMin(int x, int y) {
2   if (x < y)
3     return x;
4   return y;
5 }
```

程式碼 2.54: myMin 函數

但是可以取最小值的資料型態不只有 int，也可能是 double、long long、short、...等等，於是我們就需要寫出很多版本的最小值函數：

```
1 double myMin(double x, double y) {
2   if (x < y)
3     return x;
4   return y;
5 }
```

(a) double 版本

```
1 long long myMin(long long x, long long y) {
2   if (x < y)
3     return x;
4   return y;
5 }
```

(b) long long 版本

程式碼 2.55: 其他 myMin 函數

可以看出程式碼 2.55 中，除了宣告之外其餘部分與 2.54 完全相同，雖然我們可以利用函式多載完成所有最小值，但我們寫出了重複的程式，爲此，C++ 提供的模板功

能，模板相當於是一個「程式碼」的**範本**，當需要什麼類型的程式碼時，再自動「抄寫出」對應的一份程式碼出來。

```
1  template<class T>
2  T myMin(T x, T y) {
3    if (x < y)
4      return x;
5    return y;
6  }
7  int main() {
8    cout << myMin(1, 2) << endl;
9    cout << myMin(2.5, 3.5) << endl;
10 }
```

程式碼 2.56: 使用 template 的 myMin 函數

程式碼 2.56 使用了模板功能，會在函數前宣告 template<class T>，此時 T 可以看做是資料型態的變數，程式會在**編譯時期**檢查需要哪些種類的 myMin 函數，例如程式碼第 8 行，在編譯時期會產生函數 int myMin(int, int)，而在第 9 行，編譯器會知道需要 double 版本的 myMin，而產生 double myMin(double, double)。

此外，模板的參數也可以有很多個，程式碼 2.57 會用到兩種資料型態，一個命名爲 __Tx、另一命名爲 __Ty。

```
1  template<class __Tx, class __Ty>
```

程式碼 2.57: 使用兩種 class 的模板

模板也可套用在類別上，假設我們要儲存二維平面上的座標，類別 Point 可以寫作程式碼 2.58。

```
1  struct Point {
2    int x, y;
3  };
```

程式碼 2.58: Point 類別

若要儲存浮點數的二維座標的話，可以考慮利用 template，如程式碼 2.59 第 5、6 行，分別是儲存 int 和 double 座標的 Point 類別。C++ 的 STL 中很多是以 template 實作。

```
1 template<class T>
2 struct Point {
3   T x, y;
4 };
5 Point<int> p1;
6 Point<double> p2;
```

程式碼 2.59: Point 類別

競賽中，可以對模板進行遞迴，因爲模板會在編譯時期生成程式，而編譯時間不會計算在執行時間，因此有一些建表策略可以考慮在模板當中實行。這個方法被稱爲**模板後設編程 (Template metaprogramming)**。

程式碼 2.60 用費氏數列作爲例子，前四行定義了費式數列的遞迴式，。

最簡單使用模板的方法。

要注意的是，程式碼 2.61 fib<1> 中，靜態方法 init 會呼叫 fib<0>::init，若 fib<0> 在此之前未定義會出現編譯錯誤。

(二) 迭代器

(三) 計算頻率

四、 <algorithm> 函式庫

五、 其他注意事項

(一) 有關測試資料

範例測資無法代表一切 有些人會有疑問：「阿我範例測資過了，爲啥還是不會 AC？」所謂的範例測資，就是「**範例**」測資，只是提供幾個當作參考，可能故意漏掉一些重要的資訊讓你 WA 掉，或是誤導你的想法可是範例測資會讓你正確等等陰險手段。

```cpp
template<int n>
struct fib {
  static const int val = fib<n - 1>::val + fib<n - 2>::val;
};
template<> // fib[1] = 1;
struct fib<1> {
  static const int val = 1;
};
template<> // fib[0] = 0;
struct fib<0> {
  static const int val = 0;
};

int main() {
  cout << fib<10>::val << endl;
  int x = 10;
  cout << fib<x>::val << endl; // error
}
```

程式碼 2.60: 費氏數列遞迴

AC 未必是正解 有種東西叫做「假解」，就是存在非正解但卻可以 AC 的程式碼，例如說某個問題的正解要 DP，但有人爆搜卻過了；明明有個測資會讓你 WA 掉可是卻 AC 了 ... 以此類推。

有時是測資量不夠，沒有能夠讓假解不能過的特殊測資，或者是寫了正解，少部分的程式碼寫錯但卻 AC (這個最嚴重 ... 寫到類似題往往會認爲自己是正確的)，因此有時候 AC 了，只是僥倖，不能代表你程式碼完全正確。

邊界測資 很多人會忽略所謂「**邊界測資**」，就是題目輸入範圍內最極限的測資，忽略邊界測資往往會造成 WA 或 RE。

例如，計算費氏數列第 n 項 $(0 \le n \le 100)$，那麼第 0 項、第 100 項便是值得注意的範圍；問你 a^b 的值，那麼你要注意 0^b、a^0、0^0 等等這些異常莫名其妙的測資 (除非題目保證說沒有，不然都得考慮)。

小結 綜觀以上情形，平常最好是多思考更多測資去檢查自己的想法，很多時候不管

```cpp
template<int n>
struct fib {
  enum { val = fib<n - 1>::val + fib<n - 2>::val };
  static void init(vector<int> &v) {
    fib<n - 1>::init(v);
    v.push_back( val );
  }
};

template<>
struct fib<0> {
  enum { val = 0 };
  static void init(vector<int> &v) {
    v.push_back( val );
  }
};

template<>
struct fib<1> {
  enum { val = 1 };
  static void init(vector<int> &v) {
    fib<0>::init(v);
    v.push_back( val );
  }
};
```

程式碼 2.61: 把費氏數列裝進 vector 中

```cpp
vector<int> f;
fib<40>::init(f);
int x = 40;
for (int i = 0; i < x; i++)
  cout << f[i] << endl;
```

是 AC 還是 WA 都僅僅是參考。

想測資往往是一件滿困難的事，但多想可以加強思考力，除錯時最關鍵常常是邊界

測資。有時候邊界測資會是一個奇形怪狀的東西，只要符合題目敘述，就有可能是讀者思考中遺漏掉的邊界測資。

(二) 記憶體與程式風格

使用記憶體不見得要精準 在解題比賽時，記憶體是不是用得恰當，只要題目能 AC、寫起來寫得順就行！實作上只要開「**夠大**」的陣列就好了，例如：需要長度爲 100 的陣列，可以開長度爲 110 的陣列，原因是我們不必花太多心思去決定邊界到底是 100 還是 101 這回事上 (別忘了在比賽，有些細節可以不必太要求)。

有些題目設計上會**要求**記憶體的使用，這類題目主要是在於使用不同的資料結構而有差異，通常 110 和 100 的陣列這點浪費記憶體不足以影響結果 (機率低)。例如，某個資料結構需要 100010×100010，在大多數的機器上是無法宣告這麼大的二微陣列，但替換成另一個資料結構與演算法，可能讓使用的記憶體減少至 3×100010。

小心遞迴 遞迴時會使用到系統堆疊，這時就要注意演算法是會因爲遞迴過深造成 RE。

(三) 除錯相關

常常上傳後，得到的結果是 WA，此時最著名的第一個反應是：「不可能會寫錯呀？一定是 judge 有問題 ... bla bla。」等到抓到錯誤才恍然大悟：「哦～原來只是這樣 ... bla bla。」

除錯 (debug) 也是寫程式重要的一個環節，一個程式如果不想除錯，就不要寫出有 bug 的程式，但常常事與願違。由於程式是自己寫出來的，面對自己的程式碼，最大的困難點在於會認爲「自己沒有寫錯」。最好的解法就是要記得每次找到錯誤時，要稍微記得自己在哪裡出錯，或是要注意大家常常出錯的地方。

例如在學 C++ 初期，常常會把邏輯運算式的「相等」少寫一個 =，如程式碼 2.62。

```
1 if (a = 5) ...
```

程式碼 2.62: 比較運算子誤寫

或是陣列的使用上超出範圍 (程式碼 2.63)。

```
1 int a[100];
2 for (i = 1; i <= 100; i++)
3 cin >> a[i];
```

程式碼 2.63: 比較運算子誤寫

其他常見的 bug 如下：

- 打**錯字**，包括 == 打成 =，打錯數字、符號等等
- 應注意的事項**未注意**，例如：輸出格式不對，陣列大小不對，忘記初始化，忘記 return 等等
- 操作上的錯誤，例如：傳錯程式碼，傳錯題目，**忘記存檔**，**除錯訊息忘記拿掉**等

(四) 喇賽

當你沒辦法想出正解時，**喇賽**是必要的途徑。不要以爲隨便亂寫不重要，其實到最後不管是初學者還是高手，總會遇到不知道怎麼解的題目 (或是刻意出正解很難、但是喇賽就會過的題目)，這時候就是比較誰喇賽功力的強弱了！

通常喇賽可以是很**多個演算法**混在一起；**爆搜**、**剪枝**再加一些東西；或者根本**亂寫**。很多技巧都可以應用在喇賽領域，有些人光靠喇賽 AC 但別人寫正解卻會 TLE (這有發生過)，因此喇賽是比賽的「終極」——很多時候在分測資點的比賽中也還是會撈到一些分數，但只要我們能夠將喇賽的技巧練得純熟，或許可能很快就不知道 WA 爲何物了。

快速上手喇賽技巧，就是遇到每一題都可以嘗試亂寫看看，那麼當你每一題都 AC 的時候就是你出師的時候了。

第四節　程式執行

一、 執行時期配置

前面提到程式需要放到記憶體中執行，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

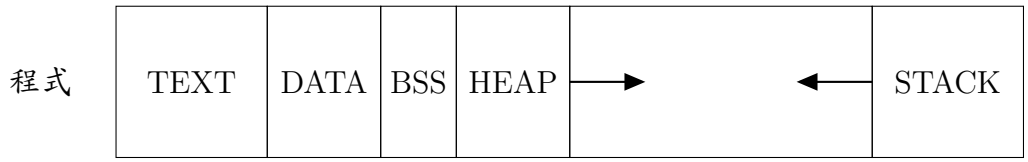


圖 2.11: 程式在記憶體執行的架構

- DATA 區塊：有初始化的全域變數、靜態變數等
- 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，使用一種稱爲**堆積 (Heap)** 的資料結構
- STACK 區塊：函數呼叫、區域變數等，使用稱爲**堆疊 (Stack)** 的資料結構

這些區塊的概念清楚有助於除錯，有興趣的讀者可以自行 google。

(一) 動態配置記憶體

```
1 int *a = new int[10];
2 delete a;
```

(二) 變數可視範圍

二、 程式語言

(一) 直敘式語言

(二) 物件導向語言

(三) 機器語言與組合語言

三、 程式編譯

第 三 章

字串處理

第一節　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，這個表格我們稱爲 **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 3.1。

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

表 3.1: 常用字元編號

(一) 特殊字元

有些字元因爲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 3.2 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 3.2: 常用特殊字元

(二) 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'

因爲 char 實際上是存整數，所以可以用大於小於來判斷，程式碼 3.1 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 3.1: 判斷數字字元

在 <cctype> 中有一些可以判斷字元類型的函式，常用的如表 3.3，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。

(三) 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' 的編號爲 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，因爲我們

函式	範圍	意義
int isalnum(int)	A 到 Z、a 到 z、0 到 9	判斷是否爲英文字母或數字字元
int isalpha(int)	A 到 Z、a 到 z	判斷是否爲英文字母
int isdigit(int)	0 到 9	判斷是否爲數字字元
int islower(int)	a 到 z	判斷是否爲小寫字母
int isupper(int)	A 到 Z	判斷是否爲大寫字母

表 3.3: <cctype> 常用函式

必須記得 '0' 的編號，在 C++ 中有提供字元相減的方法，於是我們可以寫爲程式碼 3.2，將一個數字字元減去 '0'。

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

程式碼 3.2: 數字字元轉換爲 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++ 的字串爲 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 3.3。

```
1 char str[110];
```

程式碼 3.3: 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 碼爲 0 的 '\0' (表 3.2)，代表「字串的結尾」。

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

程式碼 3.4: C 字串

例如程式碼 3.4 的字串爲 "bird"，事實上儲存情形如圖 3.1。

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

圖 3.1: 字串就是字元陣列

因爲 '\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 3.5。

```
size_t strlen(const char *str);
int    strcmp(const char *str1, const char *str2);
char*  strcpy(char* dest, const char* src);
char* strncpy(char* dest, const char* src, size_t num);
char*  strcat(char* dest, const char* src);
char* strncat(char* dest, const char* src, size_t num);
```

程式碼 3.5: 常用的字串函數

(一) 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' 爲止，並回傳長度值。

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

程式碼 3.6: strlen 範例

程式碼 3.6 回傳的長度值爲 4。當然我們也可以對字串變數做 strlen，程式碼 3.7 中 num 字串的長度爲 10。

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

程式碼 3.7: 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 3.8a 的寫法非常浪費時間，每次迴圈都會重新呼叫一次 strlen。

```cpp
for (i = 0; i < strlen(str); i++)
{
  // do something
}
```

(a) 直觀寫法

```cpp
int len = strlen(str);
for (i = 0; i < len; i++)
{
  // do something
}
```

(b) 較好的寫法

程式碼 3.8: 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 3.8b。

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。

(二) strcmp 函數

strcmp 函數用來比較兩個 C 字串，參數爲兩個字串指標。它會從兩個字串一開始的字元逐個比較，比較到**其中一個**爲 '\0' 爲止，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。

(三) 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' 爲止。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爲止，這對程式而言是非常危險的，因爲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**。

其他較爲安全的替代方案，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'。

(四) 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** (有運算子多載)，但因爲 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。

(一) 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 3.9: C++ 字串轉 C 字串

要注意的是，c_str() 產生一個**唯讀**的 C 字串。

(二) 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 3.10。

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

程式碼 3.10: 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) 它的長度爲何？

(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

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

(a) 這個字串的長度爲何？

(b) strlen(str4 + 4) 的回傳值爲何？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？

```
cout << str4 << "␣" << str4 + 4 << endl;
```

(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

```
string str5 = "bird";
char str6[110];
```

要怎樣將 str5 複製給 str6 呢？

5. 現有兩字串

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

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

第二節　輸入與輸出

一、 格式字串

(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，輸入的函數爲 scanf，而輸出的函數爲 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++ 的相比較爲繁瑣，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 3.11: scanf 和 printf

scanf 和 printf 的參數中，後面 ... 稱爲**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，這個字串我們稱爲**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 3.12。

```
1   printf("Hello␣world!\n");
```

程式碼 3.12: 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 3.13。

程式碼 3.13a 直接印出數字，也可以寫爲 printf("3\n");，程式碼 3.13b 會印出變數 x 的值，印出的位置在 "%d" 處。

輸入的話，因爲我們是呼叫函數，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爲程式碼 3.14。

```c
printf("%d\n", 3);
```

(a) 印出常數

```c
int x = 3;
printf("%d\n", x);
```

(b) 印出 int 變數

程式碼 3.13: 印出整數

```c
int x;
scanf("%d", &x);
```

程式碼 3.14: 輸入一個整數

同樣地，unsigned int、long long、unsigned long long、float、double 都有對應的的格式，如表 3.4。

型態	對應格式	備註
unsigned int	"%u"	
long long	"%lld"	windows 環境下可能會用 "%I64d"
unsigned long long	"%llu"	windows 環境下可能會用 "%I64u"
float	"%f"	
double	"%lf"	printf 時須用 "%f"
char	"%c"	
char[]	"%s"	C++ 的字串 string 沒辦法直接使用 scanf、printf

表 3.4: scanf 和 printf 格式表

因爲 % 在 scanf 和 printf 當中作爲跳脫字元，印出 % 要使用 "%%"。

和 cin 類似，scanf 中也不會用空白和換行，如程式碼 3.15。

```c
scanf("%d%d%d", &a, &b, &c);
```

程式碼 3.15: scanf 範例

(二) 字元讀取問題

讀取字元時，cin 和 scanf 的行爲會不一致。

```
1 char a, b;
2 cin >> a >> b;
3 scanf("%c%c", &a, &b);
4 printf("\'%c\' \'%c\'\n", a, b);
```

程式碼 3.16: cin 和 scanf 行爲不一致

程式碼 3.16 第 2 行和第 3 行中可以嘗試輸入「1 2」，可以發現 cin 會跳過空白，可是 scanf 並不會。

(三) scanf_s 函數

VC++ 在讀取字串時很多時候會被擋下，因爲 scanf 讀取字串不會檢查長度，會有安全問題。scanf_s 函式在讀取字串時要多傳一個參數，作爲最多讀取的長度，程式碼 3.17 爲 scanf_s 的一個範例。

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

程式碼 3.17: scanf_s 的範例

程式碼 3.17 中，因爲 str 字串最後要有 '\0' 字元，因此最多只能讀 4 個字元。

到目前爲止，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？

(四) 進位輸出

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 3.18a。

程式碼 3.18b 是對應的版本，分別使用 "%o"、"%d"、"%x"。如要將十六進位印爲大寫，cout 須加上 std::uppercase，printf 則使用 "%X"。

(五) iomanip 格式

在 C++ 中，標頭檔 <iomanip>，專門做輸入輸出的處理。表 3.5 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。

```cpp
int a = 11;
cout << oct << a << endl;
cout << dec << a << endl;
cout << hex << a << endl;
```

(a) cout 版本

```cpp
int a = 11;
printf("%o\n", a); // 13
printf("%d\n", a); // 11
printf("%x\n", a); // b
```

(b) printf 版本

程式碼 3.18: 輸出進制比較

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 3.5: <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 3.19，setprecision 的參數爲 5，因此會保留 5 位有效位數。

```cpp
double f = 3.14159;
cout << setprecision(5) << f << endl; // 3.1415
cout << fixed << setprecision(5) << f << endl; // 3.14159
```

程式碼 3.19: 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 3.19 第 3 行。

std::setw 代表輸出的寬度，例如程式碼 3.20，輸出的結果爲 "␣␣␣16"。

```cpp
int a = 16;
cout << setw(5) << a << endl;
```

程式碼 3.20: 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 3.21。

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

程式碼 3.21: 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 3.22。

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

程式碼 3.22: printf 格式範例

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 3.23a。

```
int hh, mm;
char ch;
cin >> hh >> ch >> mm;
```

(a) 用 cin 輸入

```
int hh, mm;
scanf("%d:%d", &hh, &mm);
```

(b) 用 scanf 輸入

程式碼 3.23: 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 3.23b。

(六) 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 版就會寫爲程式碼 3.24。

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

程式碼 3.24: EOF 版

二、 標準 I/O

(一) 簡介

(二) 行讀取

(三) 字元讀取

(四) 輸入輸出優化

三、 檔案 I/O

(一) 開檔讀檔

(二) fscanf 和 fprintf

(三) freopen

四、 字串 I/O

(一) sscanf 和 sprintf

(二) stringstream

C++ 中也有提供字串 I/O，稱爲 stringstream 類別，在 <sstream> 標頭檔裡面，用法與 cin、cout 差不多，如程式碼 3.25。

程式碼 3.25 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 3.26。

```cpp
stringstream ss;
string str;
ss << "Hello world!"; // 將東西塞進 stringstream
ss >> str; // 丟到字串
cout << str << endl; // Hello world!
```

程式碼 3.25: stringstream 基本用法

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

程式碼 3.26: 可以用迴圈來讀取

```cpp
while (ss >> a) {
  // do something
}
```

程式碼 3.27: stringstream 碰到 EOF

第三節　字串技巧

一、 善用 index

二、 回文

三、 二維問題

四、 子字串

五、 其他

第四節　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 3.6。

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

表 3.6: 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 寫爲「I」、數字 2 寫爲「II」、3 表示爲「III」，以此類推。
- **減法原則**：爲簡化書寫，4 不寫作「IIII」，而是用「$5-1$」寫爲「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，則爲加法；反之則爲減法。例如：11 爲 $10+1$，表示爲 XI；9 被當作 $10-1$，表示爲 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 雖然可以視爲 $100-1$，但羅馬數字會統一看做 $90+9$，也就是 XC (90) 和 IX (9)，寫爲 XCIX。
- 符號不超過**三個**：4 會寫爲 IV 而非 IIII，9 被寫爲 IX 而非 XIIII。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)。

(一) 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 3.7 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 3.7: 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 3.28 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 3.28: 儲存符號、餘數

程式碼 3.28 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。

程式碼 3.29 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爲什麼不能反過來)。

(二) 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。

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

程式碼 3.29: 阿拉伯數字轉羅馬數字

二、 字串和數字轉換

(一) 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 3.2 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。

$$\begin{aligned} 123 &= 1 \times 10^2 + 2 \times 10^1 + 3 \times 10^0 \\ &= (\text{'1'} - \text{'0'}) \times 10^2 + (\text{'2'} - \text{'0'}) \times 10^1 + (\text{'3'} - \text{'0'}) \times 10^0 \end{aligned}$$

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。

如程式碼 3.30，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。

```
int MyAtoI(string str)
{
  int res = 0, i;
  for (i = 0; i < str.size(); i++)
  {
    res *= 10;
    res += str[i] - '0';
  }
  return res;
}
```

程式碼 3.30: 從高位數開始做

(二) 數字轉字串

(三) 進位變換

三、 習題

第 四 章

計算思維與設計技巧

第一節　遞推思維

一、 排列組合

二、 常見數列

三、 可樂問題

四、 平面切割

第二節　質數

一、 質數判斷

二、 質數篩法

三、 質因數分解

第三節　模擬

一、 撲克牌

二、 西洋棋

索 引