第 7 章
陣列與指標

本章提要

- 7-1 甚麼是陣列？
- 7-2 字元陣列
- 7-3 多維陣列
- 7-4 指標與參照
- 7-5 指標與參考在函式上的應用
- 7-6 綜合演練

7-1 甚麼是陣列？

- 由於一個變數只能存放一個數值，而針對有些用來處理大批資料 (如學生、員工資料等) 的程式，就必須宣告許多的變數來存放這些資料。此時為了方便，我們便可利用陣列來代替多個變數。
- 使用陣列來代替變數的好處是可以讓程式碼容易撰寫。因為一個陣列可取代多個變數。比如說，我們要用 10 個變數來儲存 10 個學號，以之前所學的方法可能要寫成：

甚麼是陣列？

```
int  student1_ID=1001,  student2_ID=1002,
     student3_ID=1003,  student4_ID=1004,
     student5_ID=1005,  student6_ID=1006,
     student7_ID=1007,  student8_ID=1008,
     student9_ID=1009,  student10_ID=1010;
```

甚麼是陣列？

- 如果改用陣列來儲存 10 個學號, 會方便許多, 如下：

```
int  student_ID[10]={1001,1002,1003,1004,1005,
                 ↑      1006,1007,1008,1009,1010};
   //  代表可存放十個學號的陣列
```

- 陣列可說是一群同型別與同性質變數的集合, 就等於一次宣告了多個變數的儲存空間。

宣告與定義陣列

- 陣列的宣告方式和宣告一般變數一樣，都要指明資料型別和變數名稱，此外需在變數名稱後加上一對中括號 [ ] (稱為足標標算子, subscript operator)，表示這是個**陣列**變數。並需在中括號內填入陣列的容量，表示可以容納多少個同型別的變數：

```
資料型別  陣列名稱  [陣列容量];
```

宣告與定義陣列

- 例如以下就是陣列宣告的例子：

```
int     car[10];          // 可容納 10 個整數的 car 陣列

float  score[20];         // 可容納 20 個浮點數的 score 陣列
```

- 請注意，宣告陣列時所指定的陣列大小，必須是常數 (含 const 變數) 或由常數組成的運算式，不可以是在編譯時期不能確定其值的變數或運算式。換言之，陣列大小是在宣告或定義時就固定的，之後即無法改變其大小。

宣告與定義陣列

- 陣列中每個可用來存放資料的空間稱為**元素**, 例如上例中的 car[], 我們就可稱它為是十個元素的整數陣列, 每個元素就像一個整數變數一樣, 可存放一筆整數資料。要使用元素時, 需以陣列名稱及足標運算子, 並在足標運算子中指定元素的編號 (或稱為元素的**索引**) 請注意, 元素的編號是從 0 開始, 所以上列中的 car 陣列, 可使用的索引值是 0~9, 例如：

宣告與定義陣列

```
car[0] = 10;        // 將第 1 個元素的值設為 10
car[3] = 15;        // 將第 4 個元素的值設為 15
car[9] = car[3];// 將第 10 個元素的值設為與第 4 個元素相同
car[10] = 1000; // 錯誤, 索引超出範圍
```

- 以下程式簡單示範陣列宣告及存取陣列元素的語法：

宣告與定義陣列

程式 Ch07-01.cpp 宣告陣列變數及存取陣列元素

```
01 #include <iostream>
02 using namespace std;
03
04 int main()
05 {
06   int iArray[5];                // 宣告陣列
07
08   for(int i=0;i<5;i++)          // 用迴圈設定各元素值
09     iArray[i] = i * 5;
10
11   cout << "iArray 陣列的大小爲 " << sizeof(iArray)
12        << " 個位元組" << endl;
13   for(int i=0;i<5;i++)          // 用迴圈輸出各元素值
14     cout << "iArray[" << i << "] =" << iArray[i] << endl;
15 }
```

宣告與定義陣列

執行結果

```
iArray 陣列的大小爲 20 個位元組
iArray[0] =0
iArray[1] =5
iArray[2] =10
iArray[3] =15
iArray[4] =20
```

1. 第 6 行宣告可存放 5 個元素的整數陣列 iArray。

宣告與定義陣列

2. 第 8~9 行利用 for 迴圈依序設定 iArray 各元素的值。請注意用來當元素索引值的變數 i 是從 0 開始, 而不是從 1 開始。
3. 第 11~12 行輸出以 sizeof() 運算子取得的 iArray 陣列大小。由於整數資料型別佔用 4 個位元組, 因此 5 個整數陣列元素即佔用 5x4=20 個位元組的記憶體空間。
4. 第 13~14 行再次以 for 迴圈依序輸出 iArray 各元素的值。

宣告與定義陣列

- 使用迴圈來操作陣列是很常見的處理方法，而為了避免在操作時不小心超出陣列元素的索引範圍，通常會先在程式開頭定義一個代表陣列大小的唯讀變數或常數，之後在程式中都使用這個唯讀變數或常數來表示陣列大小及索引範圍，以減少程式碼出錯的情形，例如前面的範例可寫成：

宣告與定義陣列

```
const int asize = 5;          // 或用 #define 定義成巨集常數
...
int iArray[asize];            // 改用唯讀變數設定大小

for(int i=0;i<asize;i++)      // 用唯讀變數設定迴圈執行次數
...
```

設定陣列初值

- 我們可在定義變數時用指定運算子設定變數初始值，而陣列由於有許多元素，所以設定初始值的方式略有不同。要設定陣列元素初始值時，需用大括號括住各初始值，並從第 0 個元素依序開始設定，語法如下：

```
int iArray[x]={元素 0 的值, 元素 1 的值,...};
```

- 例如：

```
int iArray[5]={2,4,12,6,18}; // 相當於指定 iArray[0]=2、
                              // iArray[1]=4、iArray[2]=12、
                              // iArray[3]=6、iArray[4]=18
```

設定陣列初值

- 定義陣列及元素初始值時, 有幾點注意事項：

  1. 有設定初始值時, 可不指定陣列大小。此時元素個數就是大括號中的初始值個數, 例如 "int a[ ]={1,2,3};" 表示 a[ ] 的大小為 3 個元素。
  2. 若有指定陣列大小時, 則初始值的數量需小於或等於元素個數。只指定部份元素的初始值時, 未指定到初始值的元素其初始值一律為 0 (若是字元型別的陣列, 則為 '\0')；初始值的數量若超過元素個數, 則編譯時會出現錯誤。

- 以下就是定義陣列及元素初始值的範例：

設定陣列初值

程式 Ch07-02.cpp 定義陣列初始值

```
01 #include <iostream>
02 using namespace std;
03
04 int main()
05 {
06     int iArray[] = {1,2,3,4,5};   // 未指定陣列大小
07     int nArray[5] = {6,7,8};      // 未設定全部元素的初始值
08
09     cout << "iArray 陣列的大小爲 " << sizeof(iArray)
10          << " 個位元組" << endl;
11     cout << "nArray 陣列的大小爲 " << sizeof(nArray)
12          << " 個位元組" << endl;
13
```

設定陣列初值

```
14     for(int  i=0;i<5;i++)        //  用迴圈輸出各元素值
15       cout << "iArray[" << i << "] =" << iArray[i] << '\t'
16           << "nArray[" << i << "] =" << nArray[i] << endl;
17     }
18  }
```

執行結果

```
iArray 陣列的大小為 20 個位元組
nArray 陣列的大小為 20 個位元組
iArray[0] = 1      nArray[0] = 6
iArray[1] = 2      nArray[1] = 7
iArray[2] = 3      nArray[2] = 8
iArray[3] = 4      nArray[3] = 0
iArray[4] = 5      nArray[4] = 0
```

設定陣列初值

1. 第 6 行定義的 iArray 未指定陣列大小, 因此其元素個數依初始值個數定為 5。
2. 第 7 行定義的 nArray 陣列大小為 5, 但只指定 3 個元素值, 因此後 2 個元素的值為 0, 由執行結果可驗證之。
3. 第 9~12 行輸出以 sizeof() 運算子分別取得的 iArray 與 nArray 陣列大小。由於整數資料型別佔用 4 個位元組, 因此 5 個整數陣列元素即佔用 5x4=20 個位元組的記憶體空間。

設定陣列初值

4. 第 14~17 行再次以 for 迴圈分別依序輸出 iArray 與 nArray 中各元素的值。

- 至此我們已介紹了陣列的基本使用方法, 讀者可將陣列當成一般的變數來使用即可, 只不過要記得以下幾點差異：

設定陣列初值

- 陣列相當於多個同型變數的集合，集合中的每個元素就是一個變數。
- 陣列大小一經設定後就不能改變，但元素值則和變數一樣是隨時可更改的。
- 陣列變數不可以指定給另一個陣列變數，例如：

```
int a[] = {1,2,3};
int b[] = a[];    // 語法錯誤
```

陣列應用

- 認識陣列的基礎後，我們要進一步介紹幾項陣列的基本應用。由於陣列是一群資料的集合，所以常被用於需處理多筆同類型資料的情況，而在這種場合下有幾種應用是很常見的，例如搜尋某一筆特定的資料、尋找最大或最小的元素、將所有元素依指定的順序排列等等。以下簡介兩種陣列的應用，至於其它的應用方式也都很類似，讀者可自行舉一反三。

尋找陣列中的最大或最小值

- 在處理多筆資料時，時常需要找出其中的最大或最小值。最簡單的方法就是用迴圈將所有元素一一比較，全部都比對過後，即可找出最大或最小的元素。例如在下面的範例中，將在使用者輸入的 5 個數值中找出最

程式 Ch07-03.cpp 請使用者輸入 5 個數值，找出其中最大值

```
01 #include <iostream>
02 #define SIZE 5                          // 陣列大小常數
03 using namespace std;
04
```

尋找陣列中的最大或最小值

```
05 int main()
06 {
07    int numbers[SIZE];                // 儲存數值的陣列
08
09    cout << "請輸入 5 個數字, 程式將找出最大值" << endl;
10
11    for (int i=0;i<SIZE;i++) { // 用迴圈取得每個元素值
12     cout << "請輸入第 " << (i+1) << " 個數字:";
13     cin >> numbers[i];
14    }
15
16    int MAX = numbers[0];             // 用來儲存最大值的變數
17                                      // 先設爲第 1 個元素的值
18
```

尋找陣列中的最大或最小值

```
19    for (int i=1;i<SIZE;i++)      // 比對陣列中所有元素的迴圈
20       if (numbers[i]>MAX)        // 若 numbers[i] 大於 MAX
21          MAX = numbers[i];       // 則將最大值設為 numbers[i]
22
23    cout << "在輸入的數字中，數值最大的是 " << MAX;
24 }
```

執行結果

```
請輸入 5 個數字，程式將找出最大值
請輸入第 1 個數字：57
請輸入第 2 個數字：46
請輸入第 3 個數字：99
請輸入第 4 個數字：12
請輸入第 5 個數字：75
在輸入的數字中，數值最大的是 99
```

尋找陣列中的最大或最小值

1. 第 11~14 行的迴圈是用來控制請使用者輸入 5 個數字, 並依序存入 numbers。
2. 第 16 行定義代表最大值的變數, 並先將其值設為 numbers[0] 的值。
3. 第 19~21 行的迴圈是用來挑出最大值, 此迴圈會依序將 MAX 與陣列中 numbers[0] 以外的所有元素做比較, 若其值比 MAX 大, 就將它的值指定給 MAX。

將陣列內所有元素排序

- 所謂排序, 就是將一串隨意排列的同類型資料, 利用迴圈與條件式比較大小後, 將它重新依由小到大的昇冪或者由大到小的降冪方式排列。這個處理方式比前面單純找出最大 / 最小值更為實用, 因為排序後也可以很容易找出最大 /小值 (第 1 個或最後 1 個元素), 進一步延伸, 就能馬上變成找最大或最小的前數筆資料。

將陣列內所有元素排序

- 在排序方法中, 最普通的排序法稱為汽泡排序法 (Bubble Sort)。因為在排序過程中, 數值較大的元素其位置會漸漸的往前面移動, 就好像一個氣泡由水底浮到水面, 因為壓力減小, 氣泡的體積會慢慢變大一般, 這就是其名稱的由來。
- 以下程式是由一個包含 5 個元素的字元陣列, 利用氣泡排序法, 將 5 個字母由大到小排列, 程式如下：

將陣列內所有元素排序

程式 Ch07-04.cpp 將陣列中的資料降冪排序後，從螢幕輸出

```
01 #include <iostream>
02 #define SIZE 5                          // 陣列大小常數
03 using namespace std;
04
05 int main()
06 {
07    char array[SIZE]={'s','c','i','o','n'};
08
09    cout << "排序前：";
10    for (int i=0;i<SIZE;i++)       // 輸出排序前的陣列內容
11       cout << array[i];
12
13    for (int i=0;i<SIZE;i++)
```

將陣列內所有元素排序

```
14          for (int j=i+1;j<SIZE;j++)
15            if (array[i]<array[j])   // 若 array[i] 的值小於 array[j]
16            {                        // 就將 2 個元素的值對調
17              char temp = array[i];  // temp 是對調時用到的暫存變數
18              array[i] = array[j];
19              array[j] = temp;
20            }
21
22    cout << endl << "排序後：";
23    for (int i=0;i<SIZE;i++)         // 輸出排序後的陣列內容
24      cout << array[i];
25  }
```

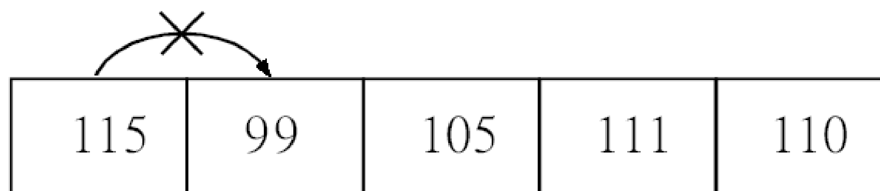
執行結果

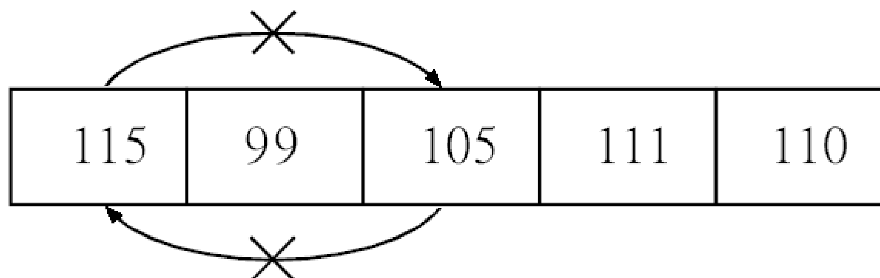
```
排序前: s c i o n
排序後: s o n i c
```

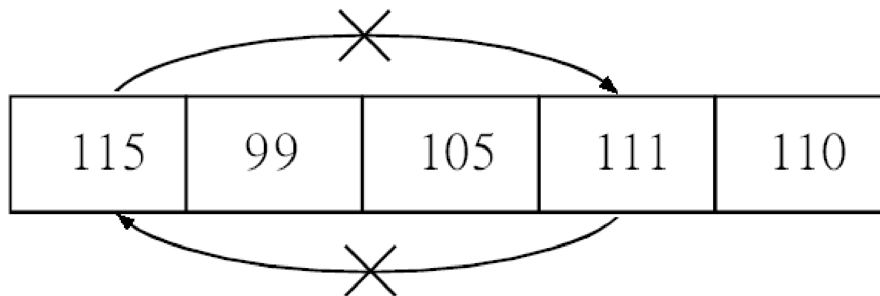
將陣列內所有元素排序

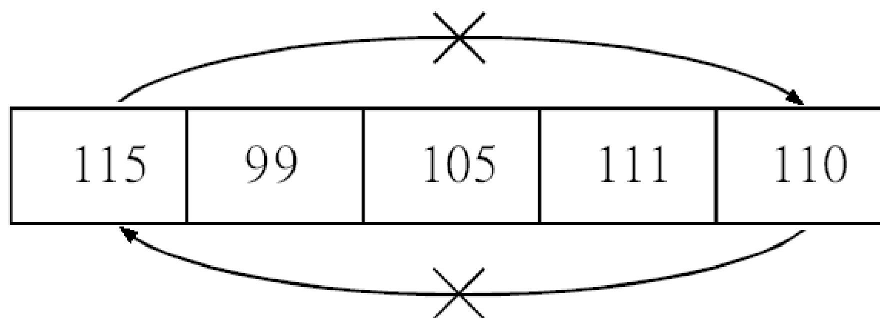
- 氣泡排序法會先取第 1 個陣列元素, 與所有陣列元素比較後, 如果第 1 個陣列元素比第 2 個小, 則交換位置, 反之則不換。i 為 0 時, 如下 圖 (字元大小的比較, 就是在比較其 ASCII 碼, s、c、i、o、n 的 ASCII 碼分別為 115、99、105、111、110)：

將陣列內所有元素排序



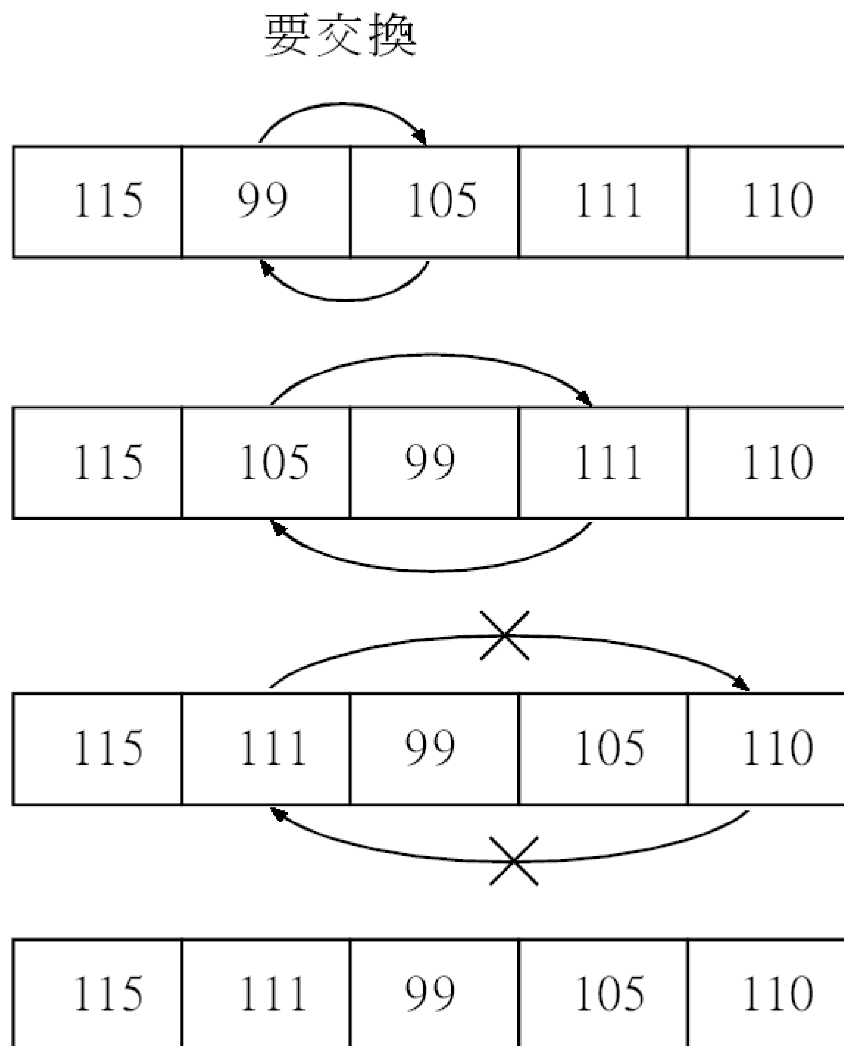






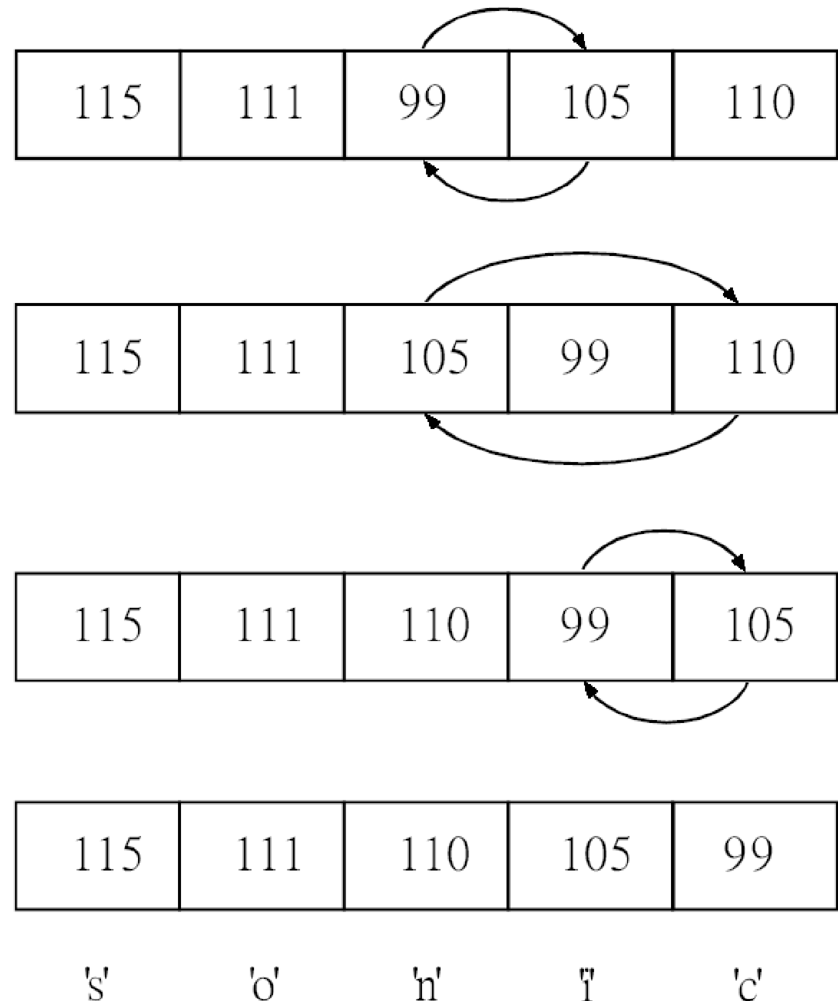
將陣列內所有元素排序

- 接著 i 為 1 時, 取第 2 個陣列元素與其後的所有元素比較, 如右圖：



將陣列內所有元素排序

- 以此類推，排序直到全部元素都比較過為止，如右圖：
- 若要將陣列改成由小到大的排列方式，只需將第 13 行的條件運算子改成大於即可。



7-2 字元陣列

- 前面的例子使用到了字元陣列, 其實字元陣列還有個特殊的用途, 就是當成字串 (String) 來使用。雖然這個用法是承襲自 C 語言, 而 C++ 本身也已專為字串的應用提供了專用的類別, 但將字元陣列當成字串使用仍相當常見, 因此本節就來簡介字元陣列的相關應用。

可當成字串的字元陣列

- 由於字串和字元陣列一樣都是一組字元的集合，所以我們可把字元陣列當成字串使用。但首先必須要認識的是，對 C++ 而言，到底什麼是字串？
- 我們已經學過，用雙引號括起來的文字就是一個常數字串，例如 "I Love You"，用如下的敘述就能將這 10 個字元 (含空白字元) 輸出到螢幕上：

```
cout << "I Love You";
```

可當成字串的字元陣列

- 但如果我們執行如下的敘述：

```
cout << sizeof("I Love You");     // 輸出字串 "I Love You" 的大小
```

- 這時候所得到的結果是 11 而非 10。這是因為對 C++ 編譯器而言，它在儲存 "I Love You" 這個字串時，除了儲存十個字元外，還在字串結尾加上一個代表字串結尾的 '\0' 字元 (ASCII 字碼 0，而非數字 '0')，所以 "I Love You" 在記憶體中所佔的空間是 11 個位元組而非 10 個位元組。

可當成字串的字元陣列

- 有了這個認識後，我們就知道字元陣列若要用來儲存字串也是一樣，必須在陣列中多出一個元素來儲存代表字串結尾的結束字元，否則就不能稱其為**字串**。在定義字元陣列時，可直接將字串內容當成初始值設定給陣列，此時就不必再用大括號將字串括起來，請參考以下範例。

程式 Ch07-05.cpp 輸出字串內容以及陣列大小

```
01 #include <iostream>
02 using namespace std;
```

可當成字串的字元陣列

```
03
04 int main()
05 {
06    char name1[]="John Smith";    // 以字串爲初始值
07    char name2[]={'M','a','r','y',' ','W','h','i','t','e'};
08                                  // 設定個別字元爲初始值
09    cout << "name1[]大小爲：" << sizeof(name1) << endl;
10    cout << "name2[]大小爲：" << sizeof(name2) << endl;
11
12    cout << "name1[]：" << name1 << endl;
13    cout << "name2[]：" << name2 << endl;
14 }
```

執行結果

```
name1[]大小爲：11
name2[]大小爲：10
name1[]：John Smith
name2[]：Mary White@
```

可當成字串的字元陣列

1. 第 6、7 行分別定義 2 個字元陣列, 但 name1[ ] 是直接以字串為其初始值；name2[ ] 則以分別指定各元素字元的方式設定初始值。
2. 第 9、10 行分別輸出兩個陣列的 sizeof( ) 運算結果, 雖然兩者的字數相同, 但 name1[ ] 最後還有個空白字元, 所以輸出結果為 11。
3. 第 12、13 行則是以陣列名稱直接輸出, 對 cout 而言, 若是字元陣列, 就會輸出其中所存放的字串內容。

可當成字串的字元陣列

- 在輸出結果中, 比較奇怪的是用 cout 輸出 name2 的結果："Mary White@", 最後面的字元 '@' 是從何而來的呢？其實以 "cout << 字元陣列名稱" 的方式輸出字元陣列的內容時, 原則上 cout 是把它當成字串來處理, 所以會輸出在結束字元 '\0' 之前的所有字元, 也就是整個字串的內容。

可當成字串的字元陣列

- 但本例中 name2 陣列並沒有結束字元, 所以程式會以為字串還沒結束, 就接著輸出記憶體中放在 "Mary White" 之後的內容, 直到遇到結束字元為止。在本例中, "Mary White" 之後先出現的是字元 ‘@’ 隨後就是字元 '\0', 所以很幸運地程式只輸出 "Mary White@" 就沒有再輸出其它奇怪的文字了。這是使用字元陣列時可能會遇到的問題, 必須小心處理, 否則會讓程式輸出一堆奇怪的內容。

標準函式庫對字元陣列的支援

- 前面提過陣列變數的一些限制, 例如陣列不能直接指定給另一個陣列, 所以若要把一個陣列的內容複製到另一個陣列, 就需將來源陣列中的元素, 用迴圈一個個指定給目的陣列的元素。

標準函式庫對字元陣列的支援

- 但使用字串時常會遇到需搬移、複製字串或其它類型的應用，若都要自己用迴圈來一一處理，實在很不方便，因此在標準函式庫中就提供了許多字串相關的函式，這些函式的宣告都放在 <cstring> 含括檔中，因此含括 <cstring> 後即可在程式中呼叫它們。以下簡介 <cstring> 中幾個函式的用法，讓讀者有初步的認識，至於其它函式的用法，讀者可自行參考整合開發環境的線上說明。

字串長度

- 所謂字串長度就是字串中所含的字元數 (不含結束字元), 有時候程式要讓使用者輸入一些資料, 接著要檢查使用者輸入的字元數, 就會用到可計算字串長度的 strlen() 函式。呼叫此函式時需以字串 (字元陣列) 為參數, 函式會傳回 unsigned int 型別的字元數 (中文字會被視為 2 個字), 如範例所示：

字串長度

程式 Ch07-06.cpp 檢查字串長度

```
01 #include <iostream>
02 #include <cstring>    // 因爲用到 strlen(), 故含括此檔案
03 using namespace std;
04
05 int main()
06 {
07   char name[80];
08
09   cout << "請輸入一字串：";
10   cin.getline(name,80);       // 讓 cin 可取得一整行的輸入內容
11
12   cout << "sizeof(name) = " << sizeof(name) << endl;
13   cout << "strlen(name) = " << strlen(name) << endl;
14 }
```

執行結果

```
請輸入一字串：Taipei City
sizeof(name) = 80
strlen(name) = 11
```

字串長度

1. 第 10 行用 cin 物件呼叫 getline() 函式, 此函式的功能是讓 cin 能取得一整行的輸入內容, 便於我們輸入中間含空白字元的字串。因為依 cin 預設的運作方式, 在讀取輸入時, 若遇到空白就會停止讀取, 所以若輸入如上例中的 "Taipei City" 時, cin 只會讀到前半的 "Taipei"。
2. 第 12、13 行分別輸出陣列的 sizeof() 運算結果及呼叫 strlen() 查看字串長度的結果。

字串長度

- 由輸出結果即可查覺 sizeof() 和 strlen() 傳回值意義的不同, sizeof() 傳回的是變數的大小, 由於宣告 name 陣列時即宣告其可存放 80 個字元, 因此傳回 80 (位元組) 但 strlen() 則是計算字串的長度, 也就是結束字元 '\0' 之前有多少個字元, 所以即使 name 陣列足以容納 80 個字元, 但呼叫 strlen() 函式時, 它所存放的字串 "Taipei City" 連空白只有 11 個字元, 所以 strlen() 傳回的數值為 11。

字串複製

- 在處理變數時, 我們可以用指定運算子 (=), 把變數 A 的值直接指定給變數 B, 如 B=A。但陣列不能做這樣的操作, 所以我們也不能透過把 A 字串指定給 B 字串的方式來作字串的複製。因此標準函式庫提供 2 個字串複製函式：

```
char*  strcpy(目的字串,來源字串);
char*  strncpy(目的字串,來源字串,要複製的字元數);
```

字串複製

- 這 2 個函式都會傳回指向目的字串的指標, 其間的差別在於 strncpy() 可做**局部**的複製, 因此可用第 3 個參數指定要複製的字元數。其實參數中的目的字串、來源字串指的都是字元陣列, 由於來源與目的陣列的大小可能不同、來源陣列中的字串長度不確定等因素, 在使用上述函式時要注意幾點：
  1. 如果複製的字元數大於目的陣列的長度時, 可能會使超出的部份覆蓋到已被使用的記憶體, 而使程式出錯。

字串複製

2. 如果複製的字元數小於目的陣列的長度, 則目的陣列中超出字串長度的部份不會受影響。
3. 用 strncpy() 複製含中文的字串時, 要注意每個中文字要視為 2 個字元, 因為中文字存放在字元陣列時, 每個字的編碼會佔 2 個元素的空間。因此要注意指定第 3 個參數時, 若恰好是某個中文字的後半所在的位置, 將使該字元只複製一半的編碼, 導致目的字串出現奇怪的內容。

字串複製

- 所以，如果要讓字串內容正確地複製，無論目的陣列內是否已經存有其它內容，其容量一定要足夠容納來源陣列中的字串。如以下程式中，我們要將第 1 個字串內容複製到第 2 個字串時，第 2 個字串的容量就比第 1 個字串大，程式如下：

字串複製

程式 Ch07-07.cpp 複製全部或局部字串

```
01 #include <iostream>
02 #include <cstring>
03 using namespace std;
04
05 int main()
06 {
07    char str1[]="Lazy Boy";
08    char str2[]="Cute Girl";
09    char str3[]="Pink Panther";
```

字串複製

```
10    cout << "第 1 個字串：" << str1 << '\n'
11         << "第 2 個字串：" << str2 << '\n'
12         << "第 3 個字串：" << str3 << endl;
13
14    strcpy(str2,str1);      // 將字串1 複製到字串 2
15    cout << "將第 1 個字串全部複製到第 2 個字串：" << '\n'
16         << "第 2 個字串：" << str2 << endl;
17
18    int n;
19    cout << "要複製第 1 個字串的前幾個字元到第 3 個字串：";
20    cin >> n;
21    strncpy(str3,str1,n); // 將字串 1 的前 n 個字元複製到字串 3
22    cout << "將第 1 個字串前 " << n << " 個字複製到第 3 個字串："
23         << "\n第 3 個字串：" << str3;
24 }
```

字串複製

執行結果

```
第 1 個字串：Lazy Boy
第 2 個字串：Cute Girl
第 3 個字串：Pink Panther
將第 1 個字串全部複製到第 2 個字串：
第 2 個字串：Lazy Boy
要複製第 1 個字串的前幾個字元到第 3 個字串：4
將第 1 個字串前 4 個字複製到第 3 個字串：
第 3 個字串：Lazy Panther
```

字串複製

- 程式共定義了三個含不同字串的字元陣列, 然後將第 1 個字串分別用 strcpy() 及 strncpy() 複製給另 2 個字串, 並請使用者輸入複製局部字串時, 要複製的字元數。

字串的比對

- 陣列不僅無法用指定運算子指定其值, 也無法使用比較運算字來比較其內容：

```
char  str1[],str2[];
...
str1  ==  str2      //   這三個比較運算式
str1  >   str2      //   都不是在比較陣列
str1  <   str2      //   元素所存放的內容
```

字串的比對

- 上列的比較運算式雖然合法, 但它們都不是在比較陣列的內容, 而是在比較 2 個變數的位址, 因此這種比較基本上是沒什麼意義的。然而在字串應用上免不了要做字串的比對及比較, 此時可使用下列 2 個函式：

```
int strcmp(字串1, 字串2);
int strncmp(字串1, 字串2, n);    // 比較前 n 個字元
```

字串的比對

- 兩個函式都是傳回參數字串**相減**的數值，也就是說，函式會取兩字串的第一個字元相減，如果差為 0 則繼續比較下一個字元，直到差不為 0 時，便傳回差值。所以，傳回值可能有 3 個結果：

```
strcmp(a,b) == 0 // 傳回值爲 0, 字串 a 與字串 b 內容相同
strcmp(a,b) > 0  // 傳回值大於 0, 字串 a 大於字串 b
strcmp(a,b) < 0  // 傳回值小於 0, 字串 a 小於字串 b
```

字串的比對

- strncmp() 的功能及用法也相似，不過 strncmp() 只會比較第 3 個參數所指定的前 n 個字元 (中文字同樣是視為 2 個字)，而不會比較到字串結束。這兩個函式的使用方式請參見以下範例：

程式 Ch07-08.cpp 比對兩字串的內容是否相同

```
01 #include <iostream>
02 #include <cstring>
03 using namespace std;
04
```

字串的比對

```
05 int main()
06 {
07    char str1[60];     //  先宣告兩個用以存放
08    char str2[60];     //  使用者輸入字串的陣列
09
10    cout << "請輸入第 1 個字串：";
11    cin.getline(str1,80);          //  讓 cin 可取得一整行的輸入內容
12    cout << "請輸入第 2 個字串：";
13    cin.getline(str2,80);          //  讓 cin 可取得一整行的輸入內容
14
15    if(strcmp(str1,str2) == 0) //  比對 str1、str2 的內容是否相同
16       cout << "兩次輸入的字串的內容相同";
17    else
18       cout << "兩次輸入的字串內容不同";
19 }
```

字串的比對

執行結果

請輸入第 1 個字串：百尺竿頭　更進一步
請輸入第 2 個字串：百尺竿頭　更近一步
兩次輸入的字串的內容不同

執行結果

請輸入第 1 個字串：全台首學
請輸入第 2 個字串：全台首學
兩次輸入的字串的內容相同

字串的比對

- 程式請使用者輸入 2 個字串後, 即在第 15 行呼叫 strcmp() 函式比較兩個字串, 並比對傳回值是否為 0 (表示兩個字串相等), 輸出對應的訊息。
- strncmp() 的用法也大致相同, 只不過需在第 3 個參數指定要比對的是前幾個字元, 在此就不多做介紹。

7-3 多維陣列

- 由多個陣列組成的多維陣列
- 二維陣列的初始值設定
- 以陣列之陣列的形式設定初始值
- 定義時省略維度值

由多個陣列組成的多維陣列

- 如果把教室的一排座位看成是陣列, 每個位子上可坐一位同學, 就好像每個元素可容納一筆資料, 若一排可坐 6 個人, 就是有 6 個元素的陣列。如果教室有多排座位, 譬如說總共有 7 排, 這時我們可將它視為是有 7 個元素的陣列, 且這個陣列中的元素本身又是有 6 個元素的陣列, 此時我們稱這個陣列為**二維**陣列。

由多個陣列組成的多維陣列

第 1 排：

第 2 排：

第 3 排：

第 4 排：

第 5 排：

第 6 排：

第 7 排：

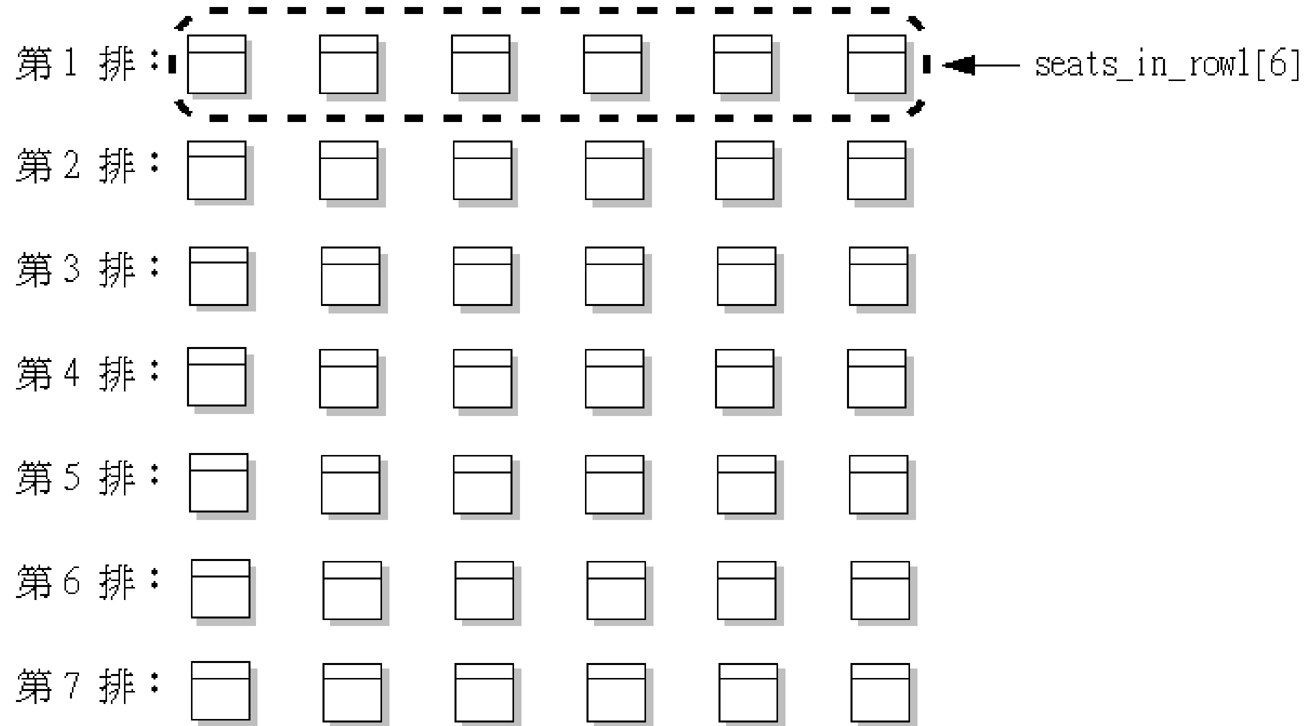
每一排本身是含 6 個元素的陣列 → 一維陣列

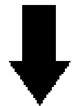
把這 7 排集合成一個陣列 → 二維陣列

由多個陣列組成的多維陣列

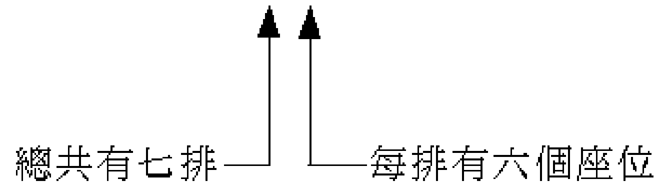
- 二維陣列就是指元素的索引是有兩個維度。對一排座位 (一維陣列) 我們只需說是第幾個位子, 即可指出座位 (元素) 的位置；對一整間教 室 (二維陣列), 就必須指明是第幾列、第幾 行 (兩個維度), 才能明確指出座位 (元素) 的位置。若用 C++ 的語法來看, 就是在陣列名稱後要 用 2 個中括號來表示：

由多個陣列組成的多維陣列





把這 7 排集合成一個陣列 ⟶ all_seats[7][6] 總共有 7x6 = 42 個元素



由多個陣列組成的多維陣列

```
資料型別  陣列名稱[n][m];        //  二維陣列的宣告語法
                                 // 可看成是有 n 列、m 行
```

- 以上就是宣告二維陣列的語法, 在這個陣列中總共有 n*m 個元素。二維陣列的其它性質都和一維陣列類似, 例如元素索引都是從 0 開始、指定元素時需指明 2 個中括號中的索引值等等。

由多個陣列組成的多維陣列

- 同理, 若一層樓有 8 間教室, 我們又可將這 8 間教室視為一 有 8 個元素的陣列, 其中每個元素又是一個有 7 個元素 (每間教室有 7 排座位) 的陣列, 而這個代表一層樓座位的陣列則稱為**三維**陣列；再進一步推演, 若教學大樓有 4 層樓, 則每一樓又可以視為一個陣列元素, 組成另一個**四維**陣列。凡是維度超過 1 的陣列, 都通稱為多維陣列。

由多個陣列組成的多維陣列

```
int floor[8][7][6];        // 共有 8x7x6 = 336 個元素 int building[4][8][7][6];
                           // 共有 4x8x7x6 = 1344 個元素
```

- 不過一般實用上很少使用超過二維的陣列，因為一來操作不方便，二來在寫程式時也容易因混淆而出錯，因此本節也只以二維陣列的應用為主。其它多維陣列的應用，都可類推。

二維陣列的初始值設定

- 二維陣列的宣告方式只是在變數名稱後多一組中括號，並不困難。但如果要定義其初始值，由於二維陣列具有二個維度，那初始值的設定順序是如何呢？

```
                    a[0][0] 的初始值
int a[2][3] = {1, 2, 3, 4, 5, 6}; // 共有 6 個元素
                    是 a[0][1] 還是 a[1][0]？
```

以陣列之陣列的形式設定初始值

- 回顧一下前面提到的，我們可將多維陣列看成是由陣列組成的陣列，所以在寫初始值時，就可以用**大括號中有大括號**的方式將初始值分隔開來，這樣就很容易分辨了。例如 a[2][3] 是**兩個子陣列，每個子陣列有 3 個元素**，所以可寫成：

```
int a[2][3] = {{1, 2, 3},       // a[0][0]、a[0][1]、a[0][2]
               {4, 5, 6}};      // a[1][0]、a[1][1]、a[1][2]
```

以陣列之陣列的形式設定初始值

- 這樣寫就很清楚了，也不會弄混。當然把內層的大括號拿掉，對程式也不會有任何影響，C++ 編譯器會依序先將初始值指定給 a[0][x] 的元素，再指定給 a[1][x] 的元素，依此類推。同樣的，若初始值數量不夠時，未指定的元素其初始值一律為 0。

以陣列之陣列的形式設定初始值

程式 Ch07-09.cpp 定義二維陣列

```
01 #include <iostream>
02 using namespace std;
03
04 int main()
05 {
06     int iArray[3][4] = {1,2,3,4,5,6};
07
08     cout << "iArray 陣列的大小為 " << sizeof(iArray)
09          << " 個位元組" << endl;
10
11     for(int i=0;i<3;i++) {    // 用巢狀迴圈遊歷所有的元素
12         for(int j=0;j<4;j++)
```

以陣列之陣列的形式設定初始值

```
13          cout << "iArray[" << i << "][" << j << "]= "
14                << iArray[i][j] << '\t';
15
16       cout << endl;              // 每輸出完一列元素就換行
17    }
18 }
```

執行結果

```
iArray 陣列的大小為 48 個位元組
iArray[0][0]= 1 iArray[0][1]= 2 iArray[0][2]= 3 iArray[0][3]= 4
iArray[1][0]= 5 iArray[1][1]= 6 iArray[1][2]= 0 iArray[1][3]= 0
iArray[2][0]= 0 iArray[2][1]= 0 iArray[2][2]= 0 iArray[2][3]= 0
```

以陣列之陣列的形式設定初始值

1. 第 6 行定義可存放 3 列、4 的二維陣列 iArray, 但只給了 6 個初始值。
2. 第 8 行輸出以 sizeof() 運算子取得的 iArray 陣列大小。由於整數資料型別佔用 4 個位元組, 因此 4x3=12 個整數陣列元素即佔用 48 個位元組的記憶體空間。
3. 第 11~16 行利用巢狀的 for 迴圈依序輸出 iArray 各元素的值。第 11 行的 for 迴圈調整**列**的變化；第 12 行的迴圈則是做**行**的變化。

以陣列之陣列的形式設定初始值

4. 第 16 行的敘述會在每輸出一列元素值後, 即換行輸出。

- 讀者可看到, 二維陣列也可使用迴圈來操作、存取陣列元素, 只不過需使用巢狀迴圈才能遊歷整個陣列中的所有元素。若維數愈大, 所需的巢狀迴圈也要愈多層。同樣的要在處理時, 注意迴圈的條件運算式內容, 避免在操作時不小心超出陣列元素的索引範圍。

定義時省略維度值

- 在定義一維陣列時，可以省略方括號中的維度大小，讓編譯器自動依大括號內的初始值數量來決定有幾個元素。定義二維陣列時，基本上也可以使用相同技巧，但要注意的是，為避免編譯器無法判斷各維度的大小，所以只能省略最左邊的維度大小數值：

```
int a[][3] = {1, 2, 3, 4, 5, 6};
int b[][ ] = {1, 2, 3, 4, 5, 6};     // 編譯器無法判斷是幾列幾行
                                     // b[2][3]、b[3][2]?
int c[3][] = {1, 2, 3, 4, 5, 6};     // 編譯器無法判斷是每列有幾行
                                     // c[3][2]、c[3][3]...?
```

7-4 指標與參照

- 除了陣列以外, C++ 還有兩種特別的資料類型, 稱為指標 (pointer) 與參照 (reference)。本節先來認識指標與參照, 及其與陣列的關係, 下一節則要介紹這兩種資料型別在函式參數上的應用。

宣告與使用指標變數

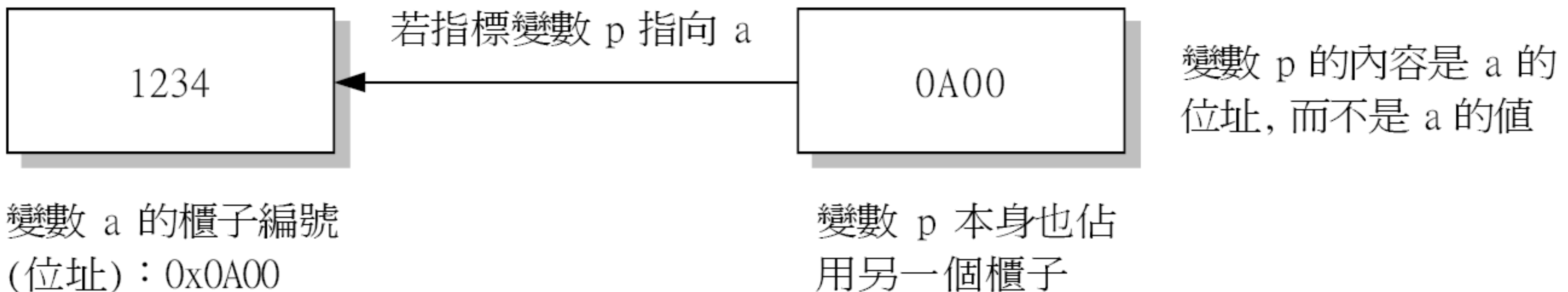
- 指標 (pointer) 也是一種變數, 但是它所儲存的並非是變數的值, 而是記憶體中的一個位址, 例如某個變數實際存放在記憶體中的位址。當指標儲存一個位址時, 我們稱此指標**指向**該位址所表示的記憶體空間。

宣告與使用指標變數

- 何謂位址？我們可以把記憶體的儲存空間，想像是一個一個排列整齊可用來裝填資料的小格子，每個小格子的大小都相等 (1 位元組)而位址就是用來區別這些小格子，也就是這些格子的代號。就好像車站中公用的儲物櫃，每個小格子都有一個編號一樣，由於這些編號有次序性，所以透過編號就能找到櫃子的位置，進而取出放在櫃中的物件。

宣告與使用指標變數

- 因此在 C++ 中的變數，就像是用來代替櫃子編號的名稱，變數中存放的是我們要用到的資料；而指標變數則是用來記錄櫃子號碼的變數，它所存放的是某筆資料所在的櫃子編號，而非資料本身。例如以下的示意圖：



宣告與使用指標變數

- 宣告指標變數的方式很簡單, 只需在變數名稱前加上一個 * 符號：
  1. **資料型別**：指標**所指變數**的型別, 必須與該指標所指向的變數型別相同。
  2. *：稱為指位運算子 (dereference operator), 而非乘法運算子。在宣告變數時, 用來表示變數為指標變數；若用在一般敘述中, 則表示要傳回指標所指的變數值。
  3. **指標變數名稱**：這個變數的名稱。

宣告與使用指標變數

- 宣告指標變數後，但要如何將其它變數的位址設給指標變數？我們要如何知道變數存放在記憶體中的位址？很簡單，只要在變數名稱前加上取址運算子 &，就會傳回該變數的位址。比如說：

```
int  *ptr,number;      //  宣告指標  ptr  與變數  number
ptr = &number;         //  讓指標  ptr  指向變數  number  的位址
```

宣告與使用指標變數

- 如此一來, ptr 便是指向 number 的指標了。
- 指標是用來存取變數空間的位址, 所以不管指標宣告成指向何種型別的變數, 編譯器都是配置相同大小的空間給指標變數, 在目前一般個人電腦上, 指標變數通常都是佔用 4 個位元組的空間 (在 64 位元系統並搭配 64 位元編譯器, 則指標將佔 8 個位元組) 以下就是查看指標變數各項資訊的範例：

宣告與使用指標變數

程式 Ch07-10.cpp 指標變數的內容及大小

```
01 #include <iostream>
02 using namespace std;
03
04 int main()
05 {
06   int    i;              // 宣告整數及變數
07   double d;              // 宣告倍精度浮點數變數
08   int    *iptr = &i;  // 定義指標變數, 並將初始值設為 i 的位址
09   double *dptr = &d;  // 定義指標變數, 並將初始值設為 d 的位址
10
11   cout << "iptr 的大小為：" << sizeof(iptr) << endl;
12   cout << "iptr 存的值為：" << iptr << endl;
```

宣告與使用指標變數

```
13
14    cout << "dptr 的大小爲：" << sizeof(dptr) << endl;
15    cout << "dptr 存的值爲：" << dptr << endl;
16 }
```

執行結果

```
iptr 的大小爲：4
iptr 存的值爲：0012FF70
dptr 的大小爲：4
dptr 存的值爲：0012FF64
```

宣告與使用指標變數

1. 第 6、7 行分別宣告整數變 數 i 及倍精度浮點數變數 d。
2. 第 8 行定義指向整數變數的指標 iptr, 並用取址運算子將 i 的**位址**設為其初始值。
3. 第 9 行定義指向倍精度浮點數變數的指標 dptr, 並用取址運算子將 d 的**位址**設為其初始值。
4. 第 11、14 行分別用 sizeof() 運算子顯示指標變數的大小, 結果顯示一律為 4 位元組。

宣告與使用指標變數

5. 第 12、15 行直接輸出指標變數的值, 也就是它們所指的位址值。cout 在輸出指標變數時, 預設會用 16 進位的方式顯示。

- 請注意, 設定指標變數的值時需使用同型別的變數位址, 例如以下的指定方式是錯誤的:

```
double *dptr = &i; // 編譯錯誤,
不能將整數變數的位址指定給 double 指標
```

存取指標變數所指的值

- 指標變數存的是記憶體位址, 那我們要如何透過指標來存取該位址的變數值？要透過指標來存取變數值, 需使用指位運算子 *, * 也稱為間接 (indirection) 運算子, 意指透過指標存取變數值時, 是**間接**的存取, 不像用變數可**直接**存取到變數值。* 的用法是放在指標變數名稱之前, 此時它就代表**指標所指位址中的資料**, 例如：

存取指標變數所指的值

```
int i = 3;
int *ptr = &i; // 讓 ptr 指向變數 i
cout << *ptr;   // *ptr 就是 i 的值
                // 因此爲輸出 3
*ptr = 9;       // 將 ptr 所指的變數值改爲 9
                // 也就是 i 變成 9
```

- 將指標變數套上間接運算子, 就相當於存取指標所指的變數一樣, 此時我們可以取得變數的值, 也可以更改其值。以下範例示範透過指標存取變數值的情形：

存取指標變數所指的值

程式 Ch07-11.cpp 透過指標變數『間接』存取變數值

```
01 #include <iostream>
02 using namespace std;
03 
04 int main()
05 {
06    int    *iptr,age=18;                 // 宣告整數型別指標與變數
07    float  *fptr,weight=65.05f;          // 宣告浮點數型別指標與變數
08    char   *cptr,bloodtype='O';          // 宣告字元型別指標與變數
09    iptr=&age;                           // iptr 指向 age 的位址
10    fptr=&weight;                        // fptr 指向 weight 的位址
11    cptr=&bloodtype;                     // cptr 指向 bloodtype 的位址
12 
```

存取指標變數所指的值

```
13     cout << "年齡：" << *iptr << "歲" << endl;
14     cout << "體重：" << *fptr << "公斤" << endl;
15     cout << "血型：" << *cptr << "型" << endl;
16 }
```

執行結果

```
年齡：18 歲
體重：65.05 公斤
血型：O 型
```

存取指標變數所指的值

- 使用指標存取變數值時，也可以做轉型，例如用整數指標參與運算，但需得到含小數點的結果，就可用以下方式：

```
int i = 100;
int *ptr = &i;
cout << ((float) *ptr /7); // 計算 100 除以 7
```

指標與陣列

- 陣列與指標有著密不可分的關係，而陣列的[ ]符號和指標的 * 符號也有異曲同工之妙。指標變數記錄著所指變數的記憶體位址，而在程式中，單寫陣列名稱時，它也代表著陣列中**第一個元素的位址**。因此：

```
char a[] = "data";
char *p = a;        // a 代表陣列的起始位址
                    // 相當於 *p = &a[0]
cout << a[0];       // 輸出 'd'
cout << *p ;        // 輸出 'd'
```

指標與陣列

- 由於陣列名稱本身所代表的意義，就是該陣列的起始位址 (即第一個元素的位址)，所以在上例中我們可以將 a 指定給 p，然後 p 便指向了 a 陣列中的第一個元素。當然，我們也可以用 &a[0] 來求得第一個陣列元素的位址，而這個位址值和 a 的值是一樣的 (均為陣列的起始位址)。

指標與陣列

- 更進一步，我們可以把指標的值加上元素的索引值，這時就等於由指標存取陣列元素了：

```
char a[] = "data";
char *p = a;        // a 代表陣列的起始位址
                    // 相當於 *p = &a[0]

cout << *p ;        // 輸出 a[0], 也就是 'd'
cout << *(p+1);     // 輸出 a[1], 也就是 'a'
cout << *(p+2);     // 輸出 a[2], 也就是 't'
cout << *(p+3);     // 輸出 a[3], 也就是 'a'
```

指標與陣列

- 由這個例子可發現, * 和 [ ] 都是 "依址取值" 的意思, 而 [ ] 中的編號就是依**陣列位址 + 索引**取值之意, 所以對應到指標, 就變成**指標 + 索引**。我們甚至可以張冠李戴, 將 [ ] 用於指標之上, 而將 * 用於陣列名稱之上, 請參考以下的範例：

程式 Ch07-12.cpp 陣列與指標的互換

```
01 #include <iostream>
02 #include <cstring>
03 using namespace std;
```

指標與陣列

```
04
05 int main()
06 {
07    char str[]="How are you?";
08    char *ptr = str;
09
10    for (unsigned i=0;i<strlen(str);i++)
11       cout << *(str+i); // 將陣列名稱 str 當成指標
12    cout << endl;
13
14    for (unsigned i=0;i<strlen(ptr);i++)
15       cout << ptr[i];    // 將指標 ptr 當成陣列名稱使用
16 }
```

執行結果

```
How are you?
How are you?
```

指標與陣列

1. 第 8 行定義 ptr 字元指標並將它指向 str 字元陣列。
2. 第 10、11 行的迴圈將 str 當成字元指標使用, 並每次輸出 (str+i) 位址所存放的字元, 因此會輸出整個 "How are you?" 字串的內容。
3. 第 14、15 行的迴圈則反其道而行, 將指標變數 ptr 當成陣列名稱使用, 並從第 0 個元素開始, 每次輸出第 i 個元素, 結果也是輸出整個 "How are you?" 字串的內容。

指標與陣列

- 將指標的數值做加減時，就相當於在對位址值做加減。但要特別注意一點，將指標加減 1，並不代表位址值加減 1，而是**索引值加減 1**，至於位址值則是**加減 1 個指標所指型別的大小**。舉例來說，在上例中，指標定義為 char 型別，由於 char 的大小是 1 個位元組，所以對指標做加減，就是以 1 個位元組為單位做加減。然而若指標為 int 型別，則對指標做加減時，一次是以 4 個位元組為單位做加減。其實只要將指標加減法想成是陣列索引值增減就很容易瞭解，我們再用以下範例驗證：

指標與陣列

程式 Ch07-13.cpp 指標的加減

```
01 #include <iostream>
02 using namespace std;
03
04 int main()
05 {
06    int a[5] = { 1,22,333,4444,55555};
07    int *ptr = a;
08
09    for(int i=0;i<5;i++)
10       cout << "指標 ptr+" << i << " 的位址：" << (ptr+i)
11             << "\t所指的記憶體存放的資料爲：" << *(ptr+i) << endl;
12 }
```

指標與陣列

執行結果

```
指標 ptr+0 的位址：0012FF60    所指的記憶體存放的資料爲：1
指標 ptr+1 的位址：0012FF64    所指的記憶體存放的資料爲：22
指標 ptr+2 的位址：0012FF68    所指的記憶體存放的資料爲：333
指標 ptr+3 的位址：0012FF6C    所指的記憶體存放的資料爲：4444
指標 ptr+4 的位址：0012FF70    所指的記憶體存放的資料爲：55555
```

指標與陣列

- 如執行結果所示, 將 int 指位器的值加 1 時, 位址增加的值為 4, 符合 int 型別的大小, 如此才能指到下一個元素的位址並取得其值。因此程式中對指位器的運算式 ptr + i, 實際上會變成如下的形式：

```
ptr + ( i * sizeof(*ptr) )  //  加上『i 乘上 ptr 所指型別的大小』
```

指標與陣列

- 雖然此處我們將指標的位址做數值運算，但並不表示指標可做所有類型的數值運算，例如位址乘位址、位址乘索引等都是無意義的計算。基本上只有下列三種指標 (位址) 的數值運算是有意義的：

操作	功能	運算結果
位址 ± N	將位址往後或往前移 N 個元素	位址
位址 - 位址	求出二個位址之間相距多少元素	整數
位址 op 位址 (op 爲比較運算子)	求出二位址值的大小關係	假或真

字元陣列與指標

- 當我們要定義一個二維的字元陣列來存放多個字串時, 也可用一個指標來指向它, 並透過指標做相關的操作, 例如下個這個簡單

```
char a[3][4] = { "abc", "def", "ghi" };
```

- 上例的 a 是一個二維陣列, 其內部包含 3 個一維陣列：a[0]、a[1] 和 a [2], 而每個一維陣列又包含 4 個元素。如果我們直接操作 a[0]、a[1] 或 a[2], 那麼它們就代表其個別一維陣列的起始位址：

字元陣列與指標

程式 Ch07-14.cpp 以指標加減存取二維字元陣列中的字串

```
01 #include <iostream>
02 using namespace std;
03
04 int main()
05 {
06    char a[3][4] = { "abc", "def", "ghi" };
07    char (*str)[4] = a;          // 將二維陣列轉型爲陣列指標
08
09    for(int i=0;i<3;i++)
10       cout << "指標 str+" << i << " 的位址：" << (str+i)
11            << "\ta[" << i << "]的位址：" << &a[i] << endl;
12    cout << endl;
```

字元陣列與指標

```
13
14     for(int  i=0;i<3;i++)          //  將三個字串接續輸出
15         cout << str[i];
16 }
```

執行結果

```
指標 str+0 的位址：0012FF64          a[0]的位址：0012FF64
指標 str+1 的位址：0012FF68          a[1]的位址：0012FF68
指標 str+2 的位址：0012FF6C          a[2]的位址：0012FF6C

abcdefghi
```

字元陣列與指標

- 程式先宣告一個含 3 個字串的二維陣列, 接著將定義一個指向陣列的指標, 隨後操作該指標輸出位址值做驗證並顯示 3 個字串。
- 請注意第 7 行的敘述, 我們定義的是 "char (*str)[4]" 而非 "char *str[4]", 這兩者的意義是不同的。由於 [] 的運算子的優先順序高於 *, 所以第 2 種寫法是**宣告一個含 4 個元素的陣列, 每個元素是個字元指標**, 所以可稱為指標陣列 (指標組成的陣列)至於我們在程式中的寫法,

字元陣列與指標

因為將 * 與變數名稱用括號括起來, 而 [4] 的優先順序在後, 所以 str 代表的是一個**指標變數, 而它所指的則是一個含 4 個字元的陣列**, 因此我們稱之為陣列指標 (指向陣列的指標)。

- 由於 str 所指的是含 4 個字元的陣列, 表示它所指的型別大小為 4 個位元組, 所以我們將它加上索引值時, 位址值每次都會加 4, 由執行結果中即可驗證。

可節省儲存空間的指標陣列

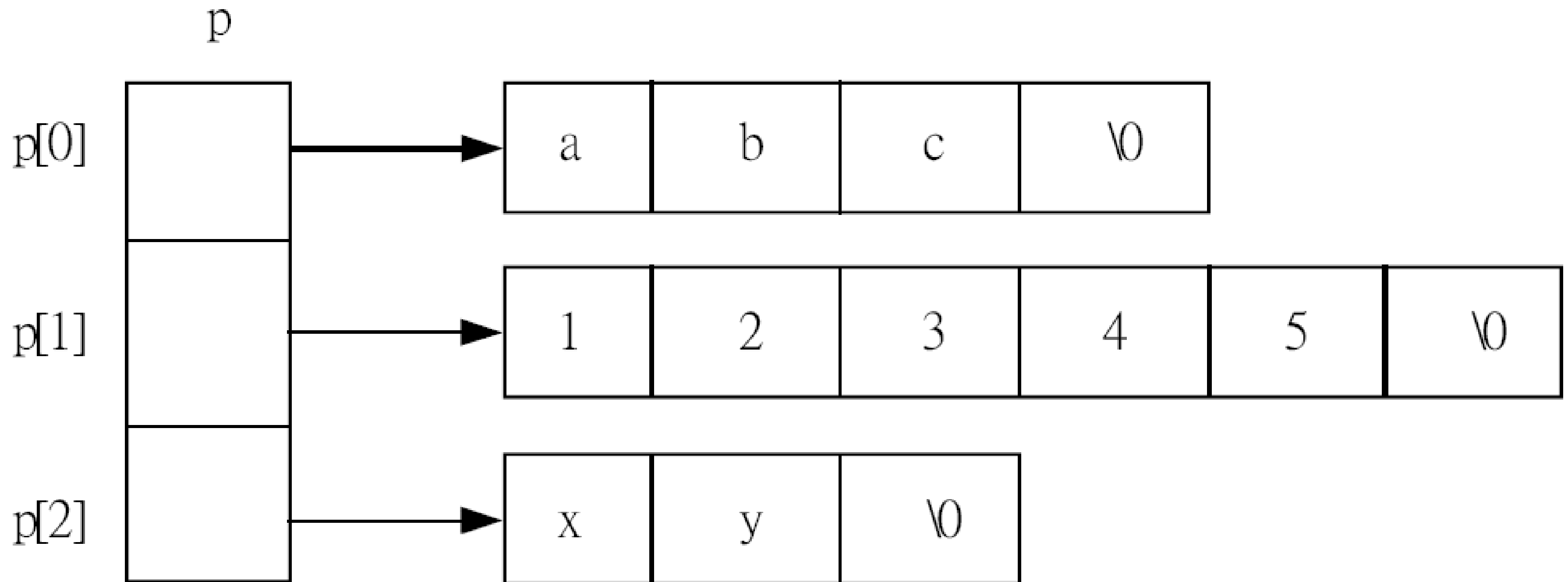
- 指標陣列是指構成陣列的元素都是指標，其定義或宣告的方式如下例：

```
char *p[3];            // 也可以寫成  char *(p[3]);
                       //         或  (char *) p[3];

p[0] = "abc";          // 陣列元素 p[0], p[1], p[2]
p[1] = "12345";        // 的值可任意更改, 使其指
p[2] = "xy";           // 向別的地方
```

可節省儲存空間的指標陣列

- 我們將每個陣列元素圖解如下：



可節省儲存空間的指標陣列

- 上面所定義的指標陣列內有 3 個元素, 每個元素都是指標, 然後我們便可將各指標指向不同的字串。由 於 p 是一個陣列名稱, 所以 p 本身的值不可更改, 但其內各元素的值則可以更改 (p[0], p[1], p[2])。
- 在定義指標陣列時也可以給定初值, 而且 [] 中的數目可以省略而交由編譯器去計算。

```
char *ptr[]={"happy","friday","fun"};
```

動態記憶體配置

- 使用指標時, 並不是只能將指標指向已宣告或定義的變數, 我們也可用**動態記憶體配置** (dynamic memory allocation) 的方式, 取得一塊記憶體空間給指標使用, 讓指標所指的是自己專用的空間。
- 前面所學的變數及陣列, 經過編譯器編譯後, 它們所能使用的記憶體空間就已固定, 不能變更。例如宣告了 10 個元素的整數陣列, 我們就只能用它來儲存 10 個整數資料, 不能儲存超過陣列容量的資料。

動態記憶體配置

- 而動態記憶體配置則不同，它是由程式在執行過程中，依當時的實際需要臨時向作業系統要求一塊沒有被其他程式使用的記憶體空間，而且要求的空間大小，可以在執行時才由程式決定。在程式執行過程中，若取得的記憶體已用不到了，也可以隨時將之釋放(release)，還給作業系統，讓作業系統可將記憶體空間提供給其它有需要的程式使用，如此可大幅的提高記憶體的使用效率。

new 運算子

- 要在程式中動態配置一塊記憶體可使用 new 運算子, 其語法如下：

```
new 資料型別 (初始值);
```

1. **資料型別**：新配置空間的資料型別, 同時也決定配置空間的大小。例如指定為 int, 則會配置 4 個位元組的空間。
2. **初始值**：要設定的初始值, 可省略。

new 運算子

- new 運算子會傳回該新配置空間的記憶體位址, 因此我們可以把這個位址指定給指標變數, 讓指標指向新配置的空間：

```
int *ip = new int(100);     // ip 指向一個新的記憶體空間
                            // 且其初始值爲 100
```

delete 運算子

- 由於可供程式使用的記憶體空間是有限的資源, 為免不必要的浪費, 我們應在不再需要該空間時, 將它釋放。要釋放由 new 運算子配置的空間, 可使用 delete 運算子, 其語法就是在 delete 關鍵字後面接著指向該空間的指標即可, 例如：

```
int *ip = new int(100);           // 配置新的記憶體空間
...
delete ip;                        // 釋放所配置的記憶體空間
```

delete 運算子

- 上面的範例, 是初始化動態記憶體配置的寫法, 也就是在宣告指標的同時, 便配置記憶體空間。配置成功後, 我們就取得了一個可存放整數型別的記憶體空間, 並可透過 ip 指標來存取這個空間。空間使用完畢, 就 用 delete 將配置的記憶體釋放。

需要記憶體時，再配置

- 我們也可以在宣告指標時不做配置記憶體空間，而是等到程式需要使用之前再配置空間。例如：

```
int *num;
...
num = new int;    // 進行一些處理後再配置空間
...
delete int;
```

需要記憶體時, 再配置

- 不管是哪種配置方式, 記憶體空間使用完畢後, 一定要以 delete 將配置的記憶體空間釋放, 這是一個很重要的習慣, 可以讓記憶體的使用更具效率。

動態配置陣列空間

- 使用 new 運算子也可以配置陣列的空間, 只要在型別的後面使用 [ ] 並註明數量即可；而且以 delete 運算子釋放陣列空間時, 必須在 delete 後加上 [ ] 符號 (但不必註明數量)：

```
int *array_ptr = new int[100];  // 配置 100 個整數的記憶體空間
...
delete [ ] array_ptr;           // 釋放全部的空間
```

動態配置陣列空間

- 以上第 1 行的 new 敘述就是配置 100 個整數的記憶體空間, 並將空間的起始位址指定給 array_ptr, 就相當於建立一 個 100 個 int 的陣列, 並將陣列起始位址指定給指標。較特別的是, 在動態配置陣列空間時, 由於是在程式執行到該敘述時, 才配置所要的空間, 因此可用變數來指定陣列大小, 而不像以非動態的方式建立陣列只能使用常數或唯讀變數指定大小。

動態配置陣列空間

- 舉個例子來說，要寫一個程式來做不定個數值的計算。可用動態記憶體配置的方式，在程式執行時才決定要配置多少記憶體空間來使用，如以下範例所示：

程式 Ch07-15.cpp 計算不定數量數值的算術平均及幾何平均值

```
01 #include <iostream>
02 #include <cmath>
03 using namespace std;
04
```

動態配置陣列空間

```
05 int main()
06 {
07   int how_many;
08   cout << "請問要計算多少個數字的算術及幾和平均：";
09   cin >> how_many;
10
11   double *dptr  = new double[how_many]; //  配置指定數量的記憶體
12
13   for(int i=0;i<how_many;i++) {
14     cout << "請輸入第 " << (i+1) << " 個數值：";
15     cin >> *(dptr+i);
16   }
17
18   double sum = 0;
```

動態配置陣列空間

```
19     for(int  i=0;i<how_many;i++)                // 計算所有數值的總和
20        sum += *(dptr+i);
21     cout << "算數平均為：" << (sum / how_many) << endl;
22
23     sum = 1;
24     for(int  i=0;i<how_many;i++)                // 計算所有數值的乘積
25        sum *= *(dptr+i);
26     cout << "幾何平均為：" << pow(sum,1.0/how_many);
27
28     delete [] dptr;
29  }
```

動態配置陣列空間

執行結果

```
請問要計算多少個數字的總和：3
請輸入第　1　個數值：15
請輸入第　2　個數值：38
請輸入第　3　個數值：46
算數平均爲：33
幾何平均爲：29.7083
```

動態配置陣列空間

1. 第 11 行, 動態配置記憶體空間, 配置空間型別 為 double, 配置空間為 sizeof (double)*how_many, how_many 為第 9 行中, 由鍵盤輸入的數值個數。
2. 第 13~16 行, 以 for 迴圈請使用者依序輸入所有數值。
3. 第 19~21 行則計算並顯示所有數值的算數平均。
4. 第 24~26 行則計算並顯示所有數值的幾何平均。
5. 第 28 行以 "delete [ ] ..." 的語法釋放動態配置的陣列空間。

動態記憶體配置函式

- 由於 C++ 沿用了所有的 C 語言標準函式庫，因此也可使用宣告於 <cstdlib> 的動態記憶體管理函式：

```
(資料型別 *) malloc(sizeof(資料型別) * 個數);
free(指標變數);        // 釋放指標變數所指的動態記憶體空間
```

一定要指定初始值的參考型別

- 除了指標變數外, 還有一種特殊型別的變數稱為參考型別 (Reference Type) 變數, 參考型別變數的用處是讓我們為變數或常數建立**別名** (Alias), 然後便可用不同的識別字來參考到同一個資料或物件。參考型別的定義方式和指標有點像, 但需改用取址運算子 & 來定義或宣告, 例如：

```
int i = 5;
int &j = i;      // 參考型別變數 j, j 和 i 代表同一變數 (同一個櫃子)
```

一定要指定初始值的參考型別

- 在定義**參考型別**的變數時一定要設定初值，所以 "int &j;" 這樣的宣告是不合法的，因為如此編譯器就不能判斷它要作為誰的別名。同一個變數可有多個別名，這些別名的使用就和原來的變數完全一樣，而且我們再也不能更改它們之間的參考關係了：

```
int i, j;
int &a = i, &b = i;     // a, b 均和 i 同義
...
&a = j;                 // 錯誤: 無此用法
```

一定要指定初始值的參考型別

- 以下範例驗證參考型別的行為：

程式 Ch07-16.cpp 參考型別的變數行爲

```
01 #include <iostream>
02 using namespace std;
03
04 int main()
05 {
06     int age=18;
07     int &old=age;   // 定義參考型別變數
08
09     cout << "age的值：" << age << "\told的值：" << old << endl;
10     cout << endl << "請輸入 age 的新數值：";
11     cin >> age;;
```

一定要指定初始值的參考型別

```
12     cout << "age的值：" << age << "\told的值：" << old << endl;
13     cout << endl << "請輸入 old 的新數值：";
14     cin >> old;;
15     cout << "age的值：" << age << "\told的值：" << old << endl;
16 }
```

執行結果

```
age的值：18          old的值：18

請輸入 age 的新數值：99
age的值：99          old的值：99

請輸入 old 的新數值：110
age的值：110         old的值：110
```

一定要指定初始值的參考型別

- 這個程式很簡單, 只是將參考型別變數 old 設為整數變數 age 的別名, 之後則分別修改 age 及 old 的值, 並可由執行結果發現, 只要修改其中之一, 兩個變數的值都會一起更改, 因為它們都是代表同一個記憶體空間 (同一個置物櫃)。
- 參考型別主要是用在函式的參數傳遞和傳回值上。在其他情形下則應避免使用, 因為在一般情況下, 參考型別並不會帶來任何好處, 只是徒增困擾而已。

7-5 指標與參考在函式上的應用

- 前面介紹了許多有關指標與參考型別的基本用法, 但其實在一般的程式中會用到這些資料型別的情況, 有大半是因應呼叫函式時要以特殊的方式傳遞參數時使用。
- 那什麼是**以特殊的方式傳遞參數**呢？其實當我們呼叫函式時, 呼叫者 (例如 main() 函式) 傳遞參數給函式的方法共有 3 種：

指標與參考在函式上的應用

- **傳值呼叫** (Call by Value)：呼叫者會將參數的值傳給函式中的參數 (局部變數), 也就是只將數值複製過去。呼叫者呼叫函式時所用的參數, 和函式中的參數是不同的變數, 兩邊互不相干。當函式修改參數值時, 對呼叫者的變數並無影響, 這也是我們在 第 6 章所學的參數傳遞方式。

指標與參考在函式上的應用

- **傳址呼叫** (Call by Address)：函式宣告的參數型別為指標型別，所以呼叫時需以變數的位址為參數，因此函式中的指標參數會指向呼叫時所用的參數。
- **傳參考呼叫** (Call by Reference)：函式宣告的參數型別為參考型別，所以函式中的參數會變成呼叫者呼叫時所用參數的**別名**。

傳址呼叫

- 以一般的傳值呼叫，函式無法修改視野只在呼叫者中的變數。如果我們希望函式能直接存取呼叫者中的變數，就像存取自己視野中的變數一樣，則可用傳址呼叫的方式來設計函式，此時必須將函式的參數型別宣告成指標型別：

```
void func(int *i,int *j); // 宣告使用傳址呼叫的函式
...
int main()
{
```

傳址呼叫

```
    int  x,y;
    ...
    funct (&x,&y);                    //  呼叫函式時需傳遞變數位址
}
```

- 這樣一來, 變數 x 的位址會設給 func() 函式中的指標 i, y 變數的位址則會設給 j, 所以函式中可透這兩個指標, 直接存取到 main() 函式中的變數 x、y, 甚至直接修改其值。例如要設計一個將 2 個變數值對調的函式, 就可用傳址呼叫的方式來設計：

傳址呼叫

程式 Ch07-17.cpp 交換變數值

```
01 #include <iostream>
02 using namespace std;
03 void swap(int*,int*);
04
05 int main()
06 {
07     int a=5,b=10;
08     cout << "在 main()中..." << endl;
09     cout << "交換前 a = " << a << "  b = " << b << endl;
10     cout << "變數 a 的位址爲 " << &a << endl;
```

...

傳址呼叫

```
11     cout << "變數 b 的位址爲 " << &b << endl;
12
13     swap(&a,&b); // 呼叫函式, 並將變數 a,b 的位址當成參數
14     cout << "在 main()中..." << endl;
15     cout << "交換後 a = " << a << "  b = " << b << endl;
16 }
17
18 void swap(int *a,int *b)      // 將兩參數值對調的函式
19 {
20     int temp;                 // 暫存變數
21     temp = *a;
22     *a = *b;
23     *b = temp;
24     cout << "在 swap() 函式中..." << endl;
25     cout << "交換中 a = " << *a << "  b = " << *b << endl;
```

傳址呼叫

```
26     cout << "變數 a 的位址為 " << a << endl;
27     cout << "變數 b 的位址為 " << b << endl;
28 }
```

執行結果

```
在 main()中...
交換前 a = 5    b = 10
變數 a 的位址為 0012FF88
變數 b 的位址為 0012FF84
在 swap() 函式中...
交換中 a = 10    b = 5
變數 a 的位址為 0012FF88
變數 b 的位址為 0012FF84
在 main()中...
交換後 a = 10    b = 5
```

傳址呼叫

- 如執行結果所示, 在 main() 函式中將變數的位址傳遞到 swap() 函式中, 所以 swap() 函式中的 a、b 分別指向 main() 中的 a、b, 經過函式交換數值的處理, 也相當於將 main() 的局部變數 a、b 的數值交換過來。
- 除了上述需修改呼叫者數值的應用, 還有一種情況也會用到傳址呼叫, 就是需傳遞整個陣列的情況。由於陣列名稱也可看成是指標, 所以要傳遞陣列時, 可在函式中用陣列變數來接受參數, 也可以用指標變數來接受陣列的位址。

傳址呼叫

例如以下程式片段就是將函式參數型別宣告為陣列, 呼叫函式時也以陣列為參數的用法：

```
void func(int[]);      // 函式參數中不必指定陣列大小

int main()
{
  int i[10];
  func(i);             // 以陣列名稱（指標）爲參數呼叫函式
...
```

傳址呼叫

- 但要特別注意一點，由於傳遞陣列位址時，被呼叫的函式無法得知陣列的大小，為避免函式操作指標時超出陣列範圍，通常在設計這類函式時，會加上另一個參數來傳遞陣列大小。

傳參考呼叫

- 傳參考呼叫其實和傳址呼叫有些類似，都是讓函式能直接存取呼叫者中的變數。要設計傳參考呼叫的函式，須將函式的參數型別宣告成參考型別：

```
void func(int &i,int &j); // 宣告使用傳參考呼叫的函式
...
int main()
{
  int x,y;
  ...
  funct (x,y);                    // 呼叫函式時， 和傳值呼叫的方式相同
}
```

傳參考呼叫

- 傳參考呼叫的方式和傳值呼叫相同，只需以變數直接呼叫即可。但因為函式的參數為參考型別，所以此時參數傳遞的方式就彷彿執行以下敘述：

```
int &i = x;
int &j = y;
```

- 這樣一來，函式中的 I、j 就變成 main() 函式中變數 x、 y 的別名了，所以在函式中存取 i、j, 就等於存取到 main() 函式中的變數 x、y。例如前面的數值對調的函式，可改用傳參考呼叫的方式設計成如下的形式：

傳參考呼叫

程式 Ch07-18.cpp 以傳參考的方式交換變數值

```
01 #include <iostream>
02 using namespace std;
03 void swap(int&,int&);
04
05 int main()
06 {
07    int a=5,b=10;
08    cout << "在 main()中..." << endl;
09    cout << "交換前 a = " << a << "  b = " << b << endl;
10    cout << "變數 a 的位址為 " << &a << endl;
11    cout << "變數 b 的位址為 " << &b << endl;
12
13    swap(a,b); // 呼叫函式, 並將變數 a,b 當成參數
```

傳參考呼叫

```
14    cout << "\n在 main()中..." << endl;
15    cout << "交換後 a = " << a << "  b = " << b << endl;
16 }
17
18 void swap(int &i,int &j)      // 將兩參數值對調的函式
19 {
20    int temp;                  // 暫存變數
21    temp = i;
22    i = j;
23    j = temp;
24    cout << "\n在 swap() 函式中..." << endl;
25    cout << "交換中 i = " << i << "  j = " << j << endl;
26    cout << "變數 i 的位址爲 " << &i << endl;
27    cout << "變數 j 的位址爲 " << &j << endl;
28 }
```

傳參考呼叫

執行結果

```
在 main()中...
交換前 a = 5   b = 10
變數 a 的位址爲 0012FF70
變數 b 的位址爲 0012FF6C

在 swap() 函式中...
交換中 i = 10   j = 5
變數 i 的位址爲 0012FF70
變數 j 的位址爲 0012FF6C

在 main()中...
交換後 a = 10   b = 5
```

傳參考呼叫

- 如執行結果所示, 傳參考函式和傳址呼叫的函式有異曲同工之妙。那何時要採用傳參考函式？何時又要用傳址呼叫呢？我們可由參考型別和指標的性質來看：參考型別的變數一定要初始化, 也就是說若要用傳參考呼叫, 參數必須是已經存在的變數 (或物件)但指標可以是未指向任何空間的空指標(null pointer), 所以只要函式能接受, 則傳遞空指標給函式也無不可。

以指標或參考型別為傳回值

- 函式的傳回值型別也可以是指標或參考型別，以下先介紹傳回指標的用法。
  - 傳回指標
  - 傳回參考型別

傳回指標

- 由於 return 敘述只能傳回單一變數, 若想傳回字串或陣列 (多個變數), 就需傳回這個字串或陣列的起始位址, 讓呼叫者能以指標變數來取得傳回的位址值。要傳回一個位址, 我們要將函式宣告成指標型別, 例如：

```
int *func(int); // 傳回值爲整數指標
```

傳回指標

- 如此一來，傳回值便為位址。舉個例子，在標準函式庫中的字串處理函式，有很多都是直接以指標傳回處理過的字串，讓呼叫者可直接使用，這樣的設計有個好處，就是若程式中要馬上用到該傳回結果，可直接以函式表示之；不需再用一個變數去接受傳回值，然後使用該變數。舉例來說，要讓函式將某個字串中的小寫全部轉成大寫，然後將結果以另一個字串傳回，這時就可用傳回指標的方式處理：

傳回指標

程式 Ch07-19.cpp 字串大小寫轉換

```
01 #include <iostream>
02 #include <cstring>
03 #include <cctype>              // 使用 toupper() 需含括此檔
04 using namespace std;
05
06 char *toUpper(const char *);    // 宣告函式傳回值爲字元指標
07
08 int main(void)
09 {
10    cout << toUpper("happy Birthday");
```

傳回指標

```
11  }
12
13  char *toUpper(const char* ptr)              // 將字串所有小寫字母
14  {                                           // 轉成大寫的函式
15      unsigned len = strlen(ptr);
16      char *newStr = new char[len];           // 建立一新字串
17      for(unsigned i=0;i<len;i++)
18          *(newStr+i) = toupper(*(ptr+i));    // 將字元轉成大寫
19
20      return newStr;                 // 傳回轉換後的字串
21  }
```

執行結果

```
HAPPY BIRTHDAY
```

傳回指標

1. 第 3 行含括標準函式庫中的 <cctype>, 因為稍後會用到其中的 toupper() 函式。
2. 第 6 行將自訂函式的傳回值宣告為字元指標, 另外將參數型別宣告為 const 的意思, 是指函式中不會修改傳入的字串參數本身。
3. 第 10 行即呼叫函式並立即用 cout 輸出傳回的字串內容。

傳回指標

4. 第 13~21 行即為自訂的字串轉大寫函式, 第 15 行先用 strlen() 取得字串長度, 再用 new 配置新的字串空間, 用以存放轉成大寫的新字串。
5. 第 17 行的 for 迴圈逐字將 ptr 所指字串中的每個字元, 用標準函式庫的 toupper() 函式 (其功能就是傳回參數字元的大寫) 轉成大寫, 最後在第 20 行用 return 傳回轉換結果。

傳回參考型別

- 傳回參考型別的函式設計方式和傳回指標的方式類似，在此不多做介紹。但要提醒讀者，要傳回參考型別或指標時，要記得不要傳回局部變數的參考或指標，否則函式結束時局部變數的生命期也結束，呼叫者根本無法取得傳回值。

傳回參考型別

- 其實我們已使用這一類型的函式很多次而不自知, 也就是標準輸出的 cout 物件的 << 運算子。當我們將一串字串和變數用 << 運算子串在 cout 後面輸出時其實 "cout << 變數" 就是傳回 cout 物件的參考, 所以傳回值可繼續與下一 個 << 運算子參與運算。

7-6 綜合演練

- main() 函式的參數
- 以陣列為參數的應用：在陣列中搜尋資料
- 傳遞二維陣列的應用：字串排序

main() 函式的參數

- 到目前為止, 我們所寫的程式其 main() 函式都是沒有任何參數的, 但其實 main() 函式也可以加上參數。main() 函式的參數是讓程式可由作業系統取得**命令列參數**, 也就是在**命令提示字元**中執行程式時, 於程式名稱後加上的參數。例如執行 "ping xxx.yyy.zzz" 命令時, 其中 "xxx.yyy.zzz" 就是 ping 程式的參數。要讓 main() 函式取得這些參數, 需以如下形式定義函式：

main() 函式的參數

```
int main(int argc, char *argv[])
{
    ...
}
```

1.**argc**：是 argument count 的縮寫，表示 main() 取得的參數數量。

2.**argv**：由作業系統傳進來的命令參數，都以字串的形式存於此字元指標陣列中。

- main() 函式的參數名稱並非固定的，但一般都習慣使用 argc、argv 的名稱。

main() 函式的參數

- 在傳入參數時, 程式本身的路徑及檔名將會是陣列中的第 1 個元素。舉例子來說, 假設程式的檔名為 prog.exe, 執行 "prog 12 3.4 five" 時, main() 函式取得的參數將是：

```
argc = 4              // 連程式名稱在內, 共有 4 個參數
argv[0]="prog"        // 程式名稱 (若執行時有加上路徑, 字串中也會包含路徑)
argv[1]="12"          // 第 1 個參數
argv[2]="3.4"         // 第 2 個參數
argv[3]="five"        // 第 3 個參數
```

main() 函式的參數

- 請特別注意, argv[] 為 char* 型別, 因此即使命令列參數是數字, 也是以字串的形式儲存。若程式需要以數字的形式來處理該參數, 可用以下幾個宣告於 <cstdlib> 中的函式做轉換：

```
double atof(const char* str)  // 將參數 str 字串轉成 double 傳回
   int atoi(const char* str)  // 將參數 str 字串轉成 int 傳回
  long atol(const char* str)  // 將參數 str 字串轉成 long 傳回
```

- 例如我們想將第 6 章計算階乘的程式改成可讓使用者在命令列參數指定要計算階乘的數字, 可將程式改成：

main() 函式的參數

程式 Ch07-20.cpp 用有參數的 main() 函式取得命令列參數

```
01 #include <iostream>
02 #include <cstdlib>
03 using namespace std;
04
05 long double fact(int n) //  遞迴式函式
06 {
07     if(n == 1)                    // 在 n==1 時停止往下遞迴
08        return 1;                  // 傳回 1
09     else
10       return ( n * fact(n-1));     // 將參數減 1 再呼叫自己
11 }
12
13 int main(int argc, char *argv[])
```

main() 函式的參數

```
14 {
15     if (argc > 1)                  // 若有命令列參數
16         for(int i=1;i<argc;i++) {
17             int f = atoi(argv[i]);
18             cout << f << "! = " << fact(f) << endl;
19         }
20     else                           // 若沒有加參數就輸出使用說明
21         cout << "用法：\"程式名稱 數字\"" << endl;
22 }
```

執行結果

```
C:\F5700\Ch07>Ch07-22 5 7 9        ←在程式名稱後加上數字
5! = 120                           ←程式會算出 5、7、9 的階乘值
7! = 5040
9! = 362880
```

main() 函式的參數

1. 第 5 ~11 行為計算階乘的遞迴函式。
2. 第 13 行用有參數的形式定義 main() 函式。
3. 第 15 行先判斷除了程式名稱外是否還有參數, 若有就執行第 16 ~19 行的 for 迴圈；若無則執行第 21 行的敘述輸出簡短的使用說明訊息。
4. 第 16~19 行的 for 迴圈會從第 2 個元素開始, 逐一將 argv[] 陣列中的參數用 atoi() 轉成整數, 再用該數值呼叫 fact() 函式計算其階乘並輸出結果。

以陣列為參數的應用：在陣列中搜尋資料

- 以一維陣列為函式參數基本上和以基本資料型別為參數差不多，將參數宣告為陣列型別，即可在函式中透過參數存取到整個陣列的內容。

```
void func(int[]);

int main()
{
   int i[10];
   func(i);          // 以陣列爲參數呼叫函式
...
```

以陣列為參數的應用：在陣列中搜尋資料

- 不過除非我們寫的函式都只給自己用, 而且每次使用的陣列大小都相同, 否則就會面對在函式中無法得知陣列大小的問題。因為 C++ 在傳遞陣列時, 都是使用傳址呼叫, 也就是傳遞陣列指標, 從函式的觀點, 它是無法得知陣列有幾個元素。因此最好為函式加上另一個代表陣列大小的參數, 以方便函式處理。
- 以下就是利用函式在預存的營業額陣列中, 找出營業額最高與最低的月份。程式如下：

以陣列為參數的應用：在陣列中搜尋資料

程式 Ch07-21.cpp 列出營業額最高與最低的月份

```
01 #include <iostream>
02 using namespace std;
03 int findmax(int[],int);
04 int findmin(int[],int);
05
06 int main()
07 {
08     // 儲存各月份營業額的陣列
09     int income[]={156548, 152074, 176325, 120159, 94876, 163584,
10                   179541, 146587, 156472, 135587, 95443, 169994};
11
12     int i = findmax(income,12);
13     cout << "營業額最高的是 " << (i+1) << " 月："
```

以陣列為參數的應用：在陣列中搜尋資料

```
14            << income[i] << " 元" << endl;
15     i = findmin(income,12);
16     cout << "營業額最低的是 " << (i+1) << " 月：" 
17            << income[i] << " 元" << endl;
18 }
19
20 int findmax(int in[], int size)
21 {
22     int max = 0;
23     for (int i=1;i<size;i++)        // 找最大值的迴圈
24         if(in[max] < in[i])
25             max = i;
26
27     return max;
28 }
```

以陣列為參數的應用：在陣列中搜尋資料

```
29
30 int findmin(int in[], int size)
31 {
32     int min = 0;
33     for (int i=1;i<size;i++)          //  找最小值的迴圈
34        if(in[min] > in[i])
35           min = i;
36
37     return min;
38 }
```

執行結果

```
營業額最高的是 7 月：179541 元
營業額最低的是 5 月：94876 元
```

以陣列為參數的應用：在陣列中搜尋資料

1. 第 20~28 行為尋找陣列中最大值的 findmax() 函式, 其中用 for 迴圈逐一比對陣列中各元素, 以找出最大值。
2. 第 30~38 行為尋找陣列中最小值的 findmin() 函式, 同樣是用 for 迴圈逐一比對陣列中各元素, 以找出最小值。

傳遞二維陣列的應用：字串排序

- 字串的排序與數列的排序原理相同。首先，我們要宣告一個字串陣列，來儲存由鍵盤輸入的所有字串，然後再將這些字串依字母由小而大的順序重新排序。程式如下：

程式 Ch07-22.cpp 用函式將字串陣列排序

```
01  #include <iostream>
02  #include <cstring>
03  using namespace std;
04  #define LEN 80            // 定義字元陣列長度
05  void sort(char [][LEN], int);
```

傳遞二維陣列的應用：字串排序

```
06
07  int main()
08  {
09     char str[][LEN] = {"Taipei", "Taoyuan", "Hsinchu",
10                        "Miaoli", "Ilan", "Chiayi"};
11
12     sort(str,6);          // 將 str 排序, 共有 6 個字串
13
14     for (int i=0;i<6;i++)     // 輸出排序後的結果
15        cout << str[i] <<endl;
16  }
17
18  void sort (char str[][LEN], int count)
19  {
```

傳遞二維陣列的應用：字串排序

執行結果

```
Chiayi
Hsinchu
Ilan
Miaoli
Taipei
Taoyuan
```

```
20     char  temp[LEN];                         //  對調字串時的暫存陣列
21      for(int  i=0;i<count-1;i++)             //  用迴圈來比較字串的大小
22         for(int  j=i+1;j<count;j++)
23            if(strcmp(str[i],str[j])>0) {       //  比較字串
24               strcpy(temp,str[j]);             //
25               strcpy(str[j],str[i]);           //  對調字串
26               strcpy(str[i],temp);             //
27         }
28   }
```

傳遞二維陣列的應用：字串排序

1. 第 5 行宣告函式原型時, 二維陣列的參數必須指定最後一個維度的大小。
2. 第 12 呼叫 sort() 函式將陣列排序, 同時也傳入字串個數為參數。由於是傳址呼叫, 所以排序後, 陣列中的字串次序也會變動。
3. 第 20 行宣告的是暫存用的字串陣列, 當稍後比較字串要調換字串的次序時, 即需用到這個字串陣列。

傳遞二維陣列的應用：字串排序

4. 第 21 ~28 行以氣泡排序法對字串排序, 其中在比較及對調字串時, 都是使用 <ctsring> 中宣告的 strcmp()、strcpy() 函式。

- 這種程式寫法其實並不方便, 必須注意許多字元陣列的細節, 因 此 C++ 提供了一個更好用的字串類別 string 來取代以字元陣列儲存字串的方法。