國立屏東大學 資訊工程學系 C++程式設計入門教材

11. 指標與參考

指標(Pointer)往往是許多同學在C++語言的學習過程中，所遇到的第一個巨大的障礙。其實，只要將觀念建構清楚，指標其實一點都不困難。本章將就指標的基本觀念與相關語法進行說明。除了指標外，本章也將就C++語言不同於C語言的另一項創新 – 參考(Reference)進行說明。

截至目前為止，我們所寫的程式都是透過變數來儲存及操作各式的資料，同時我們也已經瞭解一個變數的值(Value)其實是儲存在某個記憶體位置內，在程式中所有對該變數的操作，其結果就是對該記憶體位置內的值進行存取及運算。在本章中我們將為你介紹一種不同於變數的資料存取方法 — 透過指標來進行資料的存取。從某個角度來看，指標其實也可以視為是一種變數(不是說不同於變數嗎？別急，請讓筆者慢慢地幫你解釋…)，它也會對應到一塊記憶體空間，只不過它是一種比較特別的變數，它所存放的並不是值，而是記憶體位址(Memory Address)。因此，在程式中我們可以透過指標來對特定的記憶體位址進行操作。本章後續將針對指標的概念、相關的語法以及操作進行詳細的說明。

11.1 基本觀念

要學習指標與參考就必須先瞭解記憶體與變數的關係。在開始之前，先讓我們先談談變數與記憶體位址的相關概念：

電腦系統在執行程式時，所有的資料都是存放在記憶體當中。一個程式在執行時，會經由作業系統的幫助取得一塊專屬的記憶體空間，所有在程式中所宣告的變數、常數、陣列等，都會在這其中分配到一塊記憶體空間，而後續在程式中，只需要透過這些變數的名稱、常數的名稱、陣列的名稱，就可子對它們所配置到的記憶體空間來進行各項操作。例如以下的變數宣告：

```
int x=38;
```

在程式執行時，上面這個宣告就會在記憶體裡取得一塊4個Bytes的空間，並將數值38存放於該記憶體空間裡。請參考figure 1，我們假設該變數被配置到了從0x7ffff34fff00記憶體位址開始的連續4個Bytes(因為int整數佔4個Bytes):

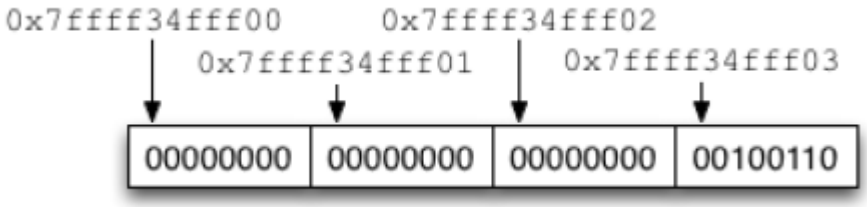

Fig. 1: 變數x所配置到的記憶體空間

你可以在figure 1裡看到x變數的初始值38已經存放在記憶體裡了 — 當然，是用二進制的數值存放，也就是00000000000000000000000000100110。

通常描述記憶體空間的圖有兩種畫法，其一為figure 1的橫式，或是figure 2的直式，兩者的意義相同。

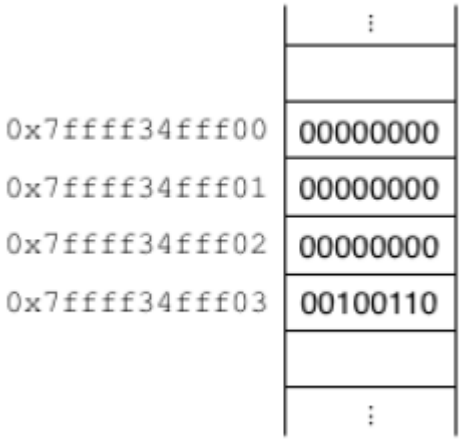

Fig. 2: 變數x所配置到的記憶體空間

記憶體位址的單位為byte，且通常以16進位來表示，0x7fff34fff00開頭處的0x就表示數值為16進位。

儘管在一個變數的背後有上述這些和記憶體相關的細節，但我們在過去的程式範例裡都只需要抽像地將變數x想像成在記憶體中的某塊空間，不用特別去注意變數到底配置在何處。figure 3就是我們常用的思考方式。

Fig. 3: 省略細節的變數x

如同我們在第3章所說明的，如果我們想要知道一個變數到底存放在哪個記憶體位址，可以使用取址運算子&來取得。請參考以下的程式，它宣告了一個整數變數，並將其值與記憶體位址加以輸出。

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

int main()
{
  int x;
  x=38;

  cout << "The value of x is " << x << endl;
  cout << "The memory address of x is "
       << showbase << hex << &x << endl;
  return 0;
}
```

11.2 指標變數(Pointer Variables)

指標變(Pointer Variable)，顧名思義即為一個變數，但其所儲存的不是“ 值 ”而是某個“ 記憶體位址(Memory Address)”，當然，我們也可以說指標所儲存的值為某個記憶體位址。其宣告方法等同一般的變數宣告，但在變數名稱前，必須加入一個星號 *，請參考以下的宣告語法：

指標變數宣告敘述，Pointer Variable Declaration Statement，語法定義

資料型態 *指標變數名稱;

註：此處的星號是語法的一部份，而不是代表可出現0次或多次的意思。

依據這個語法，我們可以宣告一個指標變數如下：

```
int *p;
```

其中p就是語法中的「指標變數名稱」，至於星號的位置可以有些彈性，以下的兩個宣告方式也都是正確的：

```
int * p;
int* p;
```

以上這三種宣告的結果都相同，都會建立一個可以儲存記憶體位址的空間，不過其中以，int *p;，這種宣告的方式是最常被使用的。細心的讀者會提出一個問題：「既然只是要得到一個可以儲存記憶體位址的空間，那又為何要宣告為int?，，這的確是一個好問題！其實這樣的宣告是要告訴編譯器，將來儲存在這裡的是一個記憶體位址，而且在那個記憶體位址中，所存放的是一個int型態的整數。讓我們回想一下前面使用過的例子：假設指標p所存放的記憶體位址是0x7ffff34fff00，而且在0x7ffff34fff00位址裡面所存放的是一個整數，也就是變數x的值。

C++語言也允許我們混合一般的變數宣告與指標變數宣告，請參考下面的這些宣告:

```
int *x, y;      // x是指標變數, y是一般變數
int i, *j;      // i是一般變數, j是指標變數
int* i,j, k;    // i是指標變數, j與k則是一般變數
```

在前述的例子中，指標變數都被宣告為 ”應該會 ”指向儲存整數(int)的記憶體位址，所以上述的宣告可視為宣告了一些“ 整數的指標變數”，C++語言允許我們將指標宣告為各種型態，例如下面這些都是可行的宣

告:

```
double *p;
char *p;
float *p;
```

請先執行下列程式，以確認你的系統上的記憶體使用情形：

```
int main()
{
  cout << "The size of an int is " << sizeof(int) << endl;
  cout << "The size of an int-pointer is " << sizeof(int *) << endl;
  return 0;
}
```

假設我們所得到的執行結果分別為4與8，由於其單位為Byte，表示一個int的整數將佔用32位元的記憶體空間，而一個整數指標佔用64位元。

以Linux系統為例，雖然一個記憶體位址佔64位元，但目前僅使用了其中的48位元，還有16位元是保留未使用的。而以48位元定址的系統，其最大的可定址空間為256TB，為來若將保留的16位元也拿來使用，則最大可定址到16EB，

(備註1024GB = 1TB, 1024TB = 1PB, 1024PB = 1EB) (EB = Exabyte, 唸做艾克薩 - 拜)

圖figure 4顯示了一個int整數x與整數指標p(它是由 int *p加以宣告)，假設整數變數x是儲存於記憶體位址中的0x7ffff34fff00 - 0x7ffff34fff03 (32位元的整數佔四個bytes)，其值為38，且假設整數指標p儲存於0x7ffff34fff68-0x7ffff34fff6f (8 bytes=64 bits)，其值為0x7ffff34fff00，

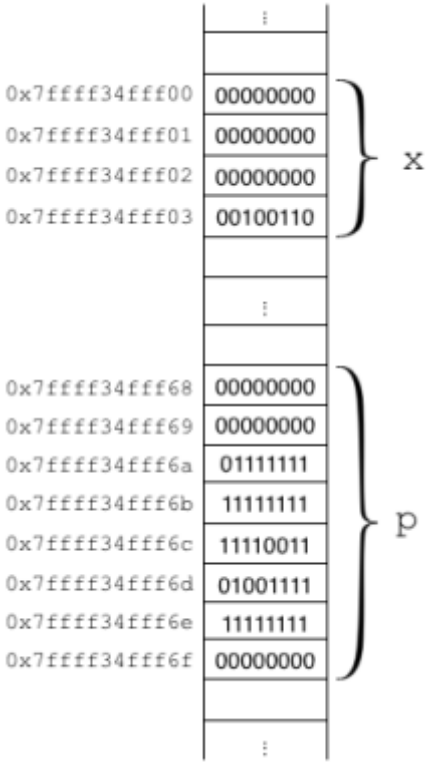

Fig. 4: 變數x與指標p

若不考慮記憶體位址的細節，我們可以將figure 4表示為figure 5。

Fig. 5: 指標p與變數x的抽象表式1

在圖中p是一個指向0x7ffff34fff00位址的指標，因為當初我們有宣告p的參考型態為int，所以C語言的compiler會知道，p所指向的是位於0x7ffff34fff00 - 0x7ffff34fff03的四個bytes，從抽象角度來看，由於p所儲存的值，正好是x所在的位址，因此我們更常用figure 6來加以表達這樣的一個關係。

Fig. 6: 指標p與變數x的抽象表式2

好了，我們現在已經學會“假設”指標變數p存放了變數x所在的記憶體位址時的表達方法，但問題是—該如何讓指標p“真的”去指向變數x呢？還記得我們曾在前面的章節中，提過一個變數可以使用取址運算子(Address-Of)&來取得其所在的記憶體位址嗎？我們可以透過下面的程式碼，來將整數變數x的位址，指定給整數指標p。

```
int x=38;
int *p;
p=&x;
```

```
int  x=38;
int *p=&x;
```

好的，如此一來，就如同figure 6一樣，p成為了指向整數變數x的指標。

還記得本章開頭時曾說「指標其實也可以視為是一種變數，只不過它是一種比較特別的變數… 存放的不是一般的數值，而是記憶體位址」 所以比照一般的變數賦值的做法

```
int  x = 38;  // 把"數值38"賦值給"變數x"
```

就好比 x ← 38

就可以把記憶體位址賦值給指標了

```
int *p = &x;  // 把"變數x所在的記憶體位址"賦值給"指標變數p"
```

就好比 p ←0x7fff34fff00

11.3 記憶體位址與間接存取運算子

除了取址運算子&可以用來取得變數所在的記憶體位址之外，C++語言還提供「間接存取運算子(Indirect Access Operator)」來存取指標變數所指向的記憶體位址中的值。間接存取運算子的運算符號是*，我們只要在指標變數前冠以*就可以取得該指標所指向的記憶體位址裡面的數值。請參考以下程式：

```
int x=38;
int *p;

cout << "x is located at " << showbase << hex << &x << endl;
```

其結果會輸出變數x所在的記憶體位址，再繼續使用下面的程式碼，則會讓指標變數p指向變數x所在的記憶體位址，並且將其值輸出。

```
p=&x;  // 讓指標變數p指向變數x所在的記憶體位址

cout << "The value of *p is " << *p << endl;
```

間接存取運算子*，不單是能夠將指標所指向的位址裡的值拿出來，它也能夠讓我們把數值放進去 — 不然為什麼要叫間接“ 存取 ”運算子，叫間接“ 取 ”運算子就好了。請參考下面的程式碼片段，想想看其執行結果為何？

```
int x, *p;

p = &x;

x = 6;
cout << "x=" << x << " *p=" << *p << endl;

*p = 8;
```

```
cout << "x=" << x << " *p=" << *p << endl;
```

11.4 指標賦值

C語言允許兩個指標，彼此間進行值的賦與 — 當然，此處的值指的是記憶體位址。考慮下面的程式碼，其執行結果為何？

```
int x=5, y=8, *p, *q;

p = &x;
q = &y;
cout << "x=" << x << " y=" << y << endl;
cout << "*p=%" << *p <<  " *q=" << *q << endl;
```

再考慮以下的程式，其執行結果又為何？

```
int x=5, y=8, *p, *q;

p = &x;
q = &y;

cout << "x=" << x << " y=" << y << endl;
cout << "*p=%" << *p <<  " *q=" << *q << endl;

*p = *q; // 將指標q所指向的記憶體位址裡面的數值，賦值給指標p所指向的記憶體位址裡

cout << "x=" << x << " y=" << y << endl;
cout << "*p=%" << *p <<  " *q=" << *q << endl;
```

再考慮以下的程式，其執行結果又為何？

```
int x=5, y=8, *p, *q;

p = &x;
q = &y;  // 請參考 fig. 4

printf("x=%d y=%d *p=%d *q=%d\n", x, y, *p, *q);

p = q;   // 請參考 fig. 5

printf("x=%d y=%d *p=%d *q=%d\n", x, y, *p, *q);
```

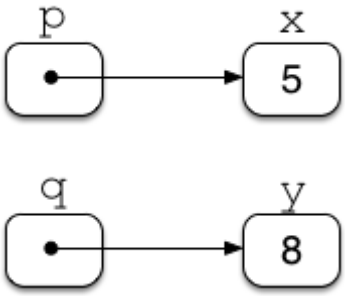

Fig. 7: 指標p與q分別指向變數x與y

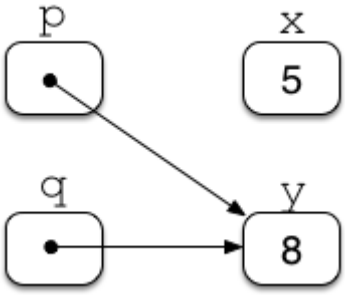

Fig. 8: 讓指標p指向指標q所指向的位址

11.5 指標與陣列

在C++語言中，因為陣列是記憶體中連續的一塊空間，因此我們也可以透過指標來存取儲存在陣列中的資料。本節將說明如何使用指標來存取陣列中的元素，並進一步探討指標與陣列的關係。

11.5.1 指標運算與陣列

考慮以下的程式碼：

```
int data[10]={1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

int i;

printf("data at %p.\n", data);
for(i=0;i<10;i++)
{
   cout << "data[" << i << "] at " << showbase << hex << &data[i] << endl;
}
```

其執行結果為：

```
data at 0x7fffb0aad1e0.
```

```
data[0] at 0x7fffb0aad1e0.
data[1] at 0x7fffb0aad1e4.
data[2] at 0x7fffb0aad1e8.
data[3] at 0x7fffb0aad1ec.
data[4] at 0x7fffb0aad1f0.
data[5] at 0x7fffb0aad1f4.
data[6] at 0x7fffb0aad1f8.
data[7] at 0x7fffb0aad1fc.
data[8] at 0x7fffb0aad200.
data[9] at 0x7fffb0aad204.
```

上述的程式碼宣告並在記憶體內配置了一塊連續十個整數的陣列，並且將其位址配置印出。由於陣列是連續的空間配置，在上面的例子中，data位於0x7fffb0aad1e0，那麼其第一筆資料(也就是data[0])就是位於同一個位址，或者更詳細的說，是從0x7fffb0aad1e0~3，這四個記憶體位址。由於一個位址代表一個byte，四個bytes就剛好表示一個32位元的整數。其後的每筆資料也都剛好間隔4個bytes。

我們可以用一個指標指向陣列的第一筆資料：

```
int *p;

p = &data[0];

或是

p = data;

或者

p = &data;  // 這行會有編譯的警告，因為p應該要指向一個整數，但data其實只是一個陣列所在的記憶體位址

p = (int *)&data; // 這樣就沒問題了
```

上述的程式碼宣告了一個整數指標p指向陣列所在之處，如figure 9。

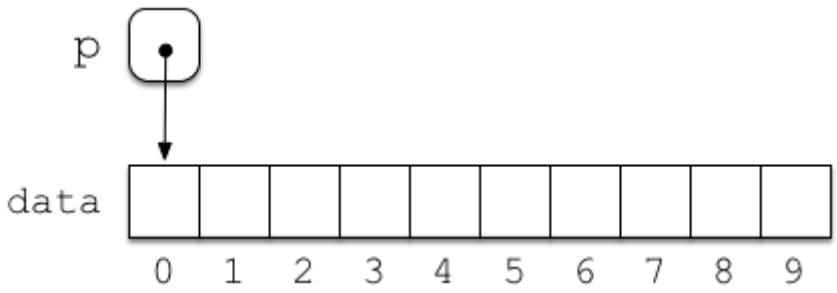

Fig. 9

我們可以透過*p來存取data[0]的數值資料。　如果我們要使用指標p來存取陣列中別的元素，例如第3筆資料(也就是data[2])，則可以用：

```
printf("%d\n", *(p+2) ); // 這行程式中的*(p+2)就等於 data[2];
```

考慮到p目前指向的是0x7fffb0aad1e0，也就是data[0]所在的位址，p+2並不會等於0x7fffb0aad1e0+2，而是等於0x7fffb0aad1e0 + 2*sizeof(int)，因為操作的對象p是一個指向整數的指標，因此若對它進行加法的運算，其運算單位是以整數的大小為依據，所以p+2會等於0x7fffb0aad1e8，這也就是data[2]所在的位址。

指標除了可以進行加法的運算外，也可以進行減法的運算：

```
int *p;
int *q;

p=&data[5];
q = p-2;  // q現在指向data[3]
p -= 3;   // p現在指向data[2]
```

前面提到，指標的運算是以其參考型態的大小為依據。當運算元皆是指標時，其運算結果也是會轉換為其參考型態的大小間的差距：

```
int i;
p = &data[5];
q = &data[2];

i = p - q; // i的值為3
i = q - p; // i的值為-3
```

指標還可以使用關係運算子(包含<, < =, ==, >=, >, !=)進行比較，依指標值其比較結果可以為 false或true。

```
p = &data[5];
q = &data[2];

if( p > q )
   printf("The position of p is after that of q.\n");
if( *p > *q)
   printf("The value of p is larger than that of q.\n");
```

11.5.2 以指標走訪陣列

下面的程式，配合迴圈的使用，利用指標來將陣列中每個元素都拜訪一次：

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

```
int main()
{
  int data[10]={1,2,3,4,5,6,7,8,9,10};

  int i;
  int *p;

  p=&data[0];
  for(i=0;i<10;i++)
    cout << "data;" << i << "]=*(p+" << i << ")=" << *(p+i) << endl;

  return 0;
}
```

當然，我們也可以直接用指標來操作:

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

#define Size 10

int main()
{
  int data[Size]={1,2,3,4,5,6,7,8,9,10};

  int i;
  int *p;

  p=&data[0];
  for(p=&data[0];p<&data[Size];p++)
    cout << *p << endl;
  return 0;
}
```

改用while迴圈，並配合「++」更新指標：

```
 p=&data[0];
  while (p<&data[Size])
  {
    cout << *p++ << endl;
  }
```

11.5.3 指標與陣列互相轉換使用

我們也可以直接把陣列當成一個指標來使用：

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

#define Size 10

int main()
{
  int data[Size]={1,2,3,4,5,6,7,8,9,10};
  int i;

  for(i=0;i<Size;i++)
    cout << *(data+i) << endl;

  cout << endl;

  int *p;
  for(p=data;p<data+Size;p++)
    cout << *p << endl;

  return 0;
}
```

當然，也可以指標當成陣列來使用：

```
  int data[Size]={1,2,3,4,5,6,7,8,9,10};
  int *p = data;
  int i;

  for(i=0;i<Size;i++)
    cout << p[i] << endl;

  cout << endl;

  int *p;
  for(p=data;p<data+Size;p++)
    cout << *p << endl;
  return 0;
}
```

11.5.4 常見的陣列處理

本節以指標操作一些常見的陣列處理：

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

#define Size 10

int main()
{
  int data[Size]={1,2,3,4,5,6,7,8,9,10};

  int i;
  int *p;
  int sum=0,sum2=0;

  for(p=&data[0];p<&data[Size];p++)
  {
    sum += *p;
  }
  cout << "sum = " << sum << endl;

  while(p>=(int *)&data)
  {
    sum2+= *p--;
  }
  cout << "sum2 = " << sum2 << endl;

  return 0;
}
```

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

#define Size 10

int main()
{
  int data[Size]={321,432,343,44,55,66,711,84,19,610};

  int i;
  int *p;
  int max, max2;

  max = *(p = &data[0]);
  for( ; p<&data[Size]; p++)
```

```
     max = (max < *p) ? *p : max;
/*{
    if( max < *p)
      max = *p;
  }*/

  cout << "max = " << max << endl;

  return 0;
}
```

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

#define Size 10

int main()
{
  int data[Size]={3451,25,763,3454,675,256,37,842,3439,510};

  int *p, *q;

  for(p=&data[0];p<&data[Size-1];p++)
    for(q=p+1; q < &data[Size]; q++)
      if(*p<*q)
      {
        int temp = *p;
        *p = *q;
        *q = temp;
      }

  for(p=&data[0];p<&data[Size];p++)
    cout << *p << endl;
  return 0;
}
```

11.5.5 以陣列做為函式的引數

在函式的設計上，可以陣列做為引數。請參考下面的例子：

```
int sum( int a[], int n)
{
   int i=0, s=0;
   for(;i<n;i++)
   {
      s+=a[i];
```

```
   }
   return s;
}
```

在上面的例子中，我們設計了一個可以計算陣列中元素和的函式。我們可以下面方式呼叫此函式：

```
int data[10]={12,522,43,3423,23,21,34,22,55,233};
int summation = 0;
...

summation = sum(data,10);
```

由於陣列與指標可互相轉換的特性，前述的函式也可改成：

```
int sum( int *a, int n)
{
   int i=0, s=0;
   for(;i<n;i++)
   {
      s+=a[i];
   }
   return s;
}
```

同樣地，在呼叫時也可以用下列的方法：

```
summation = sum( &data[0], 10);
或者
summation = sum( (int *)&data, 10);
```

另外，也可以使用

```
summation = sum( &data[3], 5 );
```

來計算從陣列第4筆元素開始，往後5筆的和。

11.5.6 指標與多維陣列

本節以二維陣列為例，探討指標與多維陣列的關係。請參考以下的例子：

```
#include <iostream>
#include <iomanip>
using namespace std;

#define ROW 3
#define COL 5

int main()
{
  int data[ROW][COL];

  int i, j;

  for(i=0;i<ROW;i++)
    for(j=0;j<COL;j++)
      data[i][j]= i*COL+j;


  for(i=0;i<ROW;i++)
  {
    for(j=0;j<COL;j++)
    {
      if(j>0)
        cout << ", ";
      cout << setw(3) << data[i][j];
    }
    cout << endl;
  }

  return 0;
}
```

在這個例子中，我們宣告了一個ROWxCOL(3×5)的二維陣列，給定其初始值後將陣列內容輸出。一般而言，我們可以將二維陣列視為一個二維的表格，如figure 10所示。

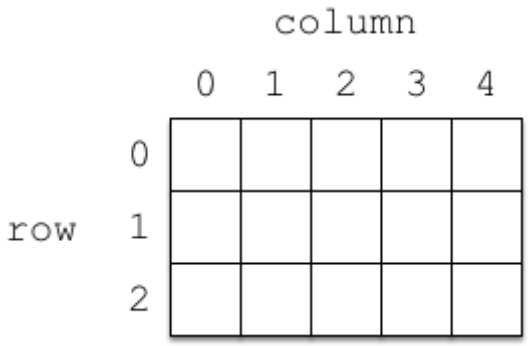

Fig. 10

我們可以宣告一個指標，來存取這個二維陣列，例如：

```
  int (*p)[COL];
```

```
i = 0;
for(p=&data[0]; p<&data[ROW] ; p++)
{
  for(j=0;j<COL;j++)
  {
    (*p)[j] = i*COL+j;
  }
  i++;
}
```

但其實在記憶體的配置上，仍是以連續的空間進行配置的，如figure 11所示。

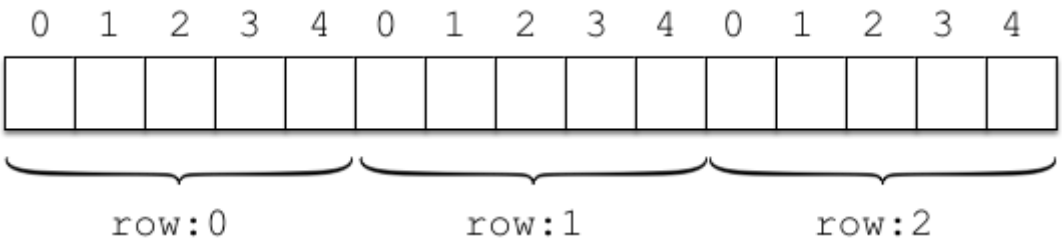

Fig. 11

請試著寫一個程式，輸出一個二維陣列各個元素的記憶體位址，來驗證上述的說法。

所以，同樣的初始值給定的程式碼也可以改寫如下：

```
int *p;

for(i=0, p=&data[0][0]; p<= &data[ROW-1][COL-1];i++, p++)
  *p = i;
```

如果要單獨取出二維陣列中的第i列(row)，那麼可以下列程式碼完成：

```
p = &data[i][0];
```

或者

```
p = data[i];
```

但是若要取回二維陣列中的某一行(column)，那麼又該如何呢？留給同學做練習吧！

11.6 指標與函式

11.6.1 以指標做為函式引數

一個C++語言的函式(Function)可以接受多個引數(Arguments)，經計算後傳回單一的數值，但若要讓函式傳回多個數值，就無法做到了。假設我們需要設計一個函式，接受一個double數值做為引數，並將其整數與小數部份分別傳回，則可以考慮以下的做法：

```
void decompose(double val, long *int_part, double *frac_part)
{
   *int_part = (long) val;
   *frac_part = val - ((long)val);
}
```

這個函式decompose並沒有任何的傳回值，可是它接受了兩個指標做為其引數，並在其計算過程中，透過*間接存取運算子進行相關的計算，所以其計算的結果過直接存放在這兩個指標所指向的記憶體位址內。如此一來，就可以做到讓一個函式可以傳回(事實上並沒有傳回的動作了)一個以上的數值。

要注意的是，函式的原型可以使用下列兩者之一進行宣告：

```
void decompose(double val, long *int_part, double *frac_part);
void decompose(double, long *, double *);
```

下面的程式碼展示了呼叫decompose的方法：

```
double d = 3.1415;
int i;
double f;

decompose(d, &i, &f);
```

11.6.2 以指標做為函式傳回值

除了可以使用指標做為函式的引數，我們也可以使用指標做為函式的傳回值，請參考以下的範例：

```
int *max(int *a, int *b)
{
   if( *a > *b)
      return a;
   else
      return b;
```

```
}
```

在呼叫時，要注意必須要以一個整數的指標來接收函式的傳回值:

```
int x, y;
int *p;

p = max( &x, &y);
```

或者，以下列的方法取回函式執行後傳回的指標，並透過間接存取，直接存取該指標所指向的位址裡的值。請參考下面的程式：

```
#include <iostream>

int *max(int *a, int *b)
{
  if(*a > *b)
    return a;
  else
    return b;
}

int main()
{
  int x=5, y=10;
  int *p;

  p = max(&x, &y);
  cout << "The maximum value is " << *p << endl;

  *max(&x, &y)=100;

  cout << "The values of x and y are " << x << " and " << y << endl;
  return 0;
}
```

11.7 傳值呼叫與傳址呼叫

本節將為讀者說明兩個與函式呼叫相關的名詞，分別是傳值呼叫(Call By Value)與傳址呼叫(Call By Address)。

- 傳值呼叫(Call By Value)是指呼叫者函式(Caller Function)在進行函式呼叫時，將數值做為引數傳遞給被呼叫函式(Callee Function)使用。

- 傳址呼叫(Call By Address)是指呼叫者函式(Caller Function)在進行函式呼叫時，將記憶體位址做為引數傳遞給被呼叫函式(Callee Function)使用。

我們把呼叫函式的函式稱為呼叫者函式(Caller Function)或簡稱為呼叫者)，並且把被呼叫的函式稱為被呼叫函式(Callee Function)或簡稱為被呼叫者)。當呼叫者函式發起了一個對被呼叫函式的呼叫時，呼叫者函式就會被暫停執行，轉而執行被呼叫函式；等到被呼叫函式結束它的工作後，才會返回到當初發起呼叫的呼叫者函式繼續它未完成的工作，與此同時，如果被呼叫函式是以return敘述結束的話，也會有數值傳回給呼叫者函式使用。

下面我們使用兩個程式分別就傳值呼叫與傳址呼叫做一示範：

```
void swap(int x, int y)
{
   int temp=x;
   x=y;
   y=temp;
}

int main()
{
   int a, b;
   cin >> a >> b;
   swap(a,b);
   cout << "a=" << a << " b=" << b << endl;
}
```

上面這個程式取得使用者輸入的兩個整數a與b後，發起一個函式呼叫去呼叫swap()函式，並將a與b的數值做為引數傳遞給swap()函式裡的參數x與y。swap()函式在執行時，會將這兩個數值進行交換。

但此程式的執行結果與我們預期的並不一樣，其執行後的輸出的a與b的值並沒有交換成功。這是因為當我們呼叫swap()時，所傳入的是a與b的數值，雖然在swap()中將兩者做了交換，但回到main()時，在swap()內交換好的a與b的值，已經不存在。其原因在於這些變數都是屬於所謂的區域變數(Local Variable)，離開函式回到main()時，那些在函式內的變數就不再存在。

如果真的要在swap()函式內完成在main()內的兩個變數的值的交換，可以參考以下使用傳址呼(Call By Address)的程式：

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

void swap(int *x, int *y)
{
   int temp=*x;
   *x=*y;
```

```
   *y=temp;
}

int main()
{
   int a, b;
   cin >> a >> b;
   swap(&a,&b);
   cout << "a=" << a << " b=" << b << endl;
}
```

此程式在第15行針對swap()函式進行函式呼叫時，所傳遞的引數是變數a與b的記憶體位置；當這兩個記憶體位址傳送給swap()函式後，將會由兩個int整數指標x與y來接收(如第4行所定義的函式參數)。由於採用了傳址呼叫的方式，swap()函式後續在進行兩個數值交換時，它其實是透過指標x與y去間接存取實際上宣告在main()函式裡的a與b變數。換句話說，在swap()函式裡所交換的是兩個記憶體位址裡面的數值 — 而這兩個記憶體位址正是位於main()函式裡的變數a與b，等到swap()函式執行結束返回main()後，這兩個變數a與b的數值，早就已經在swap()函式裡完成了互相交換。

11.8 參考

在C++語言裡的變數可以直接用其名稱來存取，或是使用指標來對它所配置到的記憶體空間進行間接的存取。除此之外，C++語言還提供了一種新的方式來存取變數 — 參考(Reference)，所謂的參考(Reference)可以視為是變數的「別名(Alias)」，讓變數除了原本的名稱之外，又多了新的名稱可以對它進行存取。參考在使用前也必須先加以宣告，其宣告語法如下：

參考宣告敘述語法定義

資料型態 &參考名稱 = 變數名稱;

在上述語法中要特別注意的是，參考在宣告時就必須完成初始值給定，所以在上述語法中的初始值部份並不是可選擇性的，而是必須的。所謂的參考的初始值給定，其實就是要定義其參考的對象(也就是要“講明白”它是要做為誰的別名）。還要注意的是，一但宣告後，就不可以再改變其參考的對象。

請參考下面的程式碼：

```
double a = 3.1415927;
double &b = a;
```

上述程式宣告了一個名為a的double型態變數，以及一個名為b的參考，並透過初始值給定將b做為變數a的別名 — 換句話說，現在不論是使用變數名稱a，或是參考名稱b，都是對同一個記憶體空間裡的數值進行存取。請參考以下的範例程式：

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

int main ()
{

   double a;
   double &b = a;         // b is a

   a = 2.8;
   cout << "a=" << a << " b=" << b << endl;
   b = 3.5;
   cout << "a=" << a << " b=" << b << endl;
}
```

在上述範例中□b被宣告為a的別名，你會發現不論是對a或對b做操作，結果兩者都會有一樣的內容。

參考變數也可以用在函式的引數宣告，例如下面的程式碼：

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

void change (double &r, double s)
{
   r = 100;
   s = 200;
}

int main ()
{
   double k, m;

   k = 3;
   m = 4;

   change (k, m);
   cout << k << ", " << m << endl;
}
```

這種將參考傳入函式做為引數的方法，稱為「傳參考呼叫(Call by Reference)□□再讓我們看一個更為複雜的例子：

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

double &biggest (double &r, double &s)
{
```

```
   if (r > s) return r;
   else       return s;
}

int main ()
{
   double k = 3;
   double m = 7;

   cout << "k: " << k << endl;       // Displays  3
   cout << "m: " << m << endl;       // Displays  7
   cout << endl;

   biggest (k, m) = 10;

   cout << "k: " << k << endl;       // Displays  3
   cout << "m: " << m << endl;       // Displays 10
   cout << endl;

   biggest (k, m) ++;

   cout << "k: " << k << endl;       // Displays  3
   cout << "m: " << m << endl;       // Displays 11
   cout << endl;

   return 0;
}
```

這個程式蠻有趣的吧！