

## 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` 的定義為「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 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  $-2^{32}$ ！為什麼會這樣子呢？

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 為標準，有 8 個位元組，最多只能表示  $2^{64}$  種小數。因為在標準中，以 53 個位元儲存尾數，故有 52 位有效數字，精確度為  $\log 2^{52} \approx 15.95$  位十進位有效數字 (`float` 則為 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，用十六進位表示為 0x7FFFFFFF
- `long long` 範圍大概是  $-9 \times 10^{18}$  到  $9 \times 10^{18}$  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	$-3.40282 \times 10^{38}$	$3.40282 \times 10^{38}$
double	8	$-1.79769 \times 10^{308}$	$1.79769 \times 10^{308}$
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，又稱為記憶體階層 (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)。為了方便管理記憶體，計算機會幫每個位元組標記「地址」，在此我們就稱為「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

筆者的執行結果為：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」，因為 `int` 通常為 4 個位元組，因此會佔據 0x003FF07C、0x003FF07D、0x003FF07E、0x003FF07F 這四個位元組。如圖 2.2：

這個結果會因為不同機器、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 ㄣ ㄣ)，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？很多人會以為像是圖 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。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	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 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。

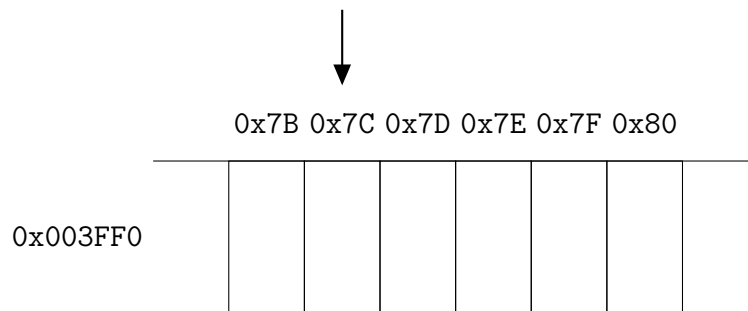


圖 2.5: 指標概念

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 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。

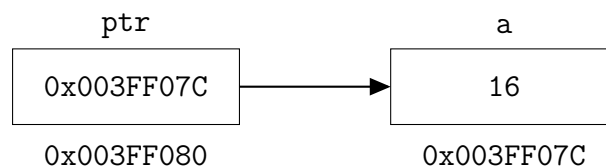


圖 2.6: 程式碼 2.6 的狀況

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 為例。

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，因為 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 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 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++`、`*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` 的值為何？
- `*&ptr` 和 `&*ptr` 有什麼不同？

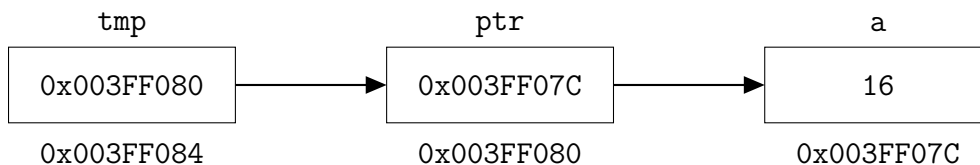


圖 2.7: 程式碼 2.10 的狀況

- `&&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 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 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。

為了簡化描述，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。

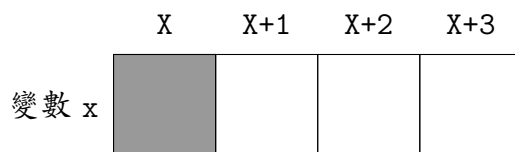


圖 2.8: 實際上 ptr 的有效範圍

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) 空指標

在 C++ 中，為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，而要經過型別轉換為指標型態後，也就是 (int\*)NULL 之類的操作，才會變為空指標。

```

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：

- 當回傳型態為 `void` 時，代表這個函式「**沒有回傳值**」；
- 當回傳型態為一個 `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 2.14 的結果會像圖 2.9。

00000000 <b>1</b>	00000000 <b>1</b>	00000000 <b>1</b>	00000000 <b>1</b>
-------------------	-------------------	-------------------	-------------------

圖 2.9: 程式碼 2.14 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。常用的值有兩個：0 和 -1 (255)，因為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為 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++ 中大括號被稱為程式區塊，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、大括號做為區隔。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。

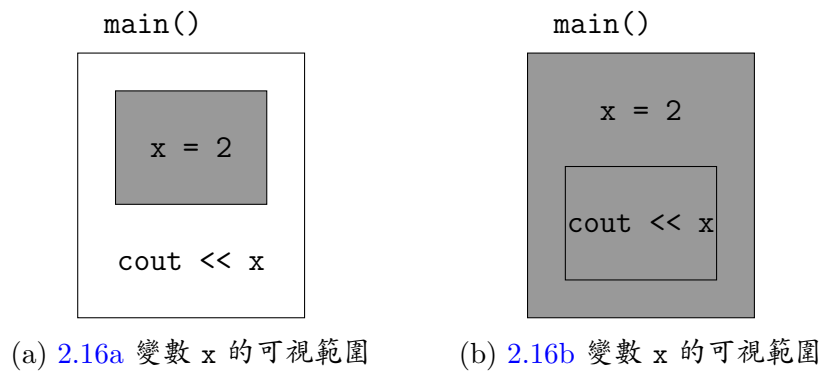


圖 2.10: 程式碼 2.16 的狀況

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 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

條件判斷更可進階為 `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為 `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 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？

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

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，如果結果為 `true` 則執行對應程式區塊；直到判斷式為 `false`，則跳出迴圈結構。

新手常見的錯誤是無法使判斷式變為 `false` 導致無法跳出迴圈，或是寫出一個判斷式為 `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。

<pre> 1  for (int i = 0; i &lt; 10; i++) { 2      cout &lt;&lt; i &lt;&lt; endl; 3  } </pre>	<pre> 1  { 2      int i = 0; 3      while (i &lt; 10) { 4          cout &lt;&lt; i &lt;&lt; endl; 5          i++; 6      } 7  } </pre>
(a) <code>for</code> 語法	(b) 對應的 <code>while</code> 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` 為 `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，因為在每次執行完迴圈時，`i` 值會遞增，直到不小於 10，判斷式變為 `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，因為 `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。

```
1  if (a < b < c)
2      cout << c << endl;
3  else if (a < c < b)
4      cout << b << endl;
5  else if (b < a < c)
6      cout << c << endl;
7  else if (b < c < a)
8      cout << a << endl;
9  else if (c < a < b)
10     cout << b << endl;
11 else
12     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` 作為判斷，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。

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

程式碼 2.26: 求  $n$  個數字最大值

如果沒辦法初始化為第一個數字時，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，則令最大值為隨便一個負數即可 `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：

```
1 while (cin >> n, n) {  
2     cout << n << endl;  
3 }
```

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，當輸入為零時，逗號右側的 `n` 會判定為 `false` 而跳出迴圈。

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

**n 行版** 典型 `n` 行版的輸入為：「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：

```
1 for (cin >> n; n; n--) {  
2     cout << n << endl;  
3 }
```

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` 值為零而退出迴圈。

```

1  for (cin >> n, cnt = 1; cnt <= n; cnt++) {
2      cout << "Case_" << cnt << ":" << endl;
3  }

```

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，因此設計上可寫為程式碼 2.30。

```

1  while (cin >> n) {
2      cout << n << endl;
3  }

```

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]`

```

1  int a[5] = { 1, 4, 9, 16, 25 };
2  int *ptr = &a[1];
3  cout << ptr << endl;
4  cout << *ptr << endl;
5  cout << ptr + 1 << endl;
6  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，此種指標型態為 `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。

```

1  int a[5] = { 1, 4, 9, 16, 25 };
2  int *p = a; // int* 指向陣列 a
3  cout << a << endl;
4  cout << a + 1 << endl; // a[1] 的位址
5  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。

```

1 void c8763(int x) {
2     x++;
3     return;
4 }
5 int main() {
6     int a = 0;
7     c8763(a);
8     cout << a << endl; // 0
9 }

```

程式碼 2.33: 傳值呼叫

```

1 void c8763(int *x) {
2     (*x)++;
3     return;
4 }
5 int main() {
6     int a = 0;
7     c8763(&a);
8     cout << a << endl; // 1
9 }

```

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、回傳值為 `int` 的函數而言，其指標型態為 `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。

```

1 void c8763(int &x) { // 參考
2     x++;
3     return;
4 }
5 int main() {
6     int a = 0;
7     c8763(a);
8     cout << a << endl; // 1
9 }

```

程式碼 2.35: 傳參考呼叫

```

1 int f(int a, int b) { return a + b; }
2 int g(int a, int b) { return a - b; }
3 int h(int (*func)(int, int), int a, int b) {
4     return func(a, b);
5 }
6 int main() {
7     cout << h(f, 1, 2) << endl; // 3
8     cout << h(g, 2, 1) << endl; // 1
9 }

```

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為一個又一個的物件 (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為模式，以及如何影響其他物件，此時這些函式稱為方法 (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 的一個實體 (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，依據種類，我們可以把類別內的

東西分爲以下四類：

<pre>1 struct Complex { 2     double real, imag; 3 };</pre>	<pre>1 class Complex { 2     public: 3     double real, imag; 4 };</pre>
(a) Complex 結構	(b) Complex 類別

程式碼 2.37: 結構和類別的對應

## (二) 建構子與解構子

我們上一節提到過，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，因此我們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，我們希望每個複數一開始都能被歸零，在 C 語言的寫法中，我們變數宣告完就：

## (三) 運算子多載

## (四) 命名空間

# 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，筆者傾向於做以下的函式化：

```
1 while ( input() ) {
2     sol();
3     output();
4 }
```

程式碼 2.38: 函式化程式



程式碼 2.38 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。

以下是一個函式化的範例程式碼：

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <iostream>
4  using namespace std;
5
6  #define MAX 1010
7  int n, a[MAX];
8
9  inline int input() {
10     for (n = 0; scanf("%d", &a[n]), a[n]; n++)
11         if (n) a[n - 1] -= a[n];
12     return n;
13 }
14
15 int gcd(int a, int b) {
16     if (b) return gcd(b, a % b);
17     return a;
18 }
19
20 int sol() {
21     int i, g = abs(a[0]);
22     for (i = 1; i < n - 1; i++)
23         g = gcd(g, abs(a[i]));
24     return g;
25 }
26
27 int main() {
28     while (input()) printf("%d\n", sol());
29 }
```

程式碼 2.39: 函式化範例

上述範例，除去 inline、define 等等看不懂的法之外，雖然我們不知道題目是什麼，但是我們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)、input 就是輸入測資然後做一些處理、sol 就是求出答案，而 main 裡面只要循環輸入和輸出答案就好了，這種寫法很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。

寫成函式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 (有時候除錯會遇到不知道怎麼錯的情況 ... 相信大家都有經驗)，容易理解的程式碼除錯也會加快一定的速度。

## 二、 #define 與 inline

## 三、 標準模板函式庫

### (一) 模板

### (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

### (二) 記憶體與程式風格

### (三) 除錯相關

### (四) 喇賽

## 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
- DATA 區塊：有初始化的全域變數、靜態變數等

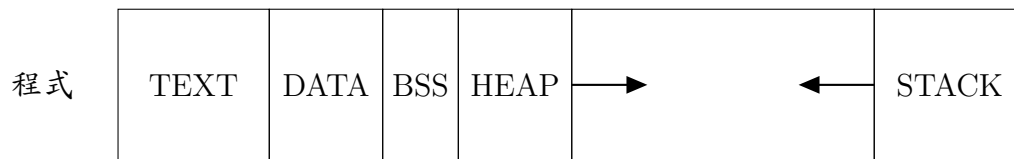


圖 2.11: 程式在記憶體執行的架構

- 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

這些區塊的概念清楚有助於除錯，有興趣的讀者可以自行 google。

## 二、 動態配置記憶體

## 三、 變數可視範圍

## 四、 程式語言

機器語言與組合語言

高階語言

## 五、 程式編譯

# 索引

void, 12

位元, 3

位元組, 3

位址, 4

取值運算子, 7

取址運算子, 4

可視範圍, 14

堆疊, 27

堆積, 27

大字節序, 5

宣告, 10

小字節序, 5

指標, 5

函數指標, 22

標頭檔

cfloat, 2

climits, 2

cstddef, 11

cstring, 11

iostream, 17

溢位, 1

程式區塊, 13

空指標, 11

記憶體, 3

運算子

下標運算子, 21