第 6 章

陣列

陣列 (array) 是程式語言中操縱資料的利器，它可以將型別相同的資料整合起來，使得存取資料變得相當方便。若以倉庫來比喻變數所佔有的記憶空間，則倉庫的大小就由變數的型別控制，變數所儲存的資料就是擺放在倉庫內的物品， 陣列可以將之想像為一排「緊鄰」在一起而且擺放相同型別的倉庫儲存區，如果此區共有 n 間倉庫，則由於倉庫間緊緊相鄰的緣故，當知道了第一個倉庫的位址後，則其它倉庫的位址就可以輕易地算出，如此使得存取陣列內的資料都變得很容易。由此陣列的首筆資料的位址就變得相當重要，我們因此將之稱為陣列位址，陣列內的各筆資料則被稱為元素 (element)。

在前一章中，我們利用指標來儲存記憶空間位址，若將陣列位址存入指標中，則陣列內所有的元素位址就可以透過指標的基本運算來求得，本章除了介紹傳統陣列外，也將簡單介紹在標準樣板程式庫 (standard template library) 內的向量陣列 (`vector`)。

6.1 陣列儲存機制

陣列為一群緊鄰且相同資料型別的元素集合在一起，由於陣列內所有的資料型別皆為相同，所以整個陣列所佔有的空間就是元素總數乘上單一筆資料型別的大小。由於陣列的元素緊緊相鄰，知道了陣列的第一個元素的位址後，則其它筆元素的位址就可以輕易的算出，例如：若要得知陣列中第 n 個元素的位址，只須要將陣列位址加上 n-1 個單位的空間大小即可。對計算機而言，不管兩數是多少，運算一個加法的時間是固定的，因此對一個有 1000 個元素的陣列而言，取用陣列的第 2 個元素與第 1000 個元素所須要的時間是一樣的，並不會因離首位元素的遠近而有差別，這也是使用陣列的一個重要特點。

由於陣列所使用的空間是連續的，所以有時會發生一些奇怪現象，例如，當計算機的總記憶空間雖然很大，卻會發生作業系統抱怨執行某個程式所須要的記憶空間不足，其中一個主要原因常常是因系統無法在其記憶區域內，找到一塊完整的記憶空間足以存放陣列所須要的空間。在這種情況之下，作業系統常常會建議使用者將一些程

式關閉，藉以擴大連續空間的大小，使得程式得以執行。

陣列的元素是存放在連續的記憶空間內，通常作業系統會預先將陣列所須要的總記憶空間數量在執行之前就加以確定，藉以分配與保留適當的空間給程式執行。同時系統也會要求陣列的元素總數不能在程式執行時隨意更改，以免造成記憶空間管理上的困擾。然而常常陣列元素的確定個數不是在程式執行之前所能預知的。當設定的陣列元素太多，會造成記憶空間的浪費，使得其他的程式無法執行。若是元素總數太少，則會造成程式無法處理大型的問題。因此設計一種動態陣列，使得其能在程式執行過程中決定所須要的元素多寡，一直是許多程式設計員所需要的。C++ 特別在標準樣板程式庫 (standard template library) 提供了向量類別 (vector class) 來處理動態陣列。我們將在本章的最後一節做簡單的介紹，詳細要等到第十八章才會加以說明。

6.2 傳統陣列

傳統的 C++ 陣列可由以下幾個例子來定義，

```
int    foo[3]  ;   // foo 為 3 個整數的陣列
char   bar[4]  ;   // bar 為 4 個字元的陣列
float  abc[5]  ;   // abc 為 5 個單精度浮點數的陣列
```

陣列元素的個數寫在中括號之內。這裡的陣列長度不能為變數，須要為確定不變的數，因此可以為常數變數，例如：

```
const int SIZE = 10 ;
int   foo[SIZE] ;

int  i  ;
cin >> i ;
int  bar[i] ;       // 錯誤，陣列的長度不可為另一變數
```

對 C++ 而言，常數變數的值是固定的且在程式執行前就已確定，所以 C++ 可以在編譯過程中就分配確定數目的空間大小給陣列使用。若陣列有 n 個元素，則元素的編號依次為 foo[0]， foo[1]，...，foo[n-1]。陣列的下標範圍是由 0 到 n-1，所以若要將在 n 個元素的陣列分別存入 1 到 n 的整數，可以用以下方式：

```
int foo[100] ;
for ( int i = 0 ; i < 100 ; ++i ) foo[i] = i+1 ;
```

陣列的下標是由 0 開始，因此 foo[i] 是指 foo 陣列的第 i+1 個元素。若要讓兩個陣列的所有元素值都相等，無法直接使用類似數學的方式來設定，而是要使用迴圈方式老老實實的一個一個指定。

```
int i , foo[10] , bar[10] ;
for ( i = 0 ; i < 10 ; ++i ) foo[i] = i+1 ;

// 兩陣列的指定
```

```
bar = foo ;                                        // 錯誤
for ( i = 0 ; i < 10 ; ++i ) bar[i] = foo[i] ;  // 正確
```

使用陣列時要特別注意陣列的長度與所用的下標，若指定的下標超過陣列的有效下標範圍時，C++ 在編譯時並不會抱怨程式有誤，但卻會在程式執行時引發錯誤，造成程式中斷。

陣列初始值設定

當一個陣列被定義時，若元素初值並未設定，則可能是任何值。但 C++ 設計了幾種簡便的初始值設定方式，可以讓陣列在定義時就一併設定陣列元素的初始值，例如：

```
int  a[5] = { 1 , 2 , 3 , 4 , 5 } ;    // (A) a = 1 2 3 4 5
int  b[5] = { 1 , 2 , 3  } ;           // (B) b = 1 2 3 0 0
int  c[]  = { 1 , 2 , 3 , 4 , 5 } ;    // (C) c = 1 2 3 4 5
```

以上分別代表三種設定陣列初始值的方式，以 (A) 來說，陣列 a[5] 共分配了五個整數空間，而初始設定也為五個，因此 C++ 就由陣列的第一個元素 a[0] 開始依次存入至 a[4]，所有的元素初始設定資料都放於大括號內。 在 (B) 中，設定初始的數值數目少於定義的數目，這時 C++ 會自動將剩下的元素設定為零，由於這個性質，若要將整個陣列元素都設定為零，可以使用，

```
int d[100] = { 1 } ;    // 第 1 個元素是 1，其它皆為 0
int e[100] = { 0 } ;    // 100 個元素都是 0
```

最後一類 (C) 並沒有寫上陣列長度，這時 C++ 編譯器會自動以初始設定中的個數作為陣列長度，不須要程式設計員辛苦去算。請留意：雖然陣列的元素個數在編譯時須要確定，但個數僅是用來計算陣列所使用的空間使用總量，C++ 並沒有使用任何資料用來儲存陣列的元素個數，所以若要知道陣列元素個數，則使用者須要自行儲存。為方便起見，我們將陣列的長度定義為陣列的元素個數。

陣列長度的處理

陣列內的元素值往往是有規則的，陣列常常會透過迴圈來設定其內的元素值。迴圈的迭代次數通常是陣列的元素個數，由於陣列內並沒有儲存元素個數，有時我們可以將陣列長度存入某變數，以方便迴圈設定迭代次數，例如：

```
int  i , square[10] ;
for ( i = 0 ; i < 10 ; ++i ) square[i] = i * i ;
for ( i = 0 ; i < 10 ; ++i )
    cout << i*i << "=" << square[i] << endl ;
```

在這裡 square 陣列共有 10 個元素，而迴圈的迭代次數也是 10 次，如果以後要將陣列的個數由 10 個改成 20 個，則兩個迴圈中的 10 都要一起更改，否則會造成問題。

對大程式而言，調整陣列長度可能會因此更動相當多的地方，引起很大的麻煩。解決的方式有兩種，其一為使用常數變數設定陣列的長度，例如：

```
const int SIZE = 10 ;
int  i , square[SIZE] ;
for ( i = 0 ; i < SIZE ; ++i ) square[i] = i * i ;
for ( i = 0 ; i < SIZE ; ++i )
    cout << i*i << "=" << square[i] << endl ;
```

如此一來，當程式臨時要改變陣列長度，則只要修改常數變數一行即可，其餘皆可保持不變。第二種方法是利用 sizeof 函式計算出個數，

```
int  i , foo[] = { 1 , 3 , 5 , 7 } ;
int  size = sizeof(foo) / sizeof(int) ;
for ( i = 0 ; i < size ; ++i ) cout << foo[i] << endl ;
```

這裡 sizeof(foo) 為整數陣列 foo 所佔有的總位元組數， sizeof(int) 則為一個整數所佔用的位元組數目，兩者相除即為陣列長度。

前者的好處是方便，只須要多一個常數整數就可解決許多問題，但常數整數與陣列須看成一組。後者則是透過公式將陣列的長度算出，只須要陣列名稱與其型別即可，兩者各有其適用性。

打亂陣列元素

在統計學上的許多應用問題中，常常須要隨機的將陣列中的元素一一取出，而且元素不得重複取用，常見的應用如發牌，或者是樂透開獎等。對這類問題的處理，一般的處理方式是利用一個迴圈由前往後或者是由前往後，依次與尚未處理的元素對調其值即可，以下為一個簡單的程式範例：

打亂陣列元素 — randomize_array.cc

```
01  #include <iostream>
02  #include <cstdlib>   // 提供隨機函式 rand 與設定隨機函式
03                       // 初值的函式 srand
04  #include <ctime>     // 提供程式在執行時的時刻資料 time
05
06  using namespace std ;
07
08  int main() {
09
10      int  foo[] = { 2 , 4 , 6 , 8 , 10 };
11      int  i , j , tmp ;
12
13      // 以時間設定隨機函式初始值
14      srand( static_cast<unsigned>( time(NULL) ) ) ;
15
16      // 計算整數陣列的長度
17      int  size = sizeof(foo) / sizeof(int) ;
18
```

```
19      // 陣列元素由後往前依次與其前的另一個隨機位置對調數值
20      for ( i = size-1 ; i > 0 ; --i ) {
21          j      = rand() % (i+1) ;
22          if ( i == j ) continue ;  // 如果 i 與 j 兩位置相等則不對調
23          tmp    = foo[i] ;
24          foo[i] = foo[j] ;
25          foo[j] = tmp ;
26      }
27
28      for ( i = 0 ; i < size ; ++i )  cout << foo[i] << " " ;
29      cout << '\n' ;
30
31      return 0 ;
32
33  }
34
```

執行結果

```
01  8 4 10 2 6
02
```

在程式中，陣列長度是程式由初值設定自行判斷，我們可以使用公式將陣列長度算出。如果兩個欲對調元素的下標相同，自然無須浪費精力對調，因此我們使用 continue 直接讓程式跳入下一個迭代步驟。

程式也可以由前往後對調，不過稍微費力一些：

```
// 由前往後與剩餘的元素對調, 但最後一個不須再對調
for ( i = 0 ; i < size-1 ; ++i ) {
    j = rand() % (size-i) ;
    if ( j == 0 ) continue ;
    tmp      = foo[i] ;
    foo[i]   = foo[i+j] ;
    foo[i+j] = tmp ;
}
```

附帶提一點機率觀念，如果程式改成以下的方式：

```
for ( i = 0 ; i < size ; ++i ) {
    j = rand() % size ;
    if ( i == j ) continue ;
    tmp    = foo[i] ;
    foo[i] = foo[j] ;
    foo[j] = tmp ;
}
```

也就是在迴圈中每次仍與其它的元素對調，而不是與剩餘的元素對調，這種方式與樂透取球的步驟不太相同，會造成錯誤的機率分佈。

巴斯卡三角形：一

在數學的巴斯卡三角形 (Pascal's triangle) [註1]有許多用途，例如：可以用來代表二項式，即 $(x+y)^n$，展開後的每一項係數，

$$(x+y)^n = C_0^n x^n + C_1^n x^{n-1} y + C_2^n x^{n-2} y^2 + \cdots + C_{n-1}^n x y^{n-1} + C_n^n y^n$$

這裡的係數 $C_b^a = \frac{a!}{(a-b)!\,b!}$，可以以巴斯卡三角形來表示，不須要使用到複雜的階乘運算。

這裡為了計算每一列巴斯卡三角形的數值，程式用了兩個陣列來儲存緊鄰的兩列數值，由巴斯卡三角形得知，每一列陣列的頭尾兩個元素皆為 1。假設陣列 a 與 b 代表相鄰的上下兩陣列，則由巴斯卡三角形相加原理可以得知陣列的中間元素為 $b_i = a_{i-1} + a_i$，換成陣列即為 b[i] = a[i-1] + a[i]，每當進入下一輪迭代時，則讓下一列陣列取代上一列陣列。

```
                1
             1     1
          1     2     1
       1     3     3     1
    1     4     6     4     1
 1     5    10    10     5     1
```

巴斯卡三角形：一 — pascal1.cc

```
01   #include <iostream>
02   #include <iomanip>
03
04   using namespace std ;
05
06   int main() {
07
08       int  i , j , no ;
09
10       // a , b 分別為巴斯卡三角形的上一列與下一列陣列
11       int  a[20] , b[20] ;
12
13       cout << "> 輸入巴斯卡三角形的高度 : " ;
14       cin >> no ;
15
16       a[0] = 1 ;
17
18       for ( i = 0 ; i < no ; ++i ) {
19
20           // 列印巴斯卡第 i 列陣列
21           cout << setw(3*(no-i)) << " " ;
22           for ( j = 0 ; j <= i ; ++j ) cout << setw(6) << a[j] ;
23           cout << endl ;
24
25           // 計算下一列陣列值
```

[註1]事實上若以時間的先後，應稱為楊輝三角較為正確

```
26            b[0] = 1 ;
27            for ( j = 1 ; j <= i ; ++j ) b[j] = a[j-1] + a[j] ;
28            b[i+1] = 1 ;
29
30            // 重新設定上一列的陣列
31            for ( j = 0 ; j <= i+1 ; ++j ) a[j] = b[j] ;
32
33        }
34
35        return 0 ;
36
37    }
38
```

執行結果

```
01    > 輸入巴斯卡三角形的高度 : 6
02                            1
03                         1     1
04                      1     2     1
05                   1     3     3     1
06                1     4     6     4     1
07             1     5    10    10     5     1
08
```

為了讓巴斯卡三角形以三角形的方式呈現，在程式中特別使用 setw[66] 來控制輸出的寬度，以達到資料對齊的效果。此外，我們讓 a 與 b 兩陣列的長度都為 20，當然這對小高度的巴斯卡三角形有些浪費空間，但如果將之設定太小，則程式就無法印出整齊排列的大巴斯卡三角形，因此設定適當的陣列長度往往須要程式設計員自行斟酌，無法一概而論。

6.3 多維陣列

在許多科學計算中常常須要使用到多維陣列，例如，矩陣計算，C++ 也提供了多維陣列的使用方式，以下為一些多維陣列變數的定義方式：

```
int     a[3][4] ;          // a 為 3 乘 4 的整數陣列
double  b[2][3][4] ;       // b 為 2 乘 3 乘 4 的浮點數陣列
```

多維陣列的每一維的下標皆由 0 起算，與一維陣列並無差別，此外多維陣列記憶空間的分配方式與一維陣列無異，編譯器皆須要在編譯時就將多維陣列所須要的空間保留住，同時這些空間皆須為連續的記憶空間，在記憶空間內，多維陣列元素排列的順序是以第一維為最大的單位，最後一維為最小單位，若以二維陣列為例：

```
int    a[2][3] ; // a 為 2 列 3 行的整數矩陣
```

a[0][0]	a[0][1]	a[0][2]	a[1][0]	a[1][1]	a[1][2]

上圖的每一格代表一個整數空間，因此 a 陣列總共佔用了 24 (2×3×4) 個位元組的記憶空間。

又如 b[2][2][3] 為一個三維陣列，則其元素排列依次為 b[0][0][0]，b[0][0][1]，b[0][0][2]，...，一直到 b[1][1][2]。共 12 個元素，下圖的每一格代表一個單精確度浮點數。

```
float  b[2][2][3] ;
```

b[0][0][0]	b[0][0][1]	b[0][0][2]	b[0][1][0]	b[0][1][1]	b[0][1][2]
b[1][0][0]	b[1][0][1]	b[1][0][2]	b[1][1][0]	b[1][1][1]	b[1][1][2]

請留意：以上 b 陣列所有元素的排列是一長串連續的，並不是被分成兩列，也就是 b[1][0][0] 是緊接在 b[0][1][2] 元素之後。總之多維陣列元素的排列方式有點像數字時鐘 hh:mm:ss 的變化方式，也就先秒 (second)，再分 (**m**inute)，後時 (**h**our)。

多維陣列的總元素個數可以使用 sizeof 函式求得，例如：

```
int  foo[2][4][3] ;
cout << sizeof(foo) / sizeof(int) << endl ;   // 印出 24
```

多維陣列的初始值設定

多維陣列的初始值設定也是根據元素在其記憶空間的編排方式依次設定，初值設定的方式有兩種，第一種方式是將多維陣列當成一維陣列方式來設定初始值。

```
char a[2][3]    = { 'a' , 'b' , 'c' , 'd' , 'e' , 'f' } ;
int  b[2][3][2] = { 1 , 2 , 3 , 4 , 5 , 6 , 6 , 5 , 4 , 3 , 2 , 1 } ;
int  c[][2][3]  = { 1 , 2 , 3 , 4 , 5 , 6 , 7 } ;
```

多維陣列的元素會依元素順序依次指定，最後一種方式將第一維的下標省略，讓編譯器自行計算最小的下標，以 c 為例則為 2，但 c 陣列共有 12 個元素，而初值元素僅有 5 個，不夠的元素則補上 0，這種方式是利用到多維陣列元素的排列順序，由此也可以了解到不能使用 int c[2][2][] 或者是 int c[2][][3] 來定義多維陣列的原因。

第二種方式是由第一維起使用對應的大括號依次設定初始值。舉例來說：

```
int  a[2][3] = { { 1 } , { 2 , 3 } } ;
int  b[2][3] = { { 1 , 0 , 0 } , { 2 , 3 , 0 } } ;
int  c[3][3] = { { 1 } , { 1 , 1 } , { 1 , 1 , 1 } } ;
```

用矩陣表示則為

$$a = b = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 3 & 0 \end{bmatrix} \quad , \quad c = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix}$$

超過二維以上的陣列的初始設定也是類似，例如：

```
int  a[2][2][3] = { { { 1 } , { 2 , 3 } } ,
                    { { 4 , 0 , 5 } , { 6 , 7 , 8 } } } ;
```

以第一維的觀點來看，a 陣列共有兩個 2×3 二維陣列，分別為 { {1,0,0} , {2,3,0} }，{ {4,0,5} , {6,7,8} }，每個二維陣列的初值設定與之前一樣，因此 a 陣列的元素為 { 1 , 0 , 0 , 2 , 3 , 0 , 4 , 0 , 5 , 6 , 7 , 8 }。a[0][1][0] 之值為 2，a[1][0][0] 之值為 4， a[1][1][1] 之值為 7。同理：

```
int  b[][2][3] = { { { 1 , 2 , 3 } , { 4 , 5 , 6 } } ,
                   { { 7 , 8 , 9 } , { 2 , 4 , 6 } } ,
                   { { 1 , 3 , 7 } , { 9 , 0 , 2 } } } ;
```

則 b[0][1][2] 之值為 5，b[1][0][1] 之值為 8，b[2][1][1] 之值為 0，b 陣列的第一維會被設為 3。

矩陣相乘

矩陣相乘是線性代數計算中最基本的運算，兩個矩陣相乘如果用數學的方式來表示則為

$$[\mathbf{C}] = [\mathbf{A}][\mathbf{B}] \quad \text{或表示成} \quad C_{ij} = \sum_{k=1}^{p} A_{ik} B_{kj} \quad \forall\ i \in 1 \ldots m\ ,\ j \in 1 \ldots n$$

這裡 A 為 m×p，B 為 p×n，C 為 m×n 的矩陣，例如：以下的 A 為 2×3，B 為 3×2 的矩陣， C 則為 2×2 矩陣，

$$[\mathbf{A}] = \begin{bmatrix} 1 & 2 & 1 \\ 2 & 1 & 1 \end{bmatrix} \quad , \quad [\mathbf{B}] = \begin{bmatrix} 1 & 2 \\ 0 & 1 \\ 2 & 2 \end{bmatrix} \quad \Rightarrow \quad [\mathbf{C}] = \begin{bmatrix} 3 & 6 \\ 4 & 7 \end{bmatrix}$$

矩陣相乘　matrix_multiplication.cc

```
01   #include <iostream>
02
03   using namespace std ;
04
05   int main() {
```

```
06
07      int   i , j , k , sum ;
08      const int m = 2 , n = 3 ;
09
10      // 定義兩矩陣及乘積矩陣
11      int a[m][n] = { { 1 , 2 , 1 } , { 2 , 1 , 1 } } ;
12      int b[n][m] = { { 1 , 2 } , { 0 , 1 } , { 2 , 2 } } ;
13      int c[m][m] ;
14
15      // 相乘
16      for ( i = 0 ; i < m ; ++i ) {
17          for ( j = 0 ; j < m ; ++j ) {
18              sum = 0 ;
19              for ( k = 0 ; k < n ; ++k ) sum += a[i][k] * b[k][j] ;
20              c[i][j] = sum ;
21          }
22      }
23
24      // 列印 a 矩陣
25      for ( i = 0 ; i < m ; ++i ) {
26          for ( j = 0 ; j < n ; ++j ) cout << a[i][j] <<  " " ;
27          cout << endl ;
28      }
29
30      // 列印 b 矩陣
31      cout << endl ;
32      for ( i = 0 ; i < n ; ++i ) {
33          for ( j = 0 ; j < m ; ++j ) cout << b[i][j] <<  " " ;
34          cout << endl ;
35      }
36
37      // 列印矩陣乘積 c
38      cout << endl ;
39      for ( i = 0 ; i < m ; ++i ) {
40          for ( j = 0 ; j < m ; ++j ) cout << c[i][j] <<  " " ;
41          cout << endl ;
42      }
43
44      return 0 ;
45
46  }
47
```

執行結果

```
01  1 2 1
02  2 1 1
03
04  1 2
05  0 1
06  2 2
07
08  3 6
09  4 7
10
```

兩個矩陣的相乘雖然只是一個很簡單的程式，但其中有些關於數值計算的觀念倒是值得在此加以介紹，首先我們來看看如果 A 矩陣為 m×p 的矩陣，而 B 矩陣為 p×n 的矩陣，則由程式中我們可以知道計算兩個矩陣的乘法共須要執行的浮點數乘法次數為 m×p×n。如果 m，p，n 都為 10000。則兩個 10000×10000 的矩陣相乘所須要的總乘法次數為 10^{12}，這個計算次數須要耗費相當的電腦計算時間。此外一個 10000×10000 的矩陣，如果是使用 8 個位元組的雙精確度浮點數，則其所須要的記憶空間為 800 MB (也就是 8×10000×10000 個位元組)，這個記憶空間容量對早期的個人電腦還是無法處理的。所幸對許多計算問題而言，矩陣的型式多半為稀疏矩陣 (sparse matrix)，也就是矩陣內的大部份的元素都是零，尤其是越大的矩陣，其零元素所佔的比例更高，常常會超過 99%，這時若仍是使用全矩陣 (full matrix) 來儲存一堆零元素就變得是相當浪費空間，而且不切實際，因此就有些方法可以專門用來儲存非零元素，如此就可以節省大量的記憶空間，同時可以讓一些記憶空間不足的計算機有能力處理大型的矩陣計算。

巴斯卡三角形：二

這裡我們用二維陣列方式來計算巴斯卡三角形的值，若將巴斯卡三角形用二維陣列來表示，如下式

$$[\mathbf{P}] = \begin{bmatrix} 1 & & & & \\ 1 & 1 & & & \\ 1 & 2 & 1 & & \\ 1 & 3 & 3 & 1 & \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$$

陣列的元素若用數學方式，可以寫成

$$\texttt{P[i][j]} = \begin{cases} 1 & \texttt{i = j} \text{ 或者是 } \texttt{j = 0} \\ \texttt{P[i-1][j-1]+P[i-1][j]} & \text{其它} \end{cases}$$

這裡的矩陣，我們只使用了矩陣對角線以下的部份。

用矩陣方式計算巴斯卡三角形　pascal2.cc

```
01   #include <iostream>
02   #include <iomanip>
03
04   using namespace std ;
05
06   int main() {
07
08       int  i , j , no ;
```

```
09
10      int  pascal[20][20] ;
11
12      cout << "> 輸入巴斯卡三角形的高度 : " ;
13      cin >> no ;
14
15      for ( i = 0 ; i < no ; ++i ) {
16
17          cout << setw(3*(no-i)) << " " ;
18
19          for ( j = 0 ; j <= i ; ++j ) {
20
21              if ( j == 0 || i == j )
22                  pascal[i][j] = 1 ;
23              else
24                  pascal[i][j] = pascal[i-1][j-1] + pascal[i-1][j] ;
25
26              cout << setw(6) << pascal[i][j] ;
27          }
28
29          cout << endl ;
30
31      }
32
33      return 0 ;
34
35  }
36
```

執行結果

```
> 輸入巴斯卡三角形的高度 : 6
                     1
                  1     1
               1     2     1
            1     3     3     1
         1     4     6     4     1
      1     5    10    10     5     1

```

若與第 6.2 節中的巴斯卡三角形程式作比較，兩者輸出的結果一樣，但很明顯地，用二維陣列的方式來計算巴斯卡三角形的元素較為簡潔與便利。但也由於只使用了矩陣對角線以下的部份，整個矩陣上三角形部份的記憶空間被完全浪費掉。

6.4 向量陣列

在之前的傳統陣列中，陣列內元素總數須要在編譯時就能確定，如此編譯器才能分配足夠的記憶空間給陣列使用，但許多問題往往是在執行時才能確定須要使用多大的陣列。此外，傳統陣列在確定陣列長度後，就無法再予以加大或縮小，對許多問題而言

也是一項嚴重的限制[註2]。

以上兩種問題的關鍵在於我們須要一種陣列，可以在程式執行中動態的改變其長度。其中一個解決的方式即是利用上一章所介紹的動態空間管理機制來設計陣列，以往許多程式設計員常須要自行設計複雜的資料結構來模擬動態陣列。然而 90 年中期後，我們可以使用向量陣列 (vector) 來動態地改變陣列的長度，也就是說，向量陣列可以在程式執行中，依須要自動地增加或者是減少陣列元素的多寡。除應用遠比傳統陣列更多元外，在使用上，也可以如傳統陣列取用元素的方式來使用向量，在這裡我們僅介紹向量陣列與傳統陣列相似的部份，其它較複雜的功能則要留至第十八章再加以說明。

使用向量陣列資料型別，首先須要將 vector 標頭檔加入，即

```
#include <vector>
```

一般來說，我們稱向量變數為向量物件，在向量物件定義時，我們可以將向量陣列的元素數目寫在小括號內 (注意：非中括號)。而陣列元素的型別則寫在 vector<> 中的 <> 之內，例如：

```
vector<int>     foo(2)  ;   // foo 為整數向量陣列，有兩個元素
vector<char>    bar(5)  ;   // bar 為字元向量陣列，有五個元素
vector<float>   fbar(5) ;   // fbar 為有五個元素的向量陣列
```

向量陣列容許程式在執行時才決定向量陣列所須要的元素多寡，例如：

```
unsigned int  n ;
cin >> n ;
vector<int>  foo(n) ;   // 可以
int          bar[n] ;   // 錯誤 , 傳統陣列不適用
```

使用上，向量陣列與傳統陣列一樣，如

```
vector<int>  foo(10) ;

// foo 為 1 , 2 , ... , 10
for ( int i = 0 ; i < 10 ; ++i ) foo[i] = i+1 ;
```

若運算過程須要知道向量陣列的元素數目，可以在物件名稱之後，加上 .size() 即可，例如：

```
vector<int>  foo(10) ;
cout << "foo 向量物件有 " << foo.size() << "元素\n" ;
```

向量陣列物件的初始值設定與傳統陣列的設定大不相同，因此以下第一種方式的初始值設定是不對的，而可以使用第二種方式。

```
vector<int>  foo(3) = { 1 , 2 , 3 } ;    // 錯誤
vector<int>  bar(3,1) ;                  // bar 有 3 個整數，
```

[註2]有些編譯器現已經提供可調整的陣列

```
                                        // 初始值皆為 1
```

此外兩個同型別的陣列可以利用指定運算子來作整個向量陣列的指定，例如：

```
vector<char>  foo(5,'a') ;    // foo 有 5 個 'a' 字元
vector<char>  bar(3,'b') ;    // bar 有 3 個 'b' 字元
foo = bar ;                   // 指定後，foo 有 3 個 'b' 字元
```

這種整串陣列元素的指定方式並不適用於傳統陣列。

以上僅簡單的說明向量陣列的一些用法，其它的性質，例如定義多維向量陣列，向量陣列的運算等等，則留待以後再詳細說明。

包牌程式：迴圈篇

所謂的包牌程式是最基本的組合問題，即是由 m 個不同數字中任意選取 n 個數字所成的組合，其總數就是數學上所定義 C_n^m，

$$C_n^m = \frac{m!}{(m-n)!\,n!}$$

因此 $C_6^8 = \frac{8!}{2!\,6!} = 28$，這裡我們用簡單的程式來列印 C_3^m 的所有組合。也就是由 m 個不同號碼中任意選取 3 個不同的號碼的所有組合。基本的作法可以使用三層迴圈來將所有的組合全部找出。在這三層迴圈中，我們刻意地讓所有迴圈下標的起始值都不相同，以避免重複產生的數字。如果問題是要任選四個號碼，則程式中須要四層迴圈。腦筋靈活的人馬上會認為這樣的程式很不方便，如果當選取的號碼數目更動了，則程式也須要加以更動才行。舉例來說，假若問題是要在 100 個號碼中選取 50 個號碼，即 C_{50}^{100}，則程式須要用到 50 層迴圈，如此是很不實際的，所以使用迴圈的方式來處理這類問題有其侷限性，較方便的解決方式將留待第八章函式再加以說明。

包牌程式：迴圈篇　　combination_by_loop.cc

```
01   #include <iostream>
02   #include <vector>
03
04   using namespace std ;
05
06   //  簡單包牌程式 : 由 m 個數字中, 列印所有由三個數字的組合
07   int main() {
08
09       int  i , j , k , m , c = 1 ;
10
11       cout << "> 數字個數 : " ;
12       cin  >> m ;
13       vector<int>  number(m) ;
14
15       cout << "> 依次輸入 " << m << " 個數字 : " ;
16       for ( i = 0 ; i < m ; ++i ) cin >> number[i] ;
17
18       cout << endl ;
```

```
19
20        // 讓 i , j , k 永遠不會相等
21        for ( i = 0 ; i < m ; ++i ) {
22            for ( j = i + 1 ; j < m ; ++j ) {
23                for ( k = j + 1 ; k < m ; ++k ) {
24                    cout << c++ << " : " << number[i] << " "
25                         << number[j] << " " << number[k] << endl ;
26                }
27            }
28        }
29
30        return 0 ;
31
32    }
33
```

執行結果

```
01    > 數字個數 : 4
02    > 依次輸入 4 個數字 : 1 3 5 7
03
04    1 : 1 3 5
05    2 : 1 3 7
06    3 : 1 5 7
07    4 : 3 5 7
08
```

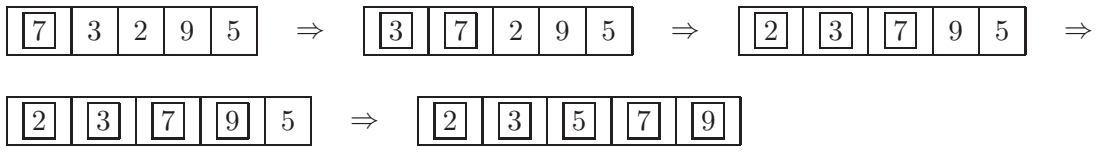
插入排序法

在許多應用問題，排序是非常重要的，例如：聯考考試的分數排序，各種統計資料的排序，如人口普查，健保資料等等。一般來說，排序有許多不同的方法，有的速度快，有的節省記憶空間，在這裡我們介紹一個簡單又不慢的排序法，即插入排序法 (insertion sort)。

插入排序法是透過數值對調方式將數列一步一步由小到大排列完成。舉例來說，假若有 n 個數字，插入排序法由第二個數字起往前比， 若數字比其前面且緊鄰的數字小時則與之對調，對調動作一直到數字比之前的數字較大為止，此時排序動作才改由下一個數字開始往前比較，同樣的動作一直會重複直到處理完最後一個數字，如此排序就完成了。

若以五個數字，7，3，2，9，5，為例，我們由第二個數字開始往前比，由於 3 比 7 小，所以兩數對調，接下來，考慮第三個數字 2，此時 2 比 7 小，須與 7 對調，接下來 2 又比 3 小，因此須再與 3 對調，如此一來，數字陣列現在成為 2，3，7，9，5，接下來考慮第四個數字 9，因 9 比 7 大， 所以就不對調，直接考慮第五個數字 5，5 會與 9 對調，再與 7 對調後結束排序。由下圖可知，當每完成一步時，其與

之前的元素就已經完成排序，因此若輸入的數列大致已排好次序，僅有少許數目未排好序，則運用此種方法來作最後的排序，其排序速度將會相當快捷。



插入排序法 insertion_sort.cc

```
01   #include <iostream>
02   #include <vector>
03
04   using namespace std ;
05
06   int main() {
07
08       int  i , j , no , tmp ;
09
10       cout << "> 輸入資料個數 : "  ;
11       cin >> no ;
12
13       // 定義一 no 筆資料的向量陣列
14       vector<int>  data(no) ;
15
16       cout << "> 請輸入 " << no << " 個數字 : " ;
17       for ( i = 0 ; i < data.size() ; ++i ) cin >> data[i]  ;
18
19       // 執行插入排序
20       for ( i = 1 ; i < data.size() ; ++i ) {
21
22           // 往前對調直到數字較之前的元素大為止
23           for ( j = i ; j > 0 ; --j ) {
24
25               if ( data[j] >= data[j-1] )
26                   break ;
27               else {
28                   // 若比之前的元素小則與之對調
29                   tmp       = data[j] ;
30                   data[j]   = data[j-1] ;
31                   data[j-1] = tmp ;
32               }
33
34           }
35
36       }
37
38       // 輸出排序後的結果
39       cout << "\n> 排序後結果 : " ;
40       for ( i = 0 ; i < data.size() ; ++i ) cout << data[i] << " " ;
41       cout << endl ;
42
```

```
43        return 0 ;
44
45    }
46
```

執行結果

```
01    > 輸入資料個數 : 10
02    > 請輸入 10 個數字 : 32 54 29 76 44 51 59 83 19 9
03
04    > 排序後結果 : 9 19 29 32 44 51 54 59 76 83
05
```

在此程式中，由於輸入的數字個數不確定，所以自然以向量陣列較為方便，同時也使用了 .size() 來表示向量陣列元素的個數。

6.5 陣列與指標

由於陣列的元素是儲存在一塊連續的記憶空間內，當知道了陣列第一個元素的位址後，陣列內所有元素的位址都可以很快的計算出來。因此若將陣列首位元素的位址存到某指標內，則陣列各個元素的位址就可以利用指標的位址運算得出，例如：

```
int   foo[3] = { 5 , 2 , 3 } ;
int  *p      = &foo[0] ;          // p 指標指向 foo 陣列的首位元素

// 利用指標位址運算列印 foo 所有的元素 : 5 2 3
for ( int i = 0 ; i < 3 ; ++i ) cout << *(p+i) << endl ;
```

在程式中，我們首先讓指標 p 指向陣列第一個元素的位址，即 &foo[0]，所以在列印時， p+i 所指的位址就是陣列 foo 第 i+1 個元素的位址，也因此 *(p+i) 也就相當於 foo[i]。如在上一章所述，下標運算子也可運用在指標上，也就是 *(p+i) 與 p[i] 同義，因此：

```
foo[i]   等同   *(p+i)   等同   p[i]
```

由此看來，一維陣列的名稱本身似乎就可看成一個指向陣列首位元素的指標，事實上，也是如此，以此為例，foo 等同 &foo[0]， foo+i 也就相當於 &foo[i]，同時 *(foo+i) 等同 foo[i]。如此以上的程式就可以改寫成：

```
int   foo[3] = { 5 , 2 , 3 } ;
int  *p      = foo ;              // p 指標指向 foo 陣列的首位元素

// 利用指標下標運算子列印 foo 所有的元素 : 5 2 3
for ( int i = 0 ; i < 3 ; ++i ) cout << p[i] << endl ;
```

請留意：陣列名稱 foo 永遠指向陣列的首位元素，不能在程式中被改變，否則陣列的元素就可能無法存取了。但是指標卻可以指向陣列內任何元素的位址，例如：

```
int   foo[] = { 5 , 2 , 3 , 4 , 1 } ;
foo += 1 ;                              // 錯誤，foo 不能被更動
int  *p      = foo + 4 ;                // 指標 p 指向 foo[4] 的位址
int  *q      = foo + 1 ;                // 指標 q 指向 foo[1] 的位址
cout << p - q << endl ;                 // 印出 3

int   bar[] = { 10 , 20 , 30 } ;
foo = bar ;                             // 錯誤，foo 不能被更動
```

以上最後一個式子是想讓陣列 foo 以指標的型式指向 bar 陣列的首位元素位址，如果這樣的處理方式可以被允許，則原來 foo 陣列內所有的元素就無從被程式中的其它敘述使用，就好像在圖書館內將某本書的封面去除一樣，事實上，這也是陣列與指標的最大不同之處，由此我們可以將陣列名稱視為一種不能改變指向位址的常數指標 (constant pointer)。

此種方式也可以使用於多維陣列，以下是以指標的觀點來看多維陣列資料位址與元素：

一維陣列

```
int foo[N] ;
int *p = &foo[0] ; // &foo[0] 為第一個元素位址
```

位址 `&foo[i]` ⟺ `p+i`

元素 `foo[i]` ⟺ `*(p+i)` ⟺ `p[i]`

二維陣列

```
int foo[R][C] ;
int *p = &foo[0][0] ; // &foo[0][0] 為第一個元素位址
```

位址 `&foo[i][j]` ⟺ `p+i*C+j`

元素 `foo[i][j]` ⟺ `*(p+i*C+j)` ⟺ `p[i*C+j]`

三維陣列

```
int foo[A][B][C] ;
int *p = &foo[0][0][0] ; // &foo[0][0][0] 為第一個元素位址
```

位址 `&foo[i][j][k]` ⟺ `p+i*B*C+j*C+k`

元素 `foo[i][j][k]` ⟺ `*(p+i*B*C+j*C+k)` ⟺ `p[i*B*C+j*C+k]`

以上的指標與其指向元素的關係式是建構在陣列的空間分配是連續的基礎上，且連續性並不會因陣列維度增加而破壞。同時，由於這裡的指標是以一個資料型別為基本單位，因此不管陣列的維度如何，我們仍是可以透過簡單的位址運算就可找出陣列內對應的元素位址。

以下的程式特別利用指標的方式存取矩陣元素，設定矩陣元素儲存值為其下標和，即 `foo[i][j] = i + j`，

```
int   i , j ;
const int R = 2 , C = 3 ;
int  foo[R][C] ;                    // 陣列有 R 列 C 行 個整數
int  *p = &foo[0][0] ;              // 指標 p 指向 foo 陣列首位元素
for ( i = 0 ; i < R ; ++i ) {
    for ( j = 0 ; j < C ; ++j ) {
        *(p+i*C+j) =  i + j  ;      // 相當於 foo[i][j] = i + j
    }
}
```

若將結果以表來表示則為

元素	foo[0][0]	foo[0][1]	foo[0][2]	foo[1][0]	foo[1][1]	foo[1][2]
資料	0	1	2	1	2	3
指標	p	p+1	p+2	p+3	p+4	p+5

須特別注意一點，對一維陣列而言，`foo` 與 `&foo[0]` 同樣都是代表陣列首位元素的位址，因此 `*foo` 與 `foo[0]` 都代表第一個元素，也就是一維陣列名稱本身就是相當於永遠指向陣列首位元素的指標，但這個性質對多維陣列而言並不成立，原因將隨即說明。

多維陣列與指標

對多維陣列而言，理解上比較麻煩，若以三列四行 (3×4) 的二維整數陣列 `int foo[3][4]` 為例，所謂的元素 `foo[1][2]` 是指由第 2 列起始位址後起算的第 3 個整數元素，而 `foo[2][3]` 則為第 3 列起始位址起算的第 4 個整數元素。由此看來，二維陣列的兩個下標的作用不盡相同，第一個下標告知某一列的「起始位址」，而第二個下標的作用則如同一維陣列的下標一般，可以得到元素的資料。若先將第二個下標遮住，則之前的 `foo[1]` 或者是 `foo[2]` 分別代表第二列與第三列的起始位址，而非各列的起始元素，由此可以推知 `foo[i]` 代表第 `i+1` 列的起始位址。

由前一章的指標運算中可知，若 `p` 為一指標，則 `p[i]` 與 `*(p+i)` 同義，都代表指標所指向的資料。同樣的，在二維陣列中，C++ 也讓 `foo[i]` 的作用等同 `*(foo+i)`，都代表二維陣列第 `i+1` 列的起始位址。以這樣的角度來理解，元素 `foo[2][3]` 也可寫成 `(*(foo+2))[3]`。同樣的方式，將之擴充到第二個下標，也可以進一步寫成 `*(*(foo+2)+3)`，因此在以下的程式中，每一行所列印的資料都是一樣：

```
int  foo[2][3] = { 1 , 2 , 3 , 4 , 5 , 6 } ;
int  i , j ;
for ( i = 0 ; i < 2 ; ++i ) {
    for ( j = 0 ; j < 3 ; ++j ) {
        cout <<  foo[i][j]      << ' '
             <<  (*(foo+i))[j]  << ' '
             <<  *(*(foo+i)+j)  << endl ;
    }
}
```

印出

```
1 1 1
2 2 2
3 3 3
4 4 4
5 5 5
6 6 6
```

當然腦筋正常的人是不會利用後兩種方式來存取陣列內的元素，但由此卻可以留意到 *(foo+i) 所代表的位址，是以列為單位，也就是 i 每增加一，其所指向的位址往前移動一整列元素單位，若每列有 r 個元素，則就是移動了 r 個單位的記憶空間。這與前一章所介紹的指標，每次移動一個型別單位不同，因此以下指標的指定方式是不太對勁：

```
// foo : 2 列，每列有 3 個元素
int  foo[2][3] = { 1 , 2 , 3 , 4 , 5 , 6 } ;
int  *p = &foo[0] ;                              // 錯誤
```

錯誤的原因在於指標 p 是以 1 個整數為移動單位，foo 是以 3 個整數為移動單位。為了讓指標也能以多個元素為一個單位的方式移動，C++ 使用了陣列式指標，也就是此指標的位址運算是以陣列長度為一移動單位，例如：

```
int  foo[2][3] = { 1 , 2 , 3 , 4 , 5 , 6 } ;

int  (*p)[3] = &foo[0] ;   // (1) p 為陣列式指標，指向 foo 第一列
                           //     位址，同時以每 3 個整數移動單位

int  (*q)[3] = foo ;       // (2) q 為陣列式指標，效果同上

int  (*r)[3] = foo+1 ;     // (3) r 為陣列式指標，指向 foo 第二列
                           //     位址，
```

在這裡的第 (1) 種指定型式，p 為陣列式指標，指向 foo 第一列位址，注意：foo[0] 之前須要有位址運算子，不可省略，其道理與下面的第二個敘述中，整數 a 有位址運算子是一樣的。

```
int  a = 3 ;
```

```
int *b = &a ;  // 指標 b 指向 a
```

第 (1) 種型式的寫法有些臃腫，為方便起見， C++ 讓二維陣列的名稱 foo 代表 &foo[0]，如第 (2) 種型式，如此一來，第 (3) 種型式的意義就不難理解了。

當陣列式指標 p 指向 foo 陣列第一列時，則 p+i 即指向陣列第 i+1 列， p+i 與 p 之間相隔了 i 列的元素個數。如果要用陣列式指標 p 指向 i 列 j 行的元素，當然不能使用 p+i+j，因為其是代表第 i+j+1 列的起始位址。原因在 p 為一陣列式指標，其移動的方式是以列為單位，若要讓 p 以單一型別為移動單位時，則須要使用 *(p+i)，如此 *(p+i)+j 就相當於 &foo[i][j]，*(*(p+i)+j) 就相當於 foo[i][j]。若使用下標運算子於陣列式指標 p，則 *(p+i)+j 相當於 p[i]+j，*(*(p+i)+j) 也相當於 p[i][j]，因此若：

```
const int R = 2 , C = 3 ;
int       foo[R][C] ;
int       (*p)[C] = foo ;
```

則

```
      *(p+i)+j     相當於    p[i]+j     相當於    &foo[i][j]
    *(*(p+i)+j)    相當於    p[i][j]    相當於     foo[i][j]
```

以下的程式是透過陣列式指標間接地來列印 foo 陣列：

```
int  i , j ;
int  foo[2][3] = {1,2,3,4,5,6} ;
int  (*p)[3] = foo ;

// 以列為單位  列印每一列的元素
for ( i = 0 ; i < 2 ; ++i ) {
    for ( j = 0 ; j < 3 ; ++j ) cout << p[i][j] << ' ' ;
    cout << endl ;
}
```

輸出：

```
1 2 3
4 5 6
```

三維以上的陣列，也是依尋一樣的道理。以下分別將一，二，三維陣列的元素位址與資料值用陣列式指標來表示。

一維陣列

```
int foo[N] ;
int *p = foo ; // p 為一般指標
```

```
位址    &foo[i]    ⟺      p+i
元素     foo[i]    ⟺    *(p+i)    ⟺     p[i]
```

二維陣列

```
int foo[R][C] ;
int (*p)[C] = foo ; // p 為陣列式指標

位址   &foo[i][j]   ⟺    *(p+i)+j    ⟺     p[i]+j
元素    foo[i][j]   ⟺  *(*(p+i)+j)  ⟺   *(p[i]+j)   ⟺    p[i][j]
```

三維陣列

```
int foo[A][B][C] ;
int (*p)[B][C] = foo ; // p 為陣列式指標

位址   &foo[i][j][k] ⟺    *(*(p+i)+j)+k    ⟺    p[i][j]+k
元素   foo[i][j][k]  ⟺  *(*(*(p+i)+j)+k)  ⟺  *(p[i][j]+k)
                                           ⟺   p[i][j][k]
```

以下是透過一陣列式指標取出三維陣列元素，將之印出：

```
int  foo[2][3][2] = {1,2,3,4,5,6,7,8,9,0,1,2} ;   // 共有 12 個元素
int  (*p)[3][2] = foo ;
int  i , j , k ;

for ( i = 0 ; i < 2 ; ++i ) {
    for ( j = 0 ; j < 3 ; ++j ) {
        for ( k = 0 ; k < 2 ; ++k ) cout << p[i][j][k] << ' ' ;
    }
    cout << endl ;
}
```

輸出：

```
1 2 3 4 5 6
7 8 9 0 1 2
```

正常的情況下，我們是不須透過指標來存取陣列元素。其真正的使用時機是將陣列以參數的型式傳入函式內，有關這方面的內容將在第八章時介紹。此外，以上使用指標存取陣列元素的方式，僅適用於傳統陣列，以上所有的討論都不適用於向量陣列。

指標陣列

陣列內的元素除了可以為各種基本資料型別外，也可以是專門儲存位址的指標。指標

陣列 (pointer array) 是指陣列內的每一個元素都是指標，這些指標可以用來儲存資料的位址，例如：

```
int   a[9] ;   // 9 個整數元素的陣列
int  *p[9] ;   // 陣列包含 9 個整數指標的元素

// 將陣列 a 的每個元素的位址存到 p 指標陣列內
for ( i = 0 ; i < 9 ; ++i )  p[i] = &a[i] ;

// 透過指標陣列設定 a 陣列資料，即 0 1 4 9 16 ... 64
for ( i = 0 ; i < 9 ; ++i ) *p[i] = i * i ;
```

由於下標運算子 ([]) 的處理順序較參照運算子 (*) 為先[51]，因此 *p[i] 相當於 *(p[i])，也就是 p 陣列的第 i+1 個指標所指到記憶空間內的整數。

如果是使用向量陣列則須將指標的型別放到 <> 之內，

```
// 向量陣列包含 9 個整數指標的元素
vector<int*>  q(9) ;

// 向系統要取一動態整數記憶空間, 並存入立方數
for ( i = 0 ; i < 9 ; ++i ) q[i] = new int(i*i*i) ;

for ( i = 0 ; i < 9 ; ++i )
    cout << i << " 立方 = " << *q[i] << endl ;

// 將動態記憶空間退回給系統重新使用
for ( i = 0 ; i < 9 ; ++i ) delete q[i] ;
```

本例與上例的主要差別在於陣列內的每個指標都指向一動態記憶空間，因此當 q 陣列要離開其存在領域之前，這些動態記憶空間的位址須要設法去除，否則會造成記憶空間流失[101]的問題發生，以此為例，我們使用了 delete 將動態空間退還給作業系統重新使用。

指標陣列內所儲存的資料是另一個資料型別的位址，對現在主流的中央處理單元而言，所有資料型別的地址都是同樣以四個位元組方式來記錄，並不會因為資料型別的複雜度不同而有所差別。指標陣列在以後的抽象類別[406] (abstract class) 中會常常使用到，我們將會在第十三章介紹類別間關係時再加以解說。

6.6 應用範例

以下我們來看看兩個使用陣列的實用程式範例：

點矩陣文字

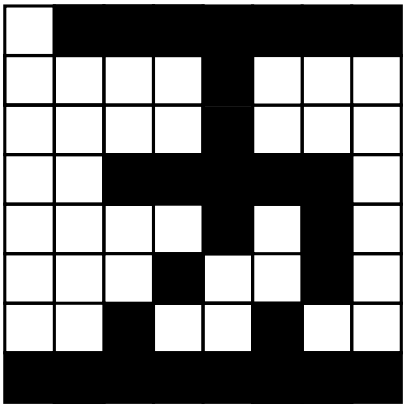
計算機輸出的文字許多都是用點矩陣來表示，所謂的點矩陣文字有點像將文字寫在方格紙上，若文字的筆劃出現在小格子內，則表示須要印出一小點，若沒有出現在格子

內，則不須要印出任何東西。現在用八個二進位數來表示八個格點，若是格子有塗滿則表示 1，若沒有塗滿則表示 0，舉例來說，數字

$$139 = 1 \times 2^7 + 0 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$$

用圖表示為

若每個文字使用 8×8 的格子來表示，則下圖的國字「五」中每一排的數字分別為



```
0  1  1  1  1  1  1  1  →  127
0  0  0  0  1  0  0  0  →    8
0  0  0  0  1  0  0  0  →    8
0  0  1  1  1  1  1  0  →   62
0  0  0  0  1  0  1  0  →   10
0  0  0  1  0  0  1  0  →   18
0  0  1  0  0  1  0  0  →   36
1  1  1  1  1  1  1  1  →  255
```

所以每一個數字可以只用一個無號字元 (unsigned char) 來儲存，則一個 8×8 點矩陣的文字，只要 8 個字元來儲存即可。下面的程式，使用者輸入 8 個介於 0 到 255 的數字，然後將其有值的點以星號印出，無值的點以空格印出。也就是當我們輸入上圖的 8 個數字後，螢幕會印出

```
* * * * * * *
      *
      *
  * * * * *
      *   *
    *     *
  *       *
* * * * * * * *
```

在此程式中，為輸入方便起見，我們將輸入的值存入整數陣列中。但在實際處理上，所有的文字資料是由二進位檔案讀入，此時當然只要將其數值存入無號的字元陣列即可。這裡的無號字元被當成一個位元組的正整數使用，藉以節省計算機的記憶空間，對小整數而言，這種方式可以節省三個位元組的空間大小。

點矩陣文字　　bitmap.cc

```
01   #include <iostream>
```

```
02  #include <vector>
03
04  using namespace std ;
05
06  // 點矩陣文字
07  int  main() {
08
09      int  i , j  , n  ;
10      cout << "> 輸入中文字所對應的 8 個點矩數字 : " ;
11
12      vector<int>    no(8) ;
13      vector<bool>   bitmap(8) ;
14
15      // 儲存文字點矩陣資料
16      for ( i = 0 ; i < 8 ; ++i ) cin >> no[i] ;
17
18      cout << endl ;
19
20      // 迴圈重複每一行
21      for ( i = 0 ; i < 8 ; ++i ) {
22
23          n = no[i] ;
24
25          // 將每一列的各個格子值存入 bitmap 矩陣中
26          for ( j = 7 ; j >= 0 ; --j ) {
27              bitmap[j] = n % 2 ;
28              n /= 2 ;
29          }
30
31          // 列印每一列
32          for ( j = 0 ; j < 8 ; ++j )
33              cout << ( bitmap[j] ? " *" : "  " ) ;
34
35          cout << endl ;
36
37      }
38
39      return 0 ;
40
41  }
42
```

執行結果

```
01  > 輸入中文字所對應的 8 個點矩數字 : 127 8 8 62 10 18 36 255
02
03     * * * * * * *
04           *
05           *
06       * * * * *
07           *   *
08         *     *
09       *     *
10   * * * * * * * *
11
```

矩陣相乘：利用指標

在這裡我們特別用指標來作矩陣相乘，假設定義兩個矩陣 A 與 B 分別為

$$[\mathbf{A}] = \begin{bmatrix} 1 & 2 & 1 \\ 2 & 1 & 1 \end{bmatrix} \quad , \quad [\mathbf{B}] = \begin{bmatrix} 2 & 3 \\ 2 & 1 \\ 0 & 2 \end{bmatrix}$$

而 C 矩陣為 A，B 兩矩陣的乘積，即

$$[\mathbf{C}] = [\mathbf{A}][\mathbf{B}] = \begin{bmatrix} 1 & 2 & 1 \\ 2 & 1 & 1 \end{bmatrix} \begin{bmatrix} 2 & 3 \\ 2 & 1 \\ 0 & 2 \end{bmatrix} = \begin{bmatrix} 6 & 7 \\ 6 & 9 \end{bmatrix}$$

在程式中我們使用一般的指標來存取二維陣列內的元素，這種處理方式當然是相當笨拙。在正常的情況下，我們寧可直接使用陣列的下標來存取其內的元素。本例僅是用來示範如何利用指標來操縱陣列元素的使用方式。

矩陣相乘：利用指標　　matrix_multiplication_by_ptr.cc

```
01   #include <iostream>
02   #include <iomanip>
03
04   using namespace std ;
05
06   int main() {
07
08       int i , j , k , sum ;
09
10       // 矩陣大小
11       const int ROW_A = 2 , COL_A = 3 ;
12       const int ROW_B = 3 , COL_B = 2 ;
13
14       // 定義兩矩陣及其乘積矩陣
15       int a[ROW_A][COL_A] = { { 1 , 2 , 1 } , { 2 , 1 , 1 } } ;
16       int b[ROW_B][COL_B] = { { 2 , 3 } , { 2 , 1 } , { 0 , 2 } } ;
17       int c[ROW_A][COL_B] ;
18
19       // 定義三個指標分別指到矩陣首位元素
20       int  *pa , *pb , *pc ;
21
22       pa = &a[0][0] ;
23       pb = &b[0][0] ;
24       pc = &c[0][0] ;
25
26       // 計算乘積結果
27       for ( i = 0 ; i < ROW_A ; ++i ) {
```

```
28          for ( j = 0 ; j < COL_B ; ++j ) {
29              sum = 0 ;
30              for ( k = 0 ; k < COL_A ; ++k ) {
31                  sum += *(pa+i*COL_A+k) * *(pb+k*COL_B+j) ;
32              }
33              *(pc+i*COL_B+j) = sum ;
34          }
35      }
36
37      // 列印矩陣 a
38      for ( i = 0 ; i < ROW_A ; ++i ) {
39          for ( j = 0 ; j < COL_A ; ++j ) {
40              cout << *(pa+i*COL_A+j) <<  " " ;
41          }
42          cout << endl ;
43      }
44
45      cout << '\n' ;
46
47      // 列印矩陣 b
48      for ( i = 0 ; i < ROW_B ; ++i ) {
49          for ( j = 0 ; j < COL_B ; ++j ) {
50              cout << *(pb+i*COL_B+j) <<  " " ;
51          }
52          cout << endl ;
53      }
54
55      cout << '\n' ;
56
57      // 列印矩陣乘積 c
58      for ( i = 0 ; i < ROW_A ; ++i ) {
59          for ( j = 0 ; j < COL_B ; ++j ) {
60              cout << *(pc+i*COL_B+j) <<  " " ;
61          }
62          cout << endl ;
63      }
64
65      return 0 ;
66
67  }
68
```

執行結果

```
01  1 2 1
02  2 1 1
03
04  2 3
05  2 1
06  0 2
07
08  6 7
09  6 9
10
```

6.7 結語

C++ 陣列的下標由零開始，對許多人而言是相當不習慣的，尤其所謂的自然數是由 1 開始，筆者經過一段時間的忍受才慢慢習慣。使用陣列對一般理工學界的人是再自然不過了，幾乎大部份的數值演算問題最後都須要使用迴圈與陣列，早期的程式設計就是將數值方法中所提供的程式執行代碼[76] (pseudo code) 原原本本的轉成程式。讀者如果有興趣，可以到圖書館借閱數值分析的書，你將會發現你已經有能力撰寫大部份的數值演算程式了。

一般初學者很容易使用超過三維以上的陣列，如果是這樣，將會使得程式很沒有效率。原因在 C++ 會花費相當多的時間計算元素在記憶空間中的確切位址，同時大部份的情況下，會浪費許多記憶空間。避免的方法則是在檢討演算的程序，重新設計程式。即使如此，對大部份的科學計算問題，最後都免不了要作矩陣計算，一個 5000×5000 的矩陣在實際的應用問題是屬於中小型的數值問題，假設每個元素都以雙精確度的浮點數 (佔用八個位元組) 來儲存資料，簡單的計算可知，矩陣所須佔要的空間大小為 200 MB (5000×5000×8) 的記憶空間。這在八零年代以前，對一般的個人電腦而言還是一個天文數字般的記憶空間容量，由於記憶空間不足，造成程式無法執行。所幸對大部份矩陣而言，都是所謂的稀疏矩陣 (sparse matrix)，也就是矩陣中的大多數的元素都為零，為了能執行程式，許多人想出了各種聰明的演算法來撰寫程式，有興趣的讀者可以在資料結構的書籍中找到答案。

6.8 練習題

1. 對使用 32 個位元來記錄位址的中央處理單元而言，假設要使用方陣 (square matrix) 來儲存元素，如果所儲存的元素型別分別為整數與雙精確度浮點數，請問在理論上，此中央處理單元所能處理的方陣大小分別為何 ？

2. 某生在其程式中使用三維的浮點數陣列 `double foo[500][500][500]` 來計算其數值問題，若其計算機僅有 128 MB 的記憶空間，請問夠用嗎 ？

3. 利用雙層指標[103]所取得的二維動態空間與直接使用二維陣列所產生的空間，請問在空間配置的連續性上有何差異 ？

4. 解釋 `int *foo[10]` 與 `int (*foo)[10]` 有何不同 ？

5. 若 `int foo[4] = { 2 , 3 , 5 , 9 }`，請問 `*foo+2` 與 `*(foo+2)` 有何差別 ？

6. 說明以下程式碼錯誤的原因 ？

```
int  a[3][2] ;
int **p = a ;        // 錯誤
```

7. 以下的 a 陣列內依次存了哪些數？

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

8. 請寫一個程式輸入方陣的大小 n，印出 n×n 的數字螺旋圖，例如，以下為 n = 5 的輸出：

```
1  2  3  4  5
6  7  8  9  6
5  4  5  0  7
4  3  2  1  8
3  2  1  0  9
```

9. 兩個四位數字作比對，每個數字的組合數字不能重複，比對時，若兩數字的同位置的數字相同，則得一 A，若只有數字相同但卻不同位置，則得一 B。請寫一程式，輸入兩個數字，印出下列右邊的比對結果。

```
3287      1238      --->    1A2B
5790      6870      --->    1A1B
```

10. 輸入一些小於 10 的正數，然後用直條圖來表示數字的大小，例如：以下為輸入 3，5，2，4，6 的輸出圖形。

```
                  *
      *           *
      *       *   *
  *   *       *   *
  *   *   *   *   *
  *   *   *   *   *
```

11. 以下為某日各國貨幣的匯率兌換資料，

貨幣	港幣	美元	日圓	新臺幣	人民幣	歐元	英鎊
匯率	4.294	33.49	0.2786	1	4.046	33.03	51.3

請利用上表資料，依次讓各國幣值為 1 單位，列印其與它國之間的兌換匯率。

12. 若高速公路臺北，桃園，新竹，臺中，嘉義，臺南，高雄等幾個城市的主要出口里程數分別為：

城市	臺北	桃園	新竹	臺中	嘉義	臺南	高雄
里程	25	49	95	178	264	327	373

如果某客運公司的票價為：總里程在 50 公里內每一公里以 2.5 元計，總里程在 51 到 200 公里之間則每一公里以 2.2 元計，總里程在 200 公里以上則每一公里為 2 元。請寫一個程式印出所有城市之間的里程表及票價表，以下為部份的輸出結果：

里程表	臺北	桃園	新竹
臺北	0	24	70
桃園	24	0	46
新竹	70	46	0

票價表	臺北	桃園	新竹
臺北	0	60	154
桃園	60	0	115
新竹	154	115	0

13. 某課程修課人數共有 n 人，若任課老師要在滿分為 100 分的考試成績，依每二十分作一橫條圖來顯示學生成績分佈的百分比人數，請寫一個程式將學生成績分佈的橫條圖畫出，並將每一區間人數寫在橫條末端，輸出結果如下：

```
 0 - 20  |***** 5
21 - 40  |********** 10
41 - 60  |***************** 17
61 - 80  |*********************** 23
81 -100  |************ 13
```

14. 上題請用直條圖顯示，區間人數寫在直條上端？

15. 請寫一個程式模擬兩個骰子點數和的出現機率。

16. 任意產生兩個整數陣列，請寫一個程式找出兩陣列的交集。

17. 任意產生兩個整數陣列，請寫一個程式找出兩陣列的聯集。

18. 假設樂透號碼介於為 [1,42] 之間，請寫程式驗證一組六個號碼中三個號碼的機率約為 2.722%，即 $\frac{C(36,3)\,C(6,3)}{C(42,6)}$。

19. 今有 n 個重量相同的雞蛋，假設其中有一個雞蛋臭掉，造成重量改變 (重量可能增加或減少)，請寫一個程式模擬天平秤重，找出壞掉的雞蛋。

20. 今有 n 個金蛋，其中有一個為偽蛋，且其重量較輕，請寫一個程式模擬天平秤重，以每次分兩堆的方式將偽蛋找出。

21. 轉蛋機內有 4 組公仔，假設每組公仔有 6 隻， 24 隻公仔隨意地擺放在轉蛋機內，請寫一個程式模擬轉蛋機，計算平均要轉幾次才能得到一組完整的公仔。

22. 52 張撲克牌中任選 5 張，請寫一個程式模擬，驗證得到兩個對子 (two pairs) 的機率約為 0.047539。兩個對子的定義為兩種不同點數的牌各兩張，另加一張不同點數的牌。

23. 若一多項式為：

$$f(x) = c_0 + c_1 x + c_2 x^2 + \ldots + c_n x^n = \sum_{i=0}^{n} c_i x^i$$

輸入多項式的最高次數 n 後，利用亂數產生兩多項式的各項係數，請計算此兩多項式的乘積。

24. 承上題，多項式 f(x) 的微分式可寫為：

$$f'(x) = c_1 + 2 c_2 x + 3 c_3 x^2 + \ldots + n c_n x^{n-1} = \sum_{i=1}^{n} i c_i x^{i-1}$$

隨意產生一多項式 f(x)，請利用以上公式印出多項式的微分式 f′(x)。

25. 若一多項式的係數大多為零，例如：$f(x) = 3 + x^{41} + x^{592}$，則可考慮使用兩個陣列只儲存多項式內的必要資料，假設第一個陣列儲存多項式的非零係數，則另一個陣列儲存非零係數所對應的次數 (degree)，請用此種儲存方式計算多項式的微分式後印出。

26. 利用上題的儲存方式計算兩個多項式的加法。請留意，兩個多項式的加法可能使得相加後的係數為零。

27. 若一多項式為：

$$f(x) = c_0 + c_1 x + c_2 x^2 + \ldots + c_n x^n = \sum_{i=0}^{n} c_i x^i$$

則可將其改寫為

$$f(x) = c_0 + x(c_1 + x(c_2 + \ldots + x(c_{n-2} + x(c_{n-1} + c_n x))\ldots)$$

這種改寫方式被稱為巢式乘法 (nested multiplication algorithm)。請撰寫程式自行產生一多項式，利用巢式乘法演算方式計算此多項式在 x 為 0, 1, ..., 5 的函式值。

28. 請寫一個程式來作橋牌的分牌動作，請依橋牌方式來輸出，花色為：黑桃 S(♠)，紅心 H(♡)，鑽石 D(♢)，黑花 C(♣)，數字為：2， ...，10，J，Q，K，A，請依東西南北方式分牌，如下：

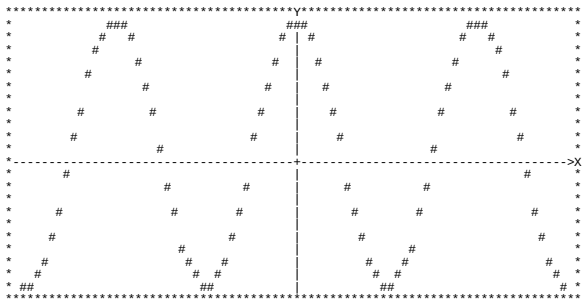
```
                    S : 9 8 4
                    H : 7 6 4
                    D : A J 10 9 5
                    C : Q 6
S : A K Q 10 7 2                          S : J 3
H : J 9 2                  北             H : Q 8 5 3
D : Q 8                西       東        D : 7 4 3
C : 8 7                    南             C : A K J 10
                    S : 6 5
                    H : A K 10
                    D : K 6 2
                    C : 9 5 4 3 2
```

29. 以下為賓果遊戲的規則，假設數字盤上有 7×7 個數字，介於 1 到 49 之間，且各不重複。現在用亂數產生 1 到 49 之間的數字，然後將數字盤上相同的數字去除，當所去除的數字呈現至少連續五個成一排時，遊戲就結束，請撰寫一個程式模擬此過程。這裡的一排數字包含數字呈現直排，橫排與斜排都算。

30. 請設計 0 到 9 十個 5×4 的點矩陣數字，然後輸入一個整數後，印出此整數的對應點矩陣圖形，例如若輸入 2987340，則列印出：

```
* * *     * *     * *   * * * *  * * *    *   *     * *
    *  *    *  *    *       *       *  *   *   *     *
  * *     * * *   * *       *   * * *   * * * *  *     *
*             *  *    *   *           *      *   *     *
* * * *  * * *     * *     *     * * *       *     * *
```

31. 早期的電腦螢幕都是所謂的文字模式顯示器， 程式設計員若要顯示某函式的圖形通常須要用點矩陣的方式將圖形依比例畫在螢幕上，請用寬度 80 格為 X 軸，長度 24 格為 Y 軸，畫出 cos(x) 在 -10 至 10 的圖形。這裡使用的 `cos(x)` 函式須要用到 `cmath` 標頭檔， 為清楚起見，請將 X 軸與 Y 軸也顯示出來，輸出形式如下：



32. 上題若要用 ANSI 跳離序列 (escape sequence)[83] 直接在螢幕上顯示又要如何撰寫？

33. 若要計算西元 Y 年 M 月 D 日為星期幾可以用以下公式，使用公式前先要將年份與月份做調整，基本上，以下的公式推導是將每年的三月一日當成一年的開始，如此一來，原有的三月 (March) 起算為第一月，四月被當成二月，其它則依此類推，因此原有的一月，二月就被當成十一，十二月，同時年份也要加以改變，推算日期若在三月之前，則年份須減一，例如：西元 2000-1-1 的 `Y = 1999`，`M = 11`，`D = 1`，而西元 1999-12-31 的 `Y = 1999`， `M = 10`，`D = 31`，公式為：

$$(\ \mathtt{Y} + [\mathtt{Y}/4] - [\mathtt{Y}/100] + [\mathtt{Y}/400] + [2.6 \times \mathtt{M} - 0.2] + \mathtt{D}\ )\ \bmod 7$$

這裡 [x] 為 x 的整數部份，mod 為餘數運算子

(a) 請利用此公式印出西元某年的年曆。

(b) 請以兩個月份為一單位印出年曆。

(c) 由於夏天的白晝時間較長，往往清晨五點太陽就已經昇起。為了節約能源，許多國家每年會在某些時段統一將時鐘往後調整一個鐘頭，這樣的方式稱為日光節約時間 (daylight time saving) 或夏令時間 (summer time)。以美國為例，每年的日光節約時間開始於每年四月的第一個星期天，結束於十月的最後一個星期天，請計算出美國在公元 2000 至 2010 年之間的夏令時間的開始及結束日期，同時印出每年夏令時間的總天數？

34. 數獨 (Sudoku) 為一 9×9 的數字方盤，其中每一行與每一列的數字都在 1 到 9 之間，且數字不得重複。此外方盤另以每三行與每三列方式將之切割成 9 個 3 × 3 的小方盤，每個小方盤的數字也是在 1 到 9 之間，數字也不得重複。例如以下是一個簡單的數獨盤：

1	2	3	4	5	6	7	8	9
4	5	6	7	8	9	1	2	3
7	8	9	1	2	3	4	5	6
9	1	2	3	4	5	6	7	8
6	7	8	9	1	2	3	4	5
3	4	5	6	7	8	9	1	2
8	9	1	2	3	4	5	6	7
5	6	7	8	9	1	2	3	4
2	3	4	5	6	7	8	9	1

根據此數獨盤，採取以下策略產生一個新的數獨盤後印出：

(a) 在每三行或每三列內，隨機對調其內的任兩行或任兩列若干次

(b) 以每三行或每三列為一對調單位，隨意對調任兩單位若干次

(c) 隨機排序 1 到 9，然後以新的排列數字取代數獨盤內的數字

35. 承上題，請利用亂數，自定規則去除上題所產生的數獨盤內的若干個數字後輸出。

36. 請自行設計 0 到 9 等十個 4 × 5 數字點矩陣，撰寫程式，讀入一整數數字與圖形放大倍數 n，列印此數字所對應的放大點矩陣圖形。所謂放大倍數是指點矩陣圖形的縱橫方向各自放大 n 倍。例如，以下為數字 345 被放大成 2 倍與 3 倍的圖形：

```
3333    4  4    5555
   3    4  4    5
 333    4444    5555
   3       4       5
3333       4    5555
```

原來大小

```
33333333    44    44    55555555
33333333    44    44    55555555
      33    44    44    55
      33    44    44    55
  333333    44444444    55555555
  333333    44444444    55555555
      33          44          55
      33          44          55
33333333          44    55555555
33333333          44    55555555
```

放大 2 倍

```
333333333333    444       444    555555555555
333333333333    444       444    555555555555
333333333333    444       444    555555555555
         333    444       444    555
         333    444       444    555
         333    444       444    555
   333333333    444444444444    555555555555
   333333333    444444444444    555555555555
   333333333    444444444444    555555555555
         333              444            555
         333              444            555
         333              444            555
333333333333              444    555555555555
333333333333              444    555555555555
333333333333              444    555555555555
```

放大 3 倍