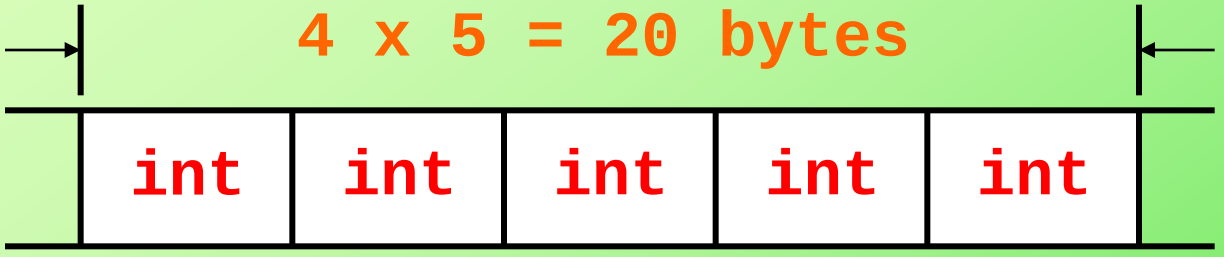
Chapter 6

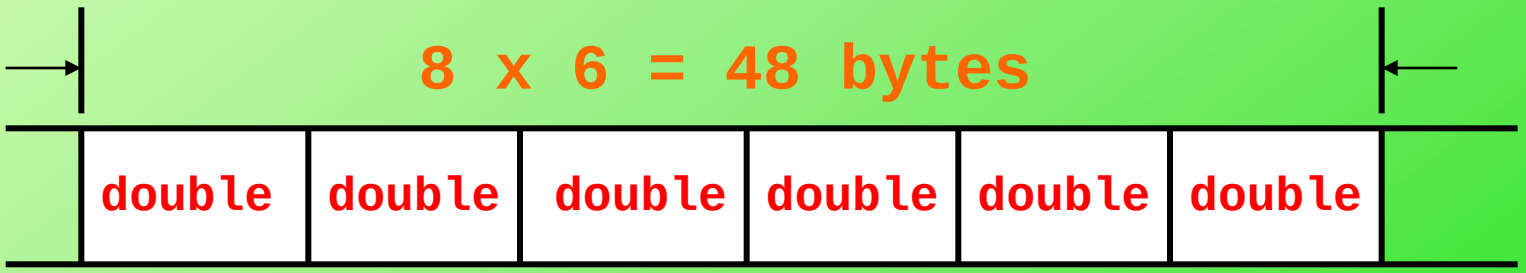
陣列

陣列儲存機制

- 陣列：一群緊鄰且相同資料型別的元素集合在一起



5 個整數



6 個雙精確度浮點數

array

傳統陣列

- 陣列長度：陣列儲存的元素個數

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

- 陣列的長度不可為變數，須在執行之前就已經確定

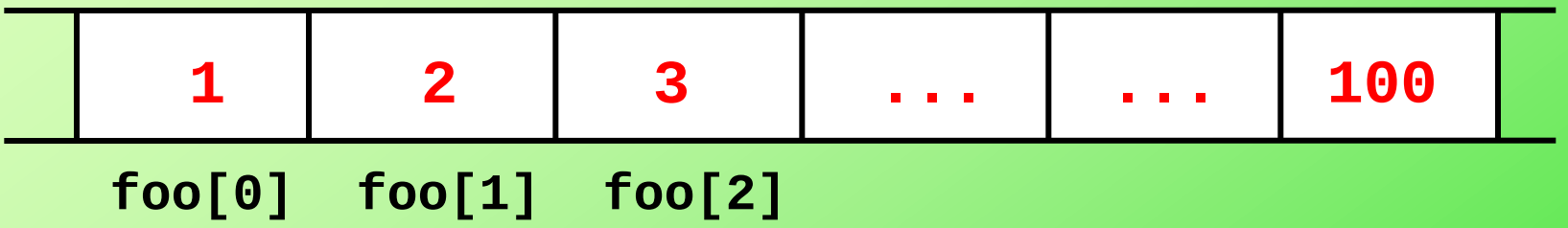
```
const int SIZE = 10;
int foo[SIZE];
int i;
cin >> i;
int bar[i];        // 錯誤，陣列的長度不可為另一變數
```

❖ 新標準容許陣列長度為變數

使用傳統陣列

- 陣列元素的指定須一個一個地指定

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



  - ❖ 陣列的下標由 **0** 開始

- 兩陣列間的指定須以個別元素的指定方式一一進行

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

傳統陣列初始值的設定

- 陣列初始值設定

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

- 若括號內的元素個素少於陣列長度，則剩下的所有元素皆會自動被設為零

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

陣列長度的處理

- 陣列長度經常用常數設定用以方便迴圈的處理

```
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;
```

- 陣列長度可以用 sizeof 函式計算出來

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

❖ sizeof(foo)：foo 陣列總共佔用的位元組數目

sizeof(int)：一個整數所佔用的位元組數目

範例：打亂陣列元素

foo 原本陣列裡的元素：

2	4	6	8	10

打亂 foo 陣列的元素後：

8	4	10	2	6

```
//由後往前一一與之前的元素對調
for( i = size-1 ; i > 0 ; --i ){
    j = rand() % (i+1);
    if (i == j) continue;
    tmp    = foo[i];
    foo[i] = foo[j];
    foo[j] = tmp;
}
```

巴斯卡三角形（一）

- 二項式定理

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

- 組合公式

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

程式　輸出

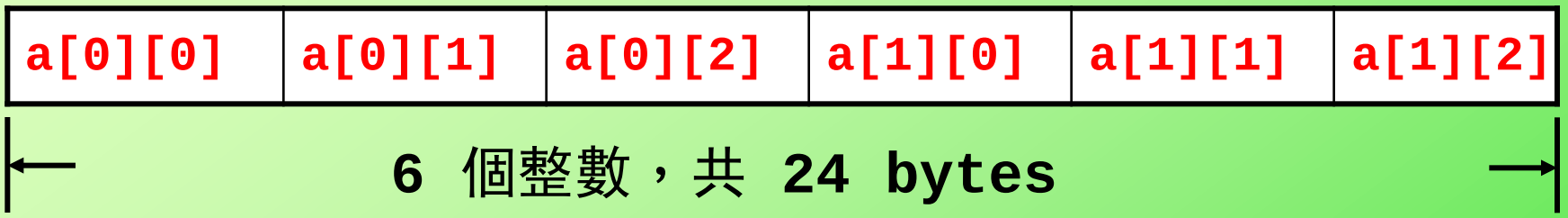
```
b[i]= { 1                  每列頭尾元素
      { a[i-1]+a[i]        其它
```

```
                1                    ← a[i]
            1       1
        1       2       1
    1       3       3       1        ← b[i]
1       …       …
```

Pascal triangle

多維陣列

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



- `float b[2][2][3] ;` `// 共 12 個元素, 48 個位元組`

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]

- 多維陣列的總個數：

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

多元陣列的初始值設定（一）

- 一個大括號方式的設定：

```
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 };
```

- 多個大括號方式的設定：

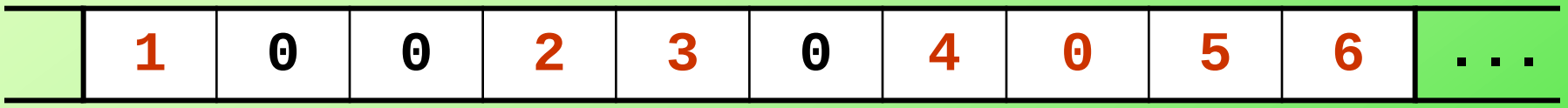
```
int a[2][3] = { {1},{2,3} };
int b[2][3] = { {1,0,0},{2,3,0} };
int c[3][3] = { {1},{1,1},{1,1,1} };
```

以上用矩陣表示為：

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

多維陣列的初始值設定（二）

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



- 多維陣列的第一維長度可省略，編譯器會自行計算

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

  - ❖ 此時第一維長度為 3

矩陣相乘

- $\begin{bmatrix} C \end{bmatrix} = \begin{bmatrix} A \end{bmatrix} \begin{bmatrix} B \end{bmatrix}$

這裡　A 為 m x p

B 為 p x n

C 為 m x n

程式　輸出

若用下標型式來表示

$$C_{ij} = \sum_{k=0}^{p-1} a_{ik}\, b_{kj}$$

i 屬於 {0..m-1}

j 屬於 {0..n-1}

巴斯卡三角形（二）

- 巴斯卡三角形：矩陣方式表示

$$\begin{pmatrix} p \end{pmatrix} = \begin{pmatrix} 1 & & & \\ 1 & 1 & & \\ 1 & 2 & 1 & \\ 1 & 3 & 3 & 1 \\ \cdots & \cdots & \cdots & \cdots \end{pmatrix}$$

程式　輸出

下三角形矩陣部分可寫成

$$p[i][j] = \begin{cases} 1 & , \quad i=j \text{ 或 } j=0 \\ p[i-1][j-1] + p[i-1][j] & , \quad \text{其它} \end{cases}$$

向量陣列

- 向量陣列： **vector<T> , T 為元素型別**

  陣列元素個數可以在執行時，動態地加以改變

```
#include <vector>
unsigned int n ;
cin >> n ;

vector<int>  a(n) ;  // a 為 n 個元素的整數向量陣列
int          b[n] ;  // 錯誤
```

❖ 使用向量陣列時，須加上 **vector** 標題檔

vector

向量陣列物件

- 向量陣列物件：向量陣列變數

```
vector<int>   a(2);   // a 為整數向量陣列，有 2 個元素
vector<char>  b(5);   // b 為字元向量陣列，有 5 個元素
```

- 使用方式與傳統陣列相同

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

- 向量陣列長度：元素個數

```
cout << " a 物件有" << a.size() << "元素 \n";
```

簡單向量陣列物件初值設定

- 簡單初值設定：

```
vector<int>  a(3,1);          // a 有 3 個整數
                              // 起始值皆為 1
vector<char> b(5,'a');        // b 有 5 個字元 'a'
```

- 無法使用傳統陣列方式設定初值：

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

向量陣列物件的指定

- 相同元素型別的向量陣列物件間，可使用指定方式複製所有元素

```
vector<char> a(5,'x');   // a 有 5 個 'x' 字元
vector<char> b(3,'y');   // b 有 3 個 'y' 字元
a = b;                   // 指定後，a 與 b 相同
```

但傳統陣列變數不能如此炮製

```
int  a[3] = {4,5,7};
int  b[7] = {0};
a = b;                   // 錯誤
```

包牌程式：迴圈篇

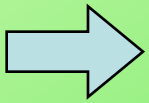
- 列印由 m 個不同數字中任選 n 個數字的所有組合其總數為

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

程式 輸出

- 從 {1 2 3 4 5} 裡任意取 3 個數字的組合

$$C_3^5$$



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

10種

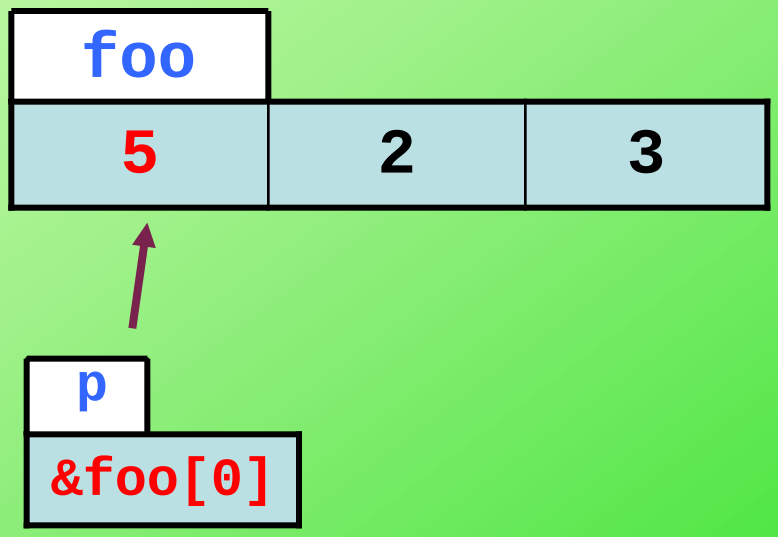
❖ 處理方式 ： 使用 n 層迴圈

combination

陣列與指標（一）

- 陣列與指標的關係

```
int foo[3] = {5,2,3};
int *p     = &foo[0];
```




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

- 陣列名稱可看成是個指標永遠指向陣列的首位元素

```
  for(i=0 ; i<3 ; ++i) cout << p[i]   << endl;
⇔ for(i=0 ; i<3 ; ++i) cout << foo[i] << endl;
```

陣列與指標（二）

- 指標可以指向陣列的任何元素

```
int foo[] = {5 , 2 , 3 , 4 , 1};
int    *p = foo;            // p 指向 foo 陣列的首位元素
```

```
// 變更 p 所指向的位址將 foo 陣列元素依次列印
for( i = 0 ; i < 5 ; ++i , ++p ) cout << *p << endl;
```

- 陣列名稱則不能改以指向其他位址

```
foo += 1;          // 錯誤
```

- 若另一陣列為

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

指標與一維陣列

- 一維陣列

```
int foo[N];          // N 為常數
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]`

指標與二維陣列

■ 二維陣列

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

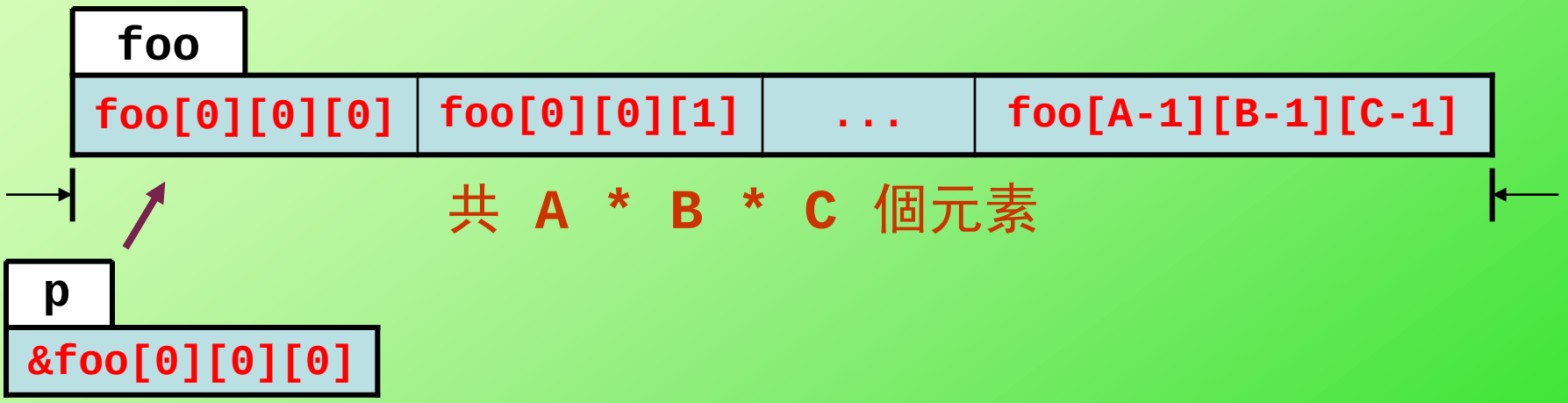
位址 &foo[i][j] <=> p+i*C+j

元素 foo[i][j] <=> *(p+i*C+j) <=> p[i*C+j]

指標與三維陣列

- 三維陣列

```
int foo[A][B][C];          // A , B , C 為常數
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]
```

指標指向多維陣列

- 指標操作：

```
const int R = 2 , C = 3;
int i , j;
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
}
```

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

多維陣列與指標

一維陣列

```
int foo[N] ;          int *p = &foo[0] ;
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] ;
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] ;
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]
```

多維陣列與陣列式指標（一）

- 定義一個 3 列 4 行 的整數陣列

```
int foo[3][4];
```

foo[1][2] : 第 2 列起始位址起算的第 3 個整數

foo[2][3] : 第 3 列起始位址起算的第 4 個整數

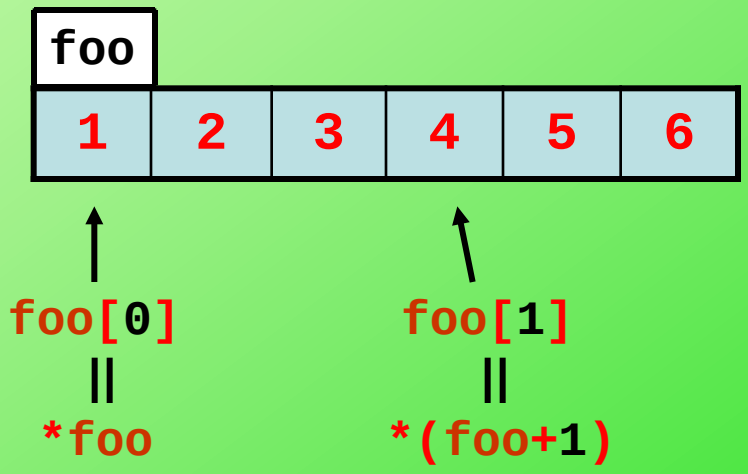
❖ foo[i][j] 的第一個下標 foo[i] 儲存第 i+1 列的起始位址，為一個指標，而非元素

多維陣列與陣列式指標（二）

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

相同的表示方法：

```
foo[i][j] <=> (*(foo+i))[j]
          <=> *(*(foo+i)+j)
```




```
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;
   }
}
```

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

多維陣列與陣列式指標（三）

- 二維陣列的第一個下標移動的單位是以行為單位

```
// 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
```

❖ 以上指標 P 是以 1 個整數為移動單位
但陣列 foo 則以 3 個整數為移動單位

多維陣列與陣列式指標（四）

- 陣列式指標：指標以每次移動若干個型別空間為單位

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

int  (*p)[3]  = &foo[0] ; // p 為指標式陣列
                          // 每次以 3 個整數為移動單位

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

int  (*r)[3]  = foo+1 ;   // r 為陣列式指標
                          // 指向 foo 第 2 列的起始位址
```

❖ 以上指標的小括號不可省略

多維陣列與陣列式指標（五）

- 陣列式指標程式

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

位址	*(p+i)+j	<=>	p[i]+j	<=> &foo[i][j]
元素	*(*(p+i)+j)	<=>	p[i][j]	<=> foo[i][j]

```
for( i = 0 ; i < R ; ++i ){
   for( j = 0 ; j < C ; ++j ){
      cout << p[i][j] << ' ';
      cout << endl;
   }
}
```

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

指標陣列 (一)

■ 指標陣列：陣列內的元素皆為指標

```
int  a[9];    // 9 個整數元素的陣列
int *p[9];    // 陣列內包含 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];

// 透過指標陣列間接設定 a 陣列資料
for( i = 0 ; i < 9 ; ++i ) *p[i] = i*i;
```

❖ 下標運算子 [] 的處理順序較參照運算子 * 為先，因此 *p[i] 相當於 *(p[i])

指標陣列（二）

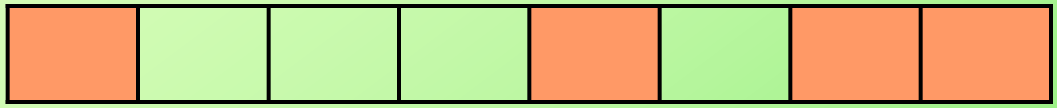
- 使用向量陣列則須將指標型別放到 <> 內

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

// 向系統取得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 ;

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

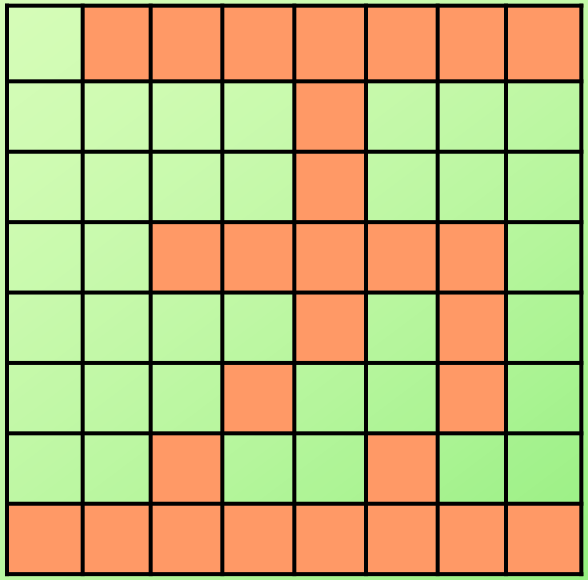
點矩陣文字

$139 = 1x2^7 + 0x2^6 + 0x2^5 + 0x2^4 + 1x2^3 + 0x2^2 + 1x2^1 + 1x2^0$

圖示如右



若一個文字用 8 個位元組來表示



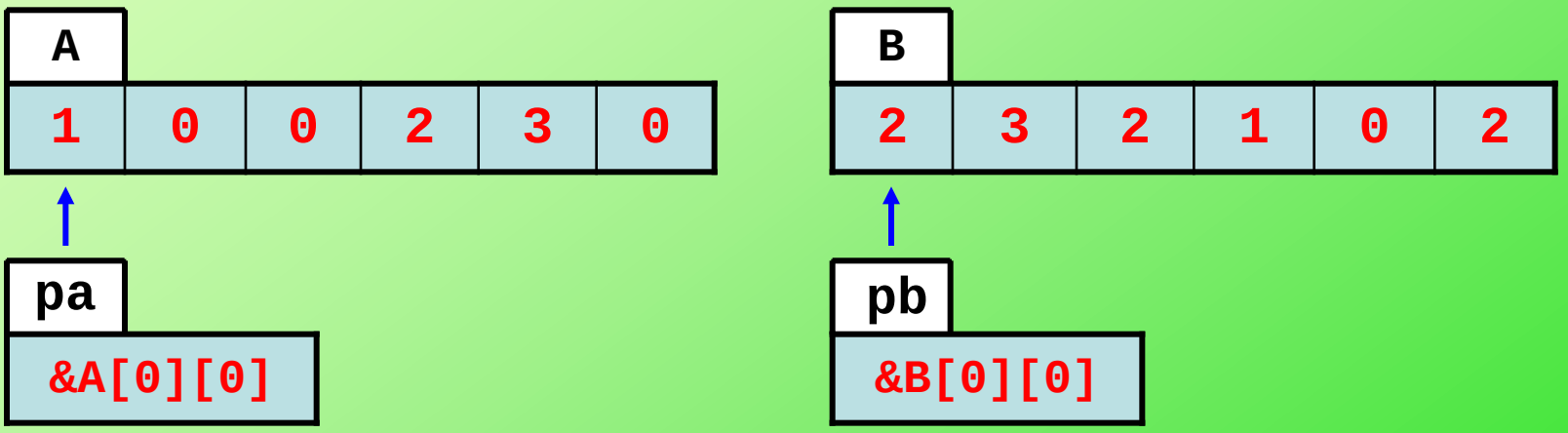
```
0 1 1 1 1 1 1 1   => 127
0 0 0 0 1 0 0 0   => 8
0 0 0 0 1 0 0 0   => 8
0 0 1 1 1 1 1 0   => 62
0 0 0 0 1 0 1 0   => 10
0 0 0 1 0 0 1 0   => 18
0 0 1 0 0 1 0 0   => 36
1 1 1 1 1 1 1 1   => 255
```

則 `int five[8] = { 127 , 8 , 8 , 62 , 10 , 18 , 36 , 255 }`

程式　輸出

矩陣相乘：利用指標

$$A = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 3 & 0 \end{pmatrix} \qquad B = \begin{pmatrix} 2 & 3 \\ 2 & 1 \\ 0 & 2 \end{pmatrix}$$




```
A[i][j]  <=>  *( pa + i*3 + j )
B[i][j]  <=>  *( pb + i*2 + j )
```

程式　輸出