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？

```
int x = 2147483647;
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)。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！(╯°□°）╯︵ ┻━┻)，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：

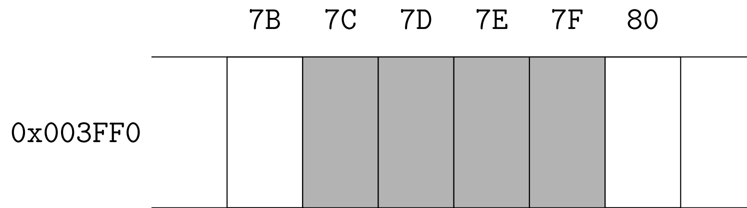


圖 2.2: 變數 a 實際的記憶體位置

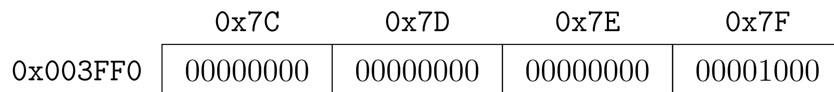
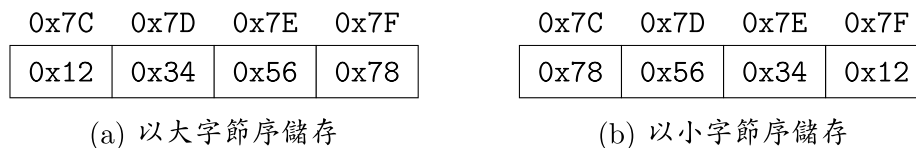


圖 2.3: 大字節序儲存方法

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：



(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。

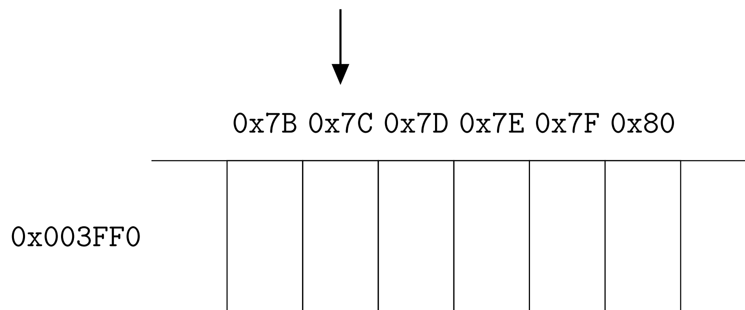


圖 2.5: 指標概念

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。

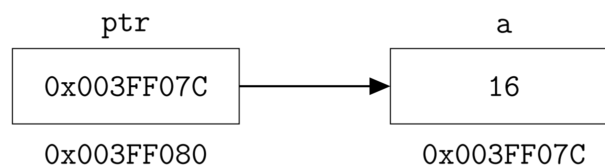


圖 2.6: 程式碼 2.6 的狀況

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」。

```cpp
int a = 16;
int b = 4;
int *ptr = &a;
cout << *ptr << endl;
ptr = &b;
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：

```cpp
int a = 16;
int *ptr = &a;
cout << "Value of a = "    << a << endl;
cout << "Address of a = " << &a << endl;
cout << "Value of *ptr = "  << *ptr << endl;
cout << "Value of ptr = "   << ptr << endl;
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。

```
int a = 16;
int *ptr = &a;
(*ptr)++;
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。

```
int a = 16;
int *ptr, **tmp;
ptr = &a;
tmp = &ptr;
cout << "a:" << endl;
cout << "Value of a = "     << a << endl;
cout << "Address of &a = " << &a << endl;
cout << "ptr:" << endl;
cout << "Value of ptr = "     << ptr << endl;
cout << "Value of *ptr = "    << *ptr << endl;
cout << "Address of &ptr = " << &ptr << endl;
cout << "tmp:" << endl;
cout << "Value of tmp = "     << tmp << endl;
cout << "Value of *tmp = "    << *tmp << endl;
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？

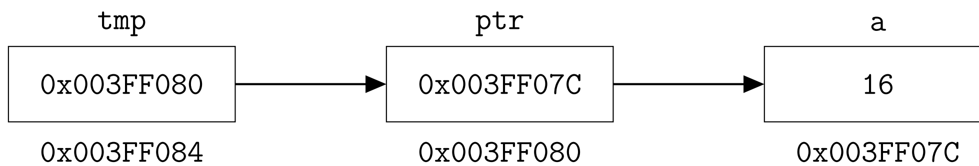


圖 2.7: 程式碼 2.10 的狀況

- &&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。

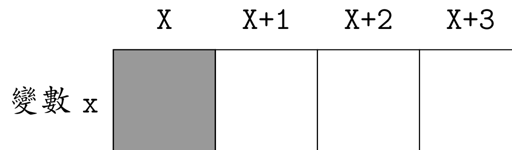


圖 2.8: 實際上 ptr 的有效範圍

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。

```cpp
#include <cstddef>

int* x = NULL;
int* y = (int*)NULL;
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：

```cpp
void* memset(void* ptr, int value, size_t num);
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

舉例來説，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。

0000000**1**	0000000**1**	0000000**1**	0000000**1**

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 }
```

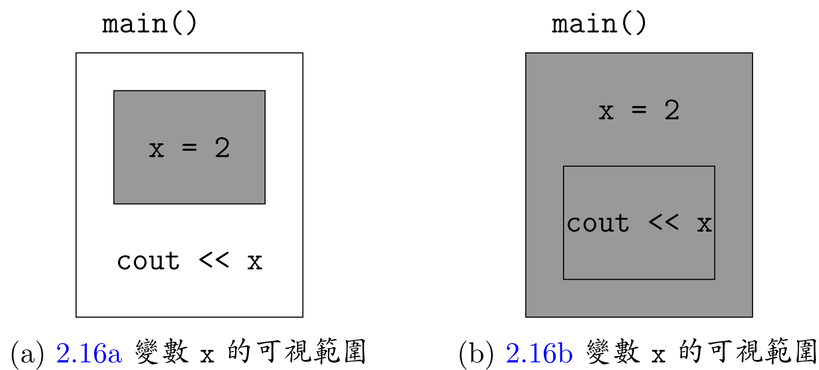
(a) 被大括號包住的 x

```
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。

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

(a) 單行指令

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

(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？

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

(a) 危險的 else

```
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。

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

(a) for 語法

```
{
  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。

```
int i = 0;
do {
  cout << i << endl;
} 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。

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

```
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：

```
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：

```
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 值爲零而退出迴圈。

```
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。

```
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]`

```cpp
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。

```cpp
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。

```
1 struct Complex {
2   double real, imag;
3 };
```

(a) Complex 結構

```
1 class Complex {
2 public:
3   double real, imag;
4 };
```

(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。

```
1 Complex c;
2 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
- 建構子名稱必為類別名稱，因此 Complex() 即為 Complex 類別的建構子

```
struct Complex {
  double real, imag;
  Complex() {
    real = imag = 0.0;
  }
};
```

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。

```
struct Complex {
  double real, imag;
  Complex() {
    real = imag = 0.0;
  }
  Complex(double r, double i) {
    real = r;
    imag = i;
  }
};

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 的實部虛部初始化為 0，而 c2 會呼叫另外一個建構子，將實部初始化為 4.0，而虛部為 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

```
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。

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

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。

```cpp
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。

```cpp
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。

```
struct Circle {
  double r;
  Circle() {}
  Circle(double c) { r = c; }
  static double PI;
  static double area(const Circle &c) {
    return c.r * c.r * PI;
  }
};

double Circle::PI = 3.14;

int main() {
  Circle c(1);
  cout << Circle::area(c) << endl;
}
```

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。

```c
#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)

```c
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)

```c
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 與 inline

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。

```
template<class T>
T myMin(T x, T y) {
  if (x < y)
    return x;
  return y;
}
int main() {
  cout << myMin(1, 2) << endl;
  cout << myMin(2.5, 3.5) << endl;
}
```

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。

```
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。

```
struct Point {
  int x, y;
};
```

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 $(0 \leq n \leq 100)$，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

```
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

```
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)。

```
int a[100];
for (i = 1; i <= 100; i++)
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) 唎賽

當你沒辦法想出正解時，**唎賽**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、但是唎賽就會過的題目)，這時候就是比較誰唎賽功力的強弱了！

通常唎賽可以是很**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唎賽領域，有些人光靠唎賽 AC 但別人寫正解卻會 TLE (這有發生過)，因此唎賽是比賽的「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，但只要我們能夠將唎賽的技巧練得純熟，或許可能很快就不知道 WA 爲何物了。

快速上手唎賽技巧，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

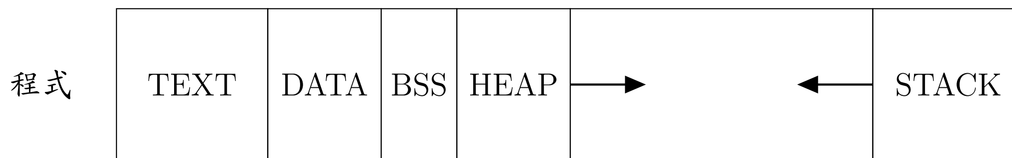


圖 2.11: 程式在記憶體執行的架構

- 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

```
int *a = new int[10];
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

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, 39
堆積, 39
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

物件導向
    命名空間, 29
    實例, 23
    建構子, 23, 24
    成員函式, 26
    成員資料, 26
    方法, 23
    物件, 23
    解構子, 23
    類別, 23
程式區塊, 13
空指標, 11

記憶體, 3
費氏數列, 34

運算子
    下標運算子, 21
    成員運算子, 26
    範圍解析運算子, 29
運算子多載, 27