**Ch4. 陣列**

課程大綱

A. 指標複習：瞭解指標是操作陣列的前提

B. 陣列與記憶體位置

C. 多維陣列

D. 陣列複雜度分析與使用時機

**第一節：指標複習**

A. 指標是一種變數

a. 變數負責儲存資料，指標儲存的資料是「記憶體位置」

b. 指標是一種變數，因此變數需要符合的規範指標都要遵守

B. 資料沒有名稱時，透過指標儲存的記憶體位置來使用資料

a. 動態記憶體配置

b. 函式間傳遞資料

c. 資料的存放與管理（鏈結串列中大量使用）

1. 使用指標的方式

A. 宣告：像變數一樣，在指標使用前必須先取一個「變數名稱」

B. 取址：取出目標變數「在記憶體中的位置」

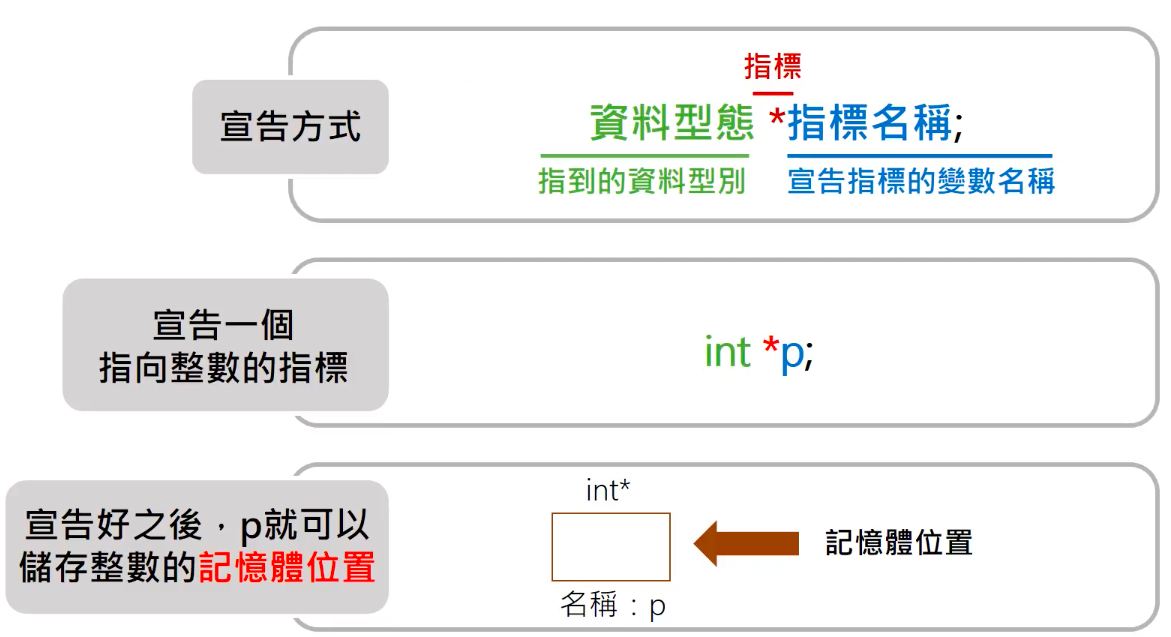
C. 取值：透過該記憶體位置存取「資料的值」



資料型別是指「如何解讀記憶體中以 0 和 1 形式存放的資料」。

取址：問電腦某個變數存在哪個位置

取值：問電腦某個記憶體位置內容為何

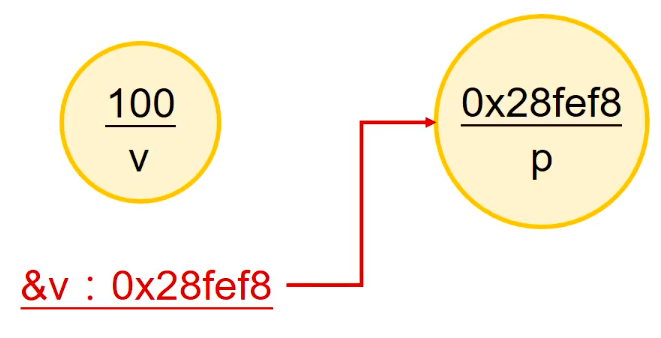
2. 指標的宣告

3. 取出變數的記憶體位置

取址運算子 & 可以用來取出變數的記憶體位置：

|  |  |
| --- | --- |
| 使用取址運算子 & | |
| 1  2  3  4 | int \*p;  int v = 100;  p = &v;  cout << p; |

要把 v 的記憶體位置取出來賦值給 p（p 是一個整數指標，可以存放一個整數的記憶體位置），可以使用「&v」的寫法。假設取得的位置是 0x28fef8，且把這個位置賦值給 p，p 的內容就是 0x28fef8。

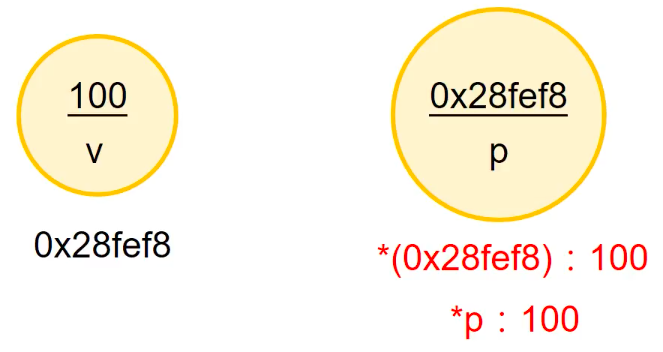


通常記憶體位置沒有什麼意義，這就像「知道某人的地址」是為了方便我們「找到這個人」而已。每次執行時，同一個變數被指派到的位址可能不同，因此比起位置，更重要的是該處存放的「值」。

4. 取得指標所指位置上存放的值

取值運算子 \* 可以用來取得指標所指位置上存放的值：

|  |  |
| --- | --- |
| 使用取值運算子 \* | |
| 1  2  3  4 | int \*p  int v = 100;  p = &v;  cout << \*p; |



在 p 前面加上 \* 運算子，相當於要「取出 0x28fef8（p 中的值）這個位置上存放的資料」。

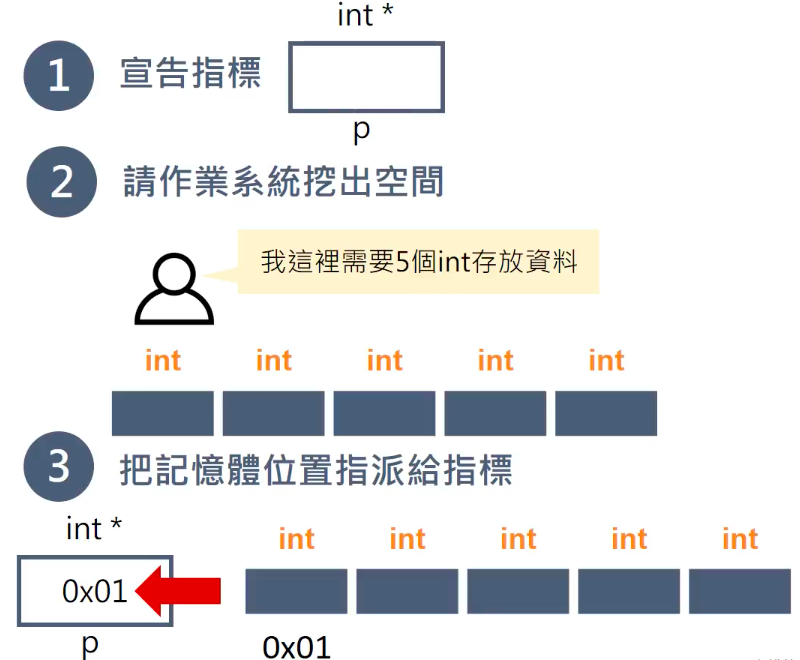
5. 動態記憶體配置

動態記憶體配置的使用時機，是在程式執行時臨時指派空間以存放資料。

臨時需要空間儲存資料時，可以向作業系統索取一塊記憶體。所有撰寫的程式都在作業系統上運行，作業系統就像媽媽一樣有很多個小孩要餵，會隨時監看哪個程序需要記憶體空間並即時配置。

通常在C 語言中，是由使用者告知需要多少記憶體空間後才進行配置；確認不再需要存取一塊記憶體空間上的資料後，才進行釋放；相對的，如果撰寫的是 C++，且支援 STL 的話，則盡量改用「vector 向量」。

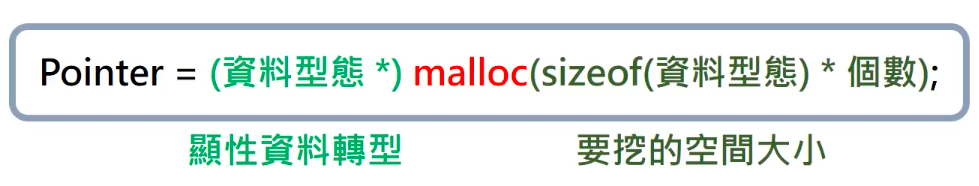
6. 使用動態記憶體配置



使用動態記憶體配置的流程

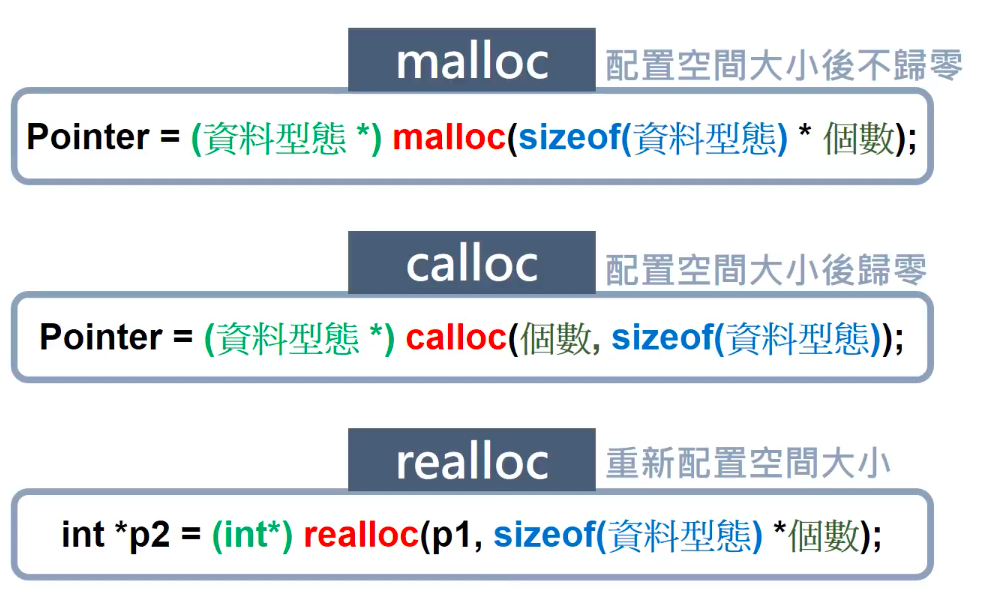
A. #include <stdlib.h>

B. 宣告出指標備用

C. 使用 malloc() 使作業系統挖出一塊空間：空間大小使用乘法形式

「sizeof(資料型態) \* 個數」，方便之後修改

D. 結束後用 free(指標名稱) 釋放記憶體

7. 配置動態記憶體的三種函式

注意三個函式需要傳入的引數都不同。

8. 記憶體洩漏 Memory Leakage

用動態記憶體配置開出陣列時，指標是存取該記憶體位置唯一的媒介，一旦遺失指標，該位置就再也無法存取，造成記憶體洩漏。

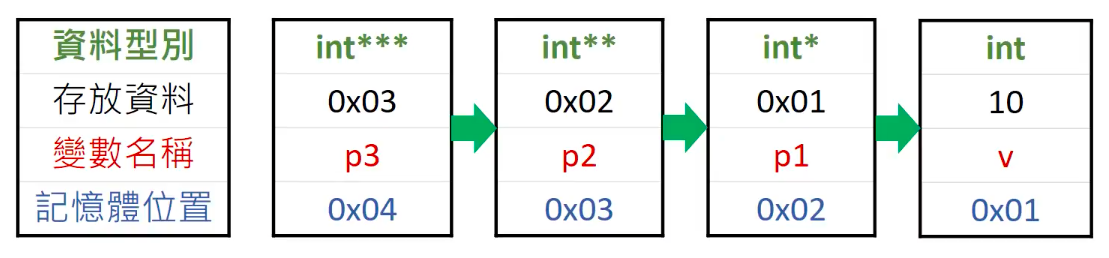
上圖中，p 被第二次 malloc 配置的記憶體位置賦值後，第一次配置的記憶體位置就無法再得知，該位置也不能被使用來存放其它資料（因為尚未釋放）。

這就像我們耳熟能詳的「桃花源」的故事：漁夫離開桃花源時，在門口做了記號（指標），這是漁夫回到桃花源的唯一方式，一旦遺失了這個記號，漁夫就再也無法回到這個地方（開出的記憶體位置）。

9. 指向指標的指標

指標的本質是一個「變數」，既然是變數，就會有自己的記憶體位置與資料。

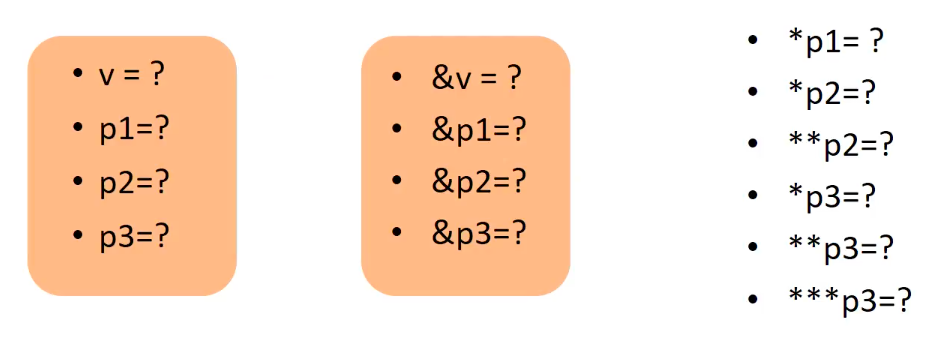
既然如此，也可以使用一個指標去指向「另一個指標本身的記憶體位置」。



上圖中，p1 指向 v 這個整數變數的位置（0x01），因此 p1 是一個「整數指標」；p2 又指向 p1 這個指標存放在記憶體中的位置（0x02），因此 p2 是一個「雙重指標」，也是「指向整數指標的指標」。

p2 的資料型別 int\*\* 可以被看成是 int\* 後面再加上 \*，既然在 int 後面加上 \* 可以代表一個 int 的指標，int\*\* 代表的就是一個指向某個 int\* 型態變數的指標。

10. 試著回答下列問題，加深對指標的理解：



|  |  |  |
| --- | --- | --- |
| 解答 | | |
| v = 10  p1 = 0x01  p2 = 0x02  p3 = 0x03 | &v = 0x01  &p1 = 0x02  &p2 = 0x03  &p3 = 0x04 | \*p1 = 10  \*p2 = 0x01  \*\*p2 = 10  \*p3 = 0x02  \*\*p3 = 0x01  \*\*\*p3 = 10 |

11. 指標的用途

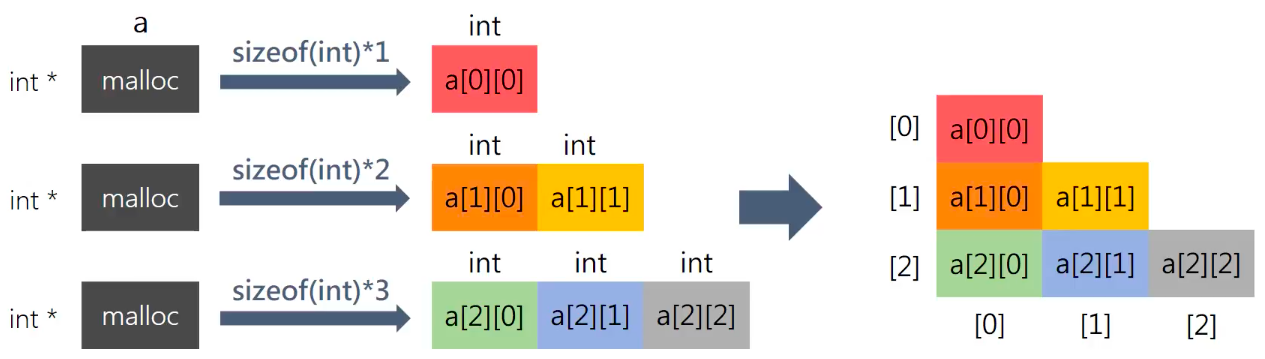
指標的一個常見用途是「開出不定長度的陣列」，一般在開出陣列時，可能常開出一個方方正正的陣列，每個 row（列）的 column（行）數都相同。

但這不一定是最好的方式，就像要使用陣列儲存學校學生的資料時，如果只按照「全校有 30 班」、「每班最多有 42 個人」來開出一個 30 x 42 的長方形陣列，就可能會有浪費空間的情形發生：第一班可能只有 28 人、第二班只有 34 人、…，這時利用指標來開出不定長度（不是長方形）的陣列會更理想。

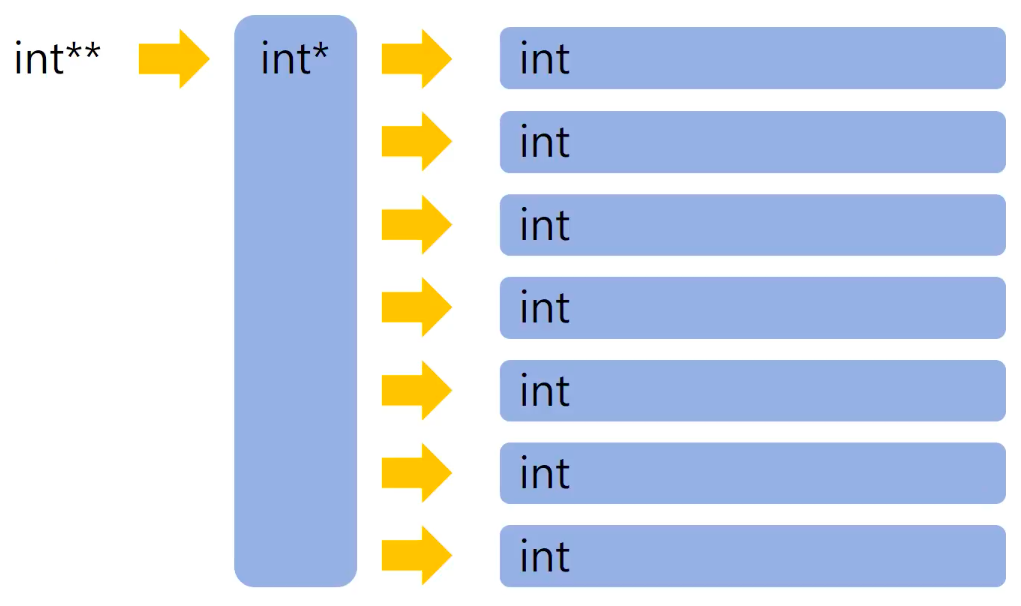
首先開出一個「整數指標」的陣列 a：

int \*a[3];

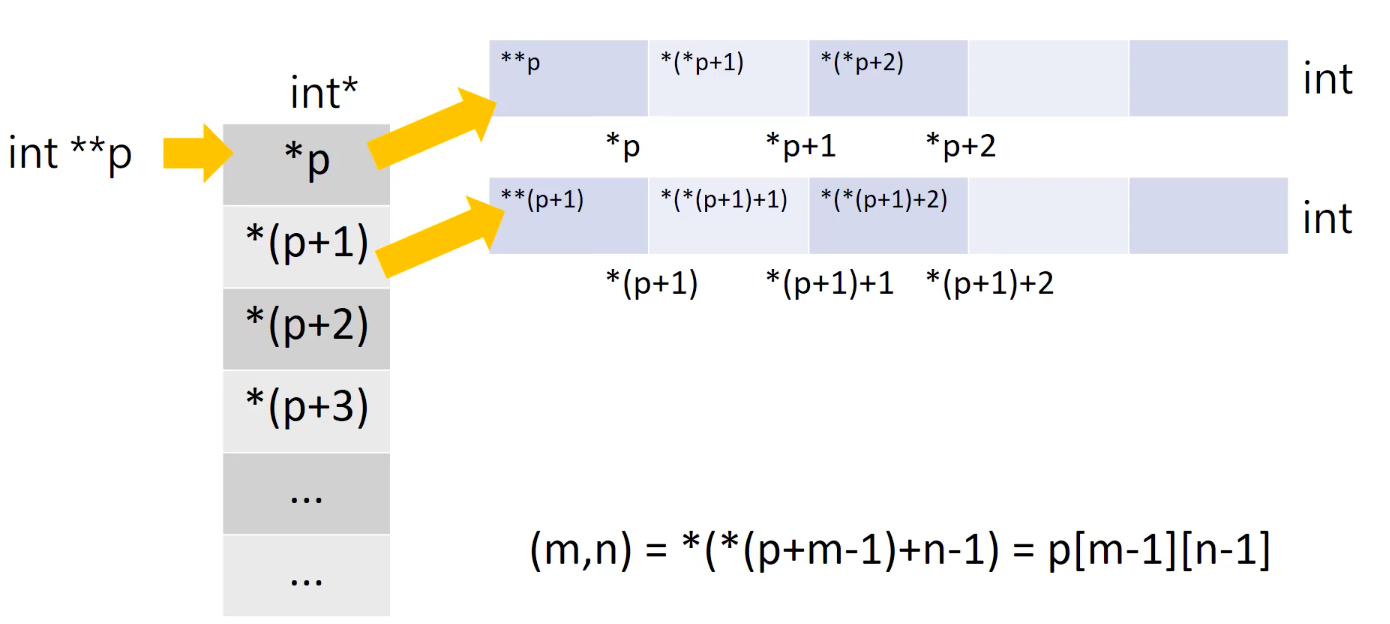
注意上面的 a 是一個陣列，這個陣列裡可以存放三個「整數指標」（而不是整數），這樣一來，每個指標又可以指到另一個陣列，且不需要都一樣長，比如第一個整數指標指向一個整數 int、第二個整數指標指向兩個整數 int、第三個整數指標指向三個整數 int，這樣就形成一個下圖中「三角形」或「梯形」的二維陣列。



12. 雙重指標的用途

雙重指標的用途，是開出一個「動態的二維陣列」。

開出一個「動態的一維陣列」時，可以使用 malloc，但是要產生一個row 和 column 的個數都由使用者指定的「動態二維陣列」，就要先開出一個存放整數指標的陣列，每個整數指標都分別再指向一個整數陣列。

「整數指標的陣列」是利用 malloc 來開出，當中每個整數指標各自指到的陣列一樣使用 malloc 建立，因此指向這個「動態二維陣列」的指標 int \*\*p 是一個「整數雙重指標 int\*\*」。

上圖中，如果要取用第 (m,n) 筆資料（m 是指整數指標 int\* 陣列中的第幾筆，n 是該整數指標指到的整數陣列中的第幾筆），可以使用 \*(\*(p+m-1)+n-1)，也可以用 p[m-1][n-1] 來表示，兩個寫法等價。

**第二節：陣列與記憶體位置**

A. 陣列名稱：表示陣列所在記憶體空間的起始位址

B. 索引值：代表該元素距離陣列開頭有多遠

索引值：距離開頭有多遠，比如 score[2] 代表由 score[0] 開始往後數兩筆。

score：開頭的記憶體位置，即 score[0]。

1. 索引值的意義

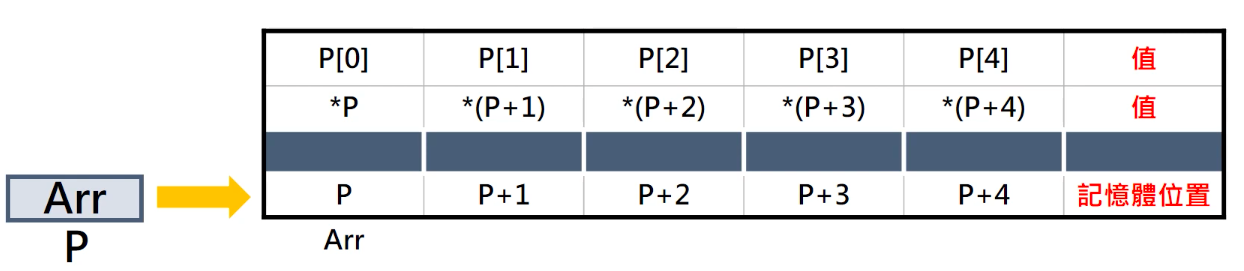
|  |  |
| --- | --- |
| 「陣列名稱」與「陣列開頭資料的記憶體位置」 | |
| 1  2  3  4  5  6  7  8  9 | #include <iostream>  using namespace std;  int main(){  int score[5]; // 開出一個長度為 5 的整數陣列 score  cout << score << endl; // 印出「score」的值  cout << score[0] << endl; // 印出「score[0]」的值  return 0;  } |
| 執行結果 | |
| 0x6dfedc // 讀者測試時值不一定與此相同，但兩行會印出相同資料  0x6dfedc | |

從上面的執行結果可以發現，陣列名稱 score 指的就是陣列中第一筆資料 score[0] 的「位置」，兩者的值相同。

另外，索引值指的是「距離陣列開頭的資料」的距離，因此最大值只會到「陣列長度 - 1」。

int Arr[5]; // 宣告一個整數陣列 Arr，長度是 5

int \*p = Arr; // 宣告一個整數指標，指向整數陣列 Arr（開頭資料的位置）



p 指到的是第一筆資料的記憶體位置，p+1 指到的是第二筆資料的記憶體位置、p+2 是第三筆資料的記憶體位置、…，這樣一來，就可以使用一個 for 迴圈來印出陣列中的每一筆資料。

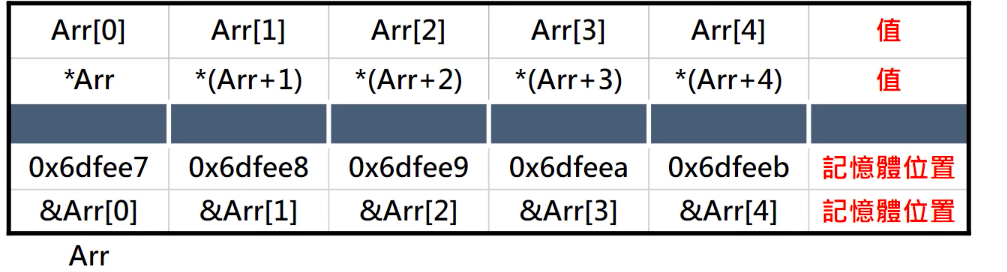
記憶體位置前面加上 \* 指的是取出該位置的資料，因此 \*p 是第一筆資料的內容、\*(p+1) 是第二筆資料的內容、\*(p+2) 是第三筆資料，依此類推。

|  |  |
| --- | --- |
| 印出陣列的內容 | |
| 1  2  3  4  5  6 | int Arr[5]={1,2,3,4,5};  int \*p = Arr;  for(int i=0;i<5,i++)  {  cout << \*(p+i) << " "; // \*(p+0), \*(p+1), \*(p+2), \*(p+3), \*(p+4)  } |

把「cout << \*(p+i) << " ";」寫成「cout << p[i] << " ";」，也就是用 p[i] 來表示 \*(p+i)，執行結果會相同。

2. 陣列元素的記憶體位置

(1) 布林陣列的記憶體位置



宣告一個 boolean 陣列，裡面有 5 個 boolean 值，因為 boolean 值的大小是一個 byte，因此5 個 boolean 變數會被放在相鄰（連續）的位置。

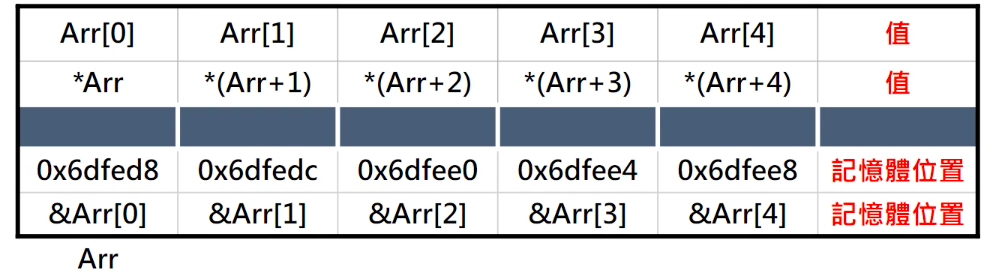
把每個元素 Arr[i] 的前面加上一個 &，就可以依序取出資料的位置檢視。

|  |  |
| --- | --- |
| 印出 boolean 陣列元素的記憶體位置 | |
| 1  2  3  4  5 | bool Arr[5] = {0,0,1,1,1};  for (int i=0 ; i<5 ; i++)  {  cout << &Arr[i] << " ";  } |
| 執行結果 | |
| 0x6dfee7 0x6dfee8 0x6dfee9 0x6dfeea 0x6dfeeb | |

從上面的輸出結果中可以看出每個元素的位置確實相鄰。

(2) 整數陣列的記憶體位置

改為宣告一個 int 陣列，再觀察一次輸出的資料位置。



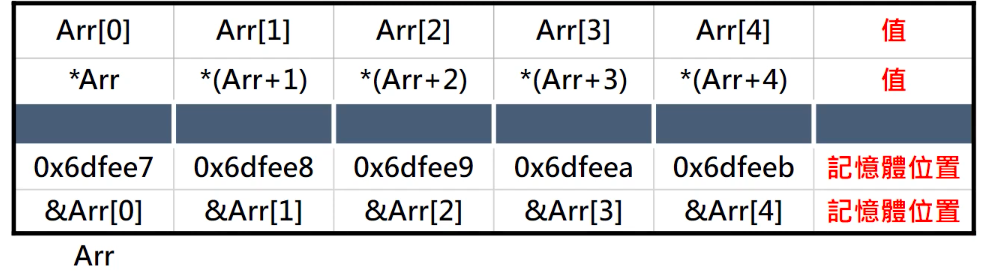
|  |  |
| --- | --- |
| 印出 int 陣列元素的記憶體位置 | |
| 1  2  3  4  5 | int Arr[5] = {0,0,1,1,1};  for (int i=0 ; i<5 ; i++)  {  cout << &Arr[i] << " ";  } |
| 執行結果 | |
| 0x6dfed8 0x6dfedc 0x6dfee0 0x6dfee4 0x6dfee8 | |

16 進位中，8 -> 9 -> a -> b -> c，8 到 c 間差了 4 個數。

同樣的，c -> d -> e -> f -> 0，c 到 0 間也差了 4 個數。

int 間分別差距 4 個 byte，從執行結果中，可以看出這個 5 個 int 變數也被放在相鄰（連續）的位置。

(3) 字元陣列的記憶體位置

再改為宣告字元 char 的陣列：

|  |  |
| --- | --- |
| 印出 char 陣列元素的記憶體位置（?） | |
| 1  2  3  4  5 | char Arr[5]={'1','2','3','4','5'};  for (int i=0 ; i<5 ; i++)  {  cout << &Arr[i] << " ";  } |
| 執行結果 | |
| 12345 2345 345 45 5 | |

上面的執行結果中，印出來的「不是」記憶體位置。為何會有這樣的結果呢？

這是因為 << 運算子在輸出「字元」型態資料的記憶體位置時，原先的功能被（寫編譯器的開發者）覆寫了，不像其他型態的資料會輸出記憶體位置，遇到 「cout << &（字元）」時，則會把「這個記憶體位置後的字串」全部印出來。

為了避免上面的狀況發生，我們可以將字元的記憶體位址轉換成 void 指標，也就是不帶任何資料型別的指標（顯性資料轉型）。

cout << (void\*) &Arr[i] << " ";

修改之後，就可以順利印出記憶體位置了。

2. Sizeof

另外，可以利用 sizeof 函式，直接得知陣列在記憶體中所佔的大小。

|  |  |
| --- | --- |
| 測試 sizeof | |
| 1  2  3  4  5  6  7  8  9  10  11 | bool Arr\_Bool[5]={0,0,1,1,1};  char Arr\_Char[5]={'1','2','3','4','5'};  int Arr\_Int[5]={1,2,3,4,5};  float Arr\_Float[5]={1.5,2.5,3.5,4.5,5.5};  double Arr\_Double[5]={1.5,2.5,3.5,4.5,5.5};  cout << "Size of bool array:\t" << sizeof(Arr\_Bool) << endl;  cout << "Size of char array:\t" << sizeof(Arr\_Char) << endl;  cout << "Size of int array:\t" << sizeof(Arr\_Int) << endl;  cout << "Size of float array:\t" << sizeof(Arr\_Float) << endl;  cout << "Size of double array:\t" << sizeof(Arr\_Double) << endl; |
| 執行結果 (單位：byte) | |
| Size of bool array: 5 // 一個 bool 佔 1 個 byte，5 個共佔 5 個 byte  Size of char array: 5  Size of int array: 20 // 一個 int 佔 4 個 byte，5 個共佔 20 個 byte  Size of float array: 20  Size of double array: 40 | |

若變數大小為 S 個位元組，陣列的起始位置為 P，則陣列中

A. 第 i 個索引值，或第 i+1 個元素的記憶體位置：

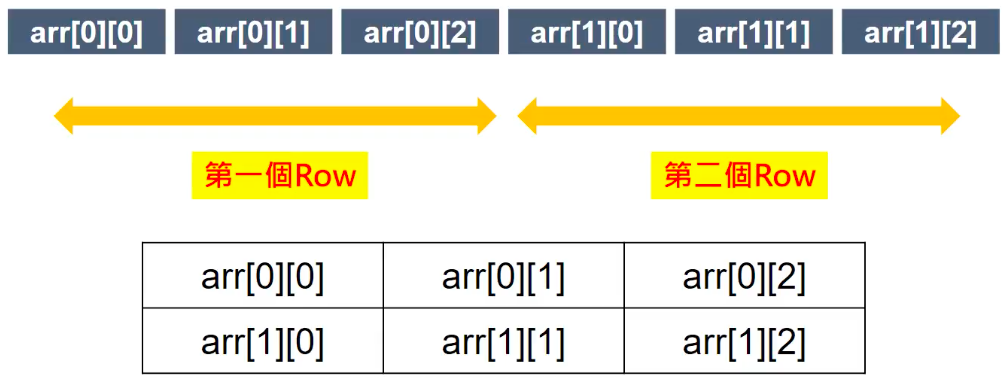
B. 陣列中的元素個數為 len，則陣列的總長度為：



**第三節：多維陣列的記憶體位置**

目前為止，我們看到的都是一維陣列元素的記憶體分佈。接下來，我們要來檢視多維陣列的情形。

1. 二維陣列的記憶體位置



假設使用下面的程式碼宣告出一個二維陣列 arr：

int arr[2][3];

這個二維陣列有兩個 row，三個 column，對於使用者而言，就像上圖中下方「兩個橫列」「三個直行」的一個長方形。

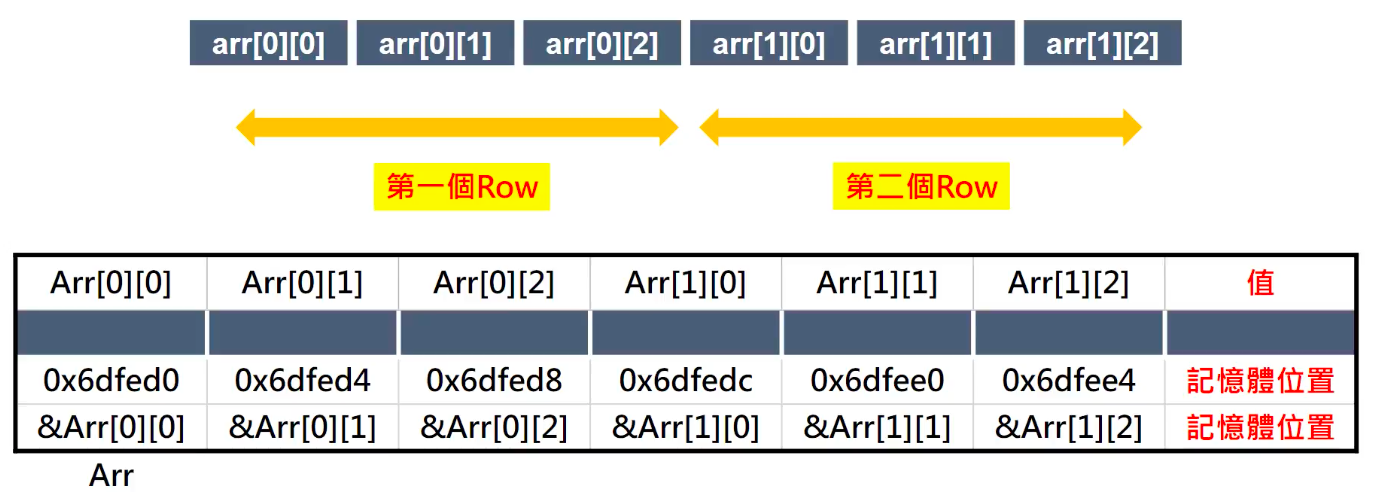
但是電腦並沒有辦法理解這種「多維」的概念。人類之所以能夠理解二維、三維，是因為本身活在三維的立體空間中；電腦則只能進行「線性」的運算。

因為電腦並沒有多維的概念，所以對記憶體空間也是採取「線性」的觀點，只是經由「換算」來方便人類使用。像上面兩列三行的長方形中總共有六筆資料，電腦會在記憶體中先存放 arr[0][0]、arr[0][1]、arr[0][2] 三筆資料，放完後，緊接著再存放 arr[1][0]、arr[1][1]、arr[1][2] 三筆資料。

這也就是說，電腦仍然是線性處理多維陣列所需的記憶體，並沒有二維或三維的概念，只是經由電腦的加減乘除運算，換算成人類容易理解的「多維」形式。

|  |  |
| --- | --- |
| 印出二維陣列的記憶體位置 | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | #include <iostream>  using namespace std;  int main(){  int arr[2][3]={1,2,3,4,5,6}; // 指定 arr 陣列的初始值  for(int i=0;i<2;i++){  for(int j=0;j<3;j++){  cout << &arr[i][j] << " "; // 印出資料的「記憶體位置」  }  }  cout << endl;  return 0;  } |
| 執行結果 | |
| 0x6dfed0 0x6dfed4 0x6dfed8 0x6dfedc 0x6dfee0 0x6dfee4 //各相差 4 個 Byte | |

為什麼第 5 行指定 arr 陣列的值時，可以只用一層大括號？這其實代表電腦是把多維陣列中的所有元素都用線性方式存放，所謂「多維」只是換算後方便人類理解的形式而已。

從上面的執行結果會發現，這六筆資料的記憶體位置各相差 4 個 Byte，是連續存放的。

除了二維陣列，三維、四維以至於更高維的陣列中，所有的資料也都是線性存放在相鄰的位置。

2. Row first 與 Column first

A. Row first：第一個 row 放完再放下一個 row（較常使用）

B. Column first：先放一個 column 再放下一個 column

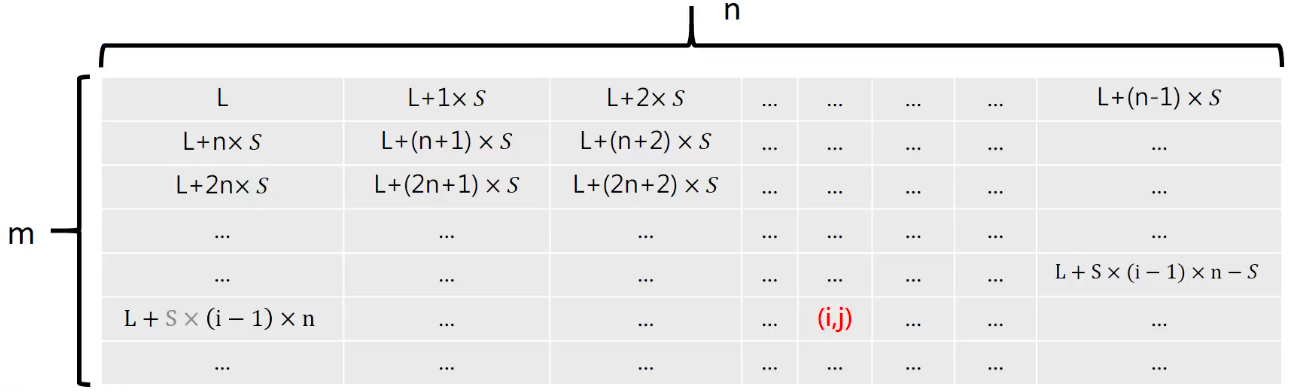
(1) Row-First

Row-First 的存放方法中，放完第一筆資料後，會存放同一個 row 中的下一筆資料（下圖中是往右邊移動），一直到放完整個 row 後（到達第一個橫列的最右邊），再開始放下一個橫列的內容。

若變數大小為 S，陣列起始位置為 L，陣列長寬為 m x n，則該二維陣列中第 (i , j) 個元素（從 1 開始計數）的記憶體位置：

第一筆資料如果存放在 L 這個位置，第二筆資料就放在 L + 1 x S、第三筆資料放在 L + 2 x S、…。開始放第二個橫列的第一筆資料時，因為已經存放 n 筆資料，所以它的位置在 L + n x S，以此類推。

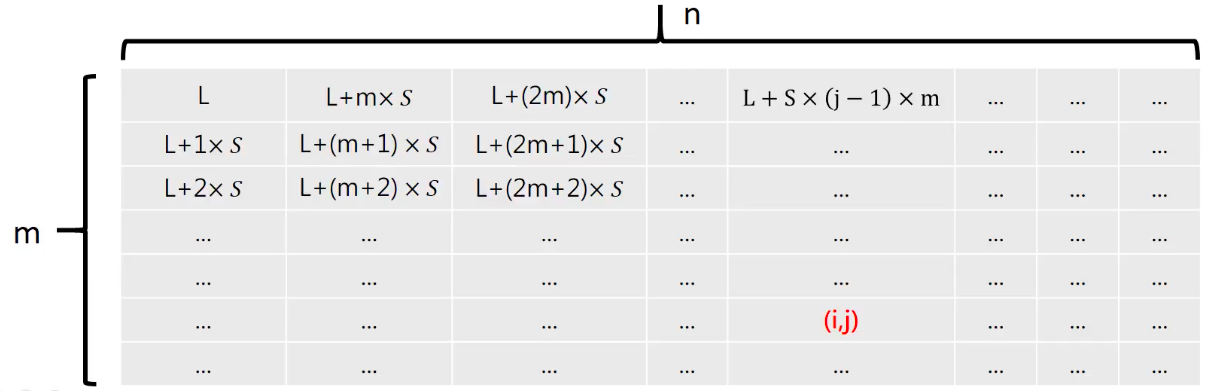
可以拆解成三個部分來看。第一個部分 L 是整個二維陣列的開頭位置，加上 後，是第 i 行開頭元素的記憶體位置，再從 i 行開頭元素 (i , 1) 開始往右移動 j-1 筆資料到達 (i , j)，所以要加上第三個部分 。



(2) Column-First

Column-First 的存放方法中，放完第一筆資料後，會存放同一個 column 中的下一筆資料（下圖中是往下方移動），一直到放完整個 column 後（到達第一個直行的最下方），再開始放下一個直行的內容。

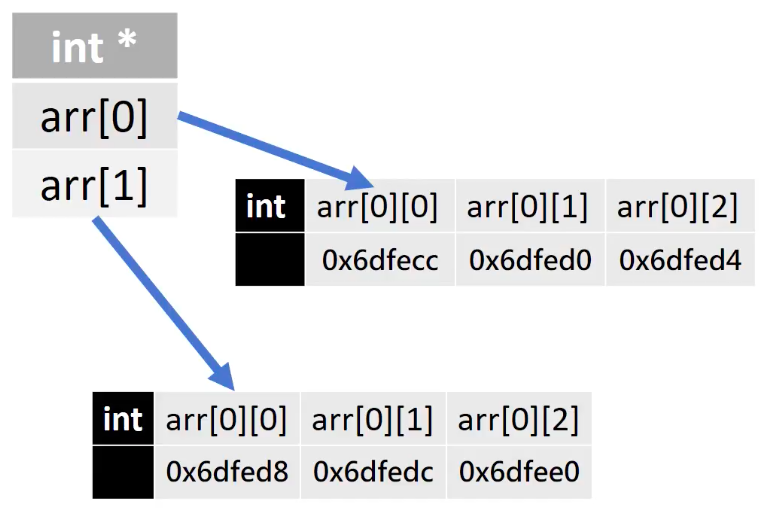
若變數大小為 S，陣列起始位置為 L，陣列長寬為 m x n，則該二維陣列中第 (i , j)個元素（從 1 開始計數）的記憶體位置：



3. 列的開頭元素

把一個二維陣列（row-first）中每個 row 的第一筆資料取出來，看看它的內容是什麼：

|  |  |
| --- | --- |
| 印出每個 row 的第一筆資料 | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | int arr[2][3]={1,2,3,4,5,6};  cout << "arr = " << arr << endl;  for(int i=0;i<2;i++){  cout << arr[i] << " ";  }  cout << endl;  for(int i=0;i<2;i++){  for(int j=0;j<3;j++){  cout << arr[i][j] << " ";  }  }  cout << endl; |
| 執行結果 | |
| arr = 0x6dfecc  0x6dfecc 0x6dfed8 // 與下一行中的第一筆和第四筆相同  0x6dfecc 0x6dfed0 0x6dfed4 0x6dfed8 0x6dfedc 0x6dfee0 | |

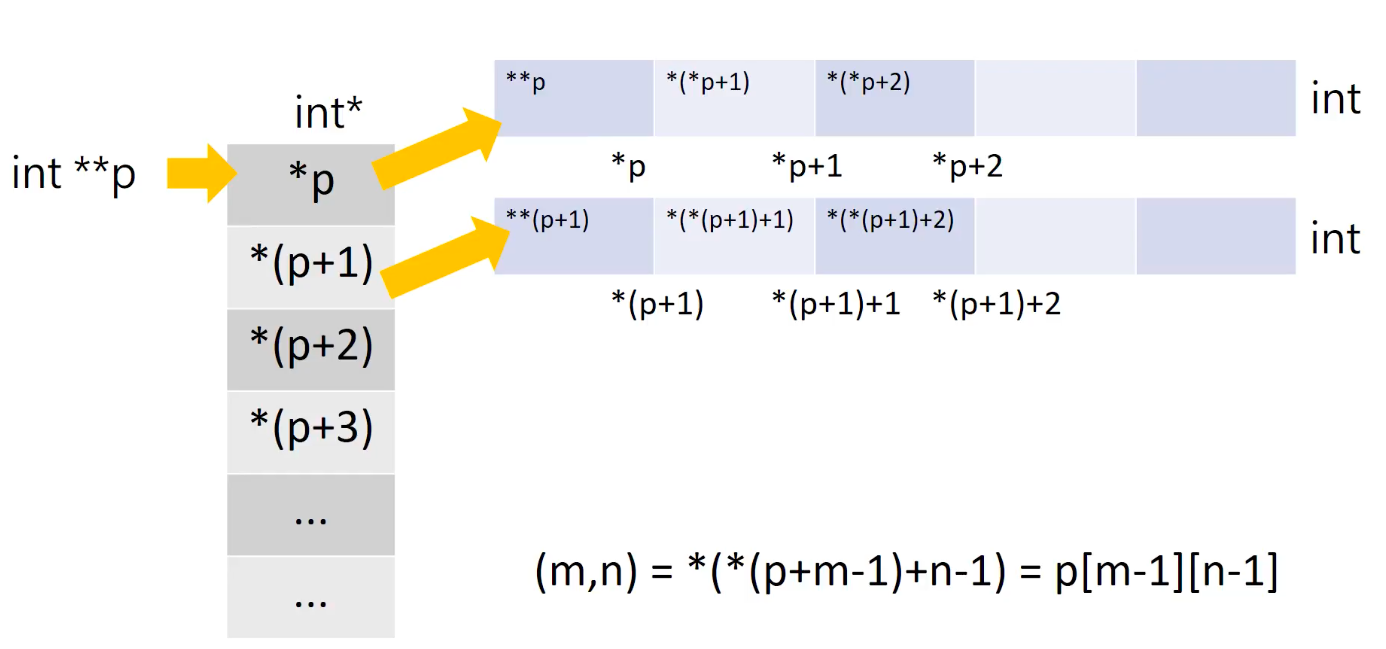


從上面的執行結果可以發現，通常二維陣列取出資料會用兩個中括號，但只用一個中括號時仍然可以取出資料。arr[i] 是一個整數指標，代表的是「第 i 列最開頭資料」的記憶體位置。

4. 用單一迴圈遍歷二維陣列

(1) 使用雙重迴圈印出陣列

\*(\*(arr+i)+j) 可以用來取出二維陣列中第 (i , j) 筆資料的內容，相當於 arr[i][j]，其中的 \*(arr+i) 相當於要取出第 i 個橫列的第一筆資料，加上 j 後再取一次值，代表要取出這個橫列的第 j 筆資料。



|  |  |
| --- | --- |
| 用雙重 for 迴圈遍歷二維陣列 | |
| 1  2  3  4  5  6  7  8  9 | int arr[2][3]={1,2,3,4,5,6};  cout << "arr = " << arr << endl;  for(int i=0;i<2;i++){  for(int j=0;j<3;i++){  cout << \*(\*(arr+i)+j) << " ";  }  }  \*(\*(arr+i)+j) = arr[i][j] |
| 執行結果 | |
| 1 2 3 4 5 6 | |

\*(p+m-1)：取出第 m 個整數指標，是那個橫列的第一個位置

\*(+n-1)：在該橫列中，往後取到第 n 個位置

(2) 使用單一迴圈印出陣列

因為一個多維陣列中，所有的元素也都是存在記憶體的連續位置，所以其實可以只用一個 for 迴圈取出所有資料。

|  |  |
| --- | --- |
| 用單一 for 迴圈遍歷二維陣列 | |
| 1  2  3  4  5  6  7 | int arr[2][3]={1,2,3,4,5,6};  cout << "arr = " << arr << endl;  for(int i=0;i<6;i++){  cout << \*(arr+i) << " ";  }  cout << endl; |
| 執行結果 | |
| arr = 0x6dfed4  0x6dfed4 0x6dfee0 0x6dfeec 0x6dfef8 0x6dfe04 0x6dfe10  // 註：為何每個輸出跳 12 個 byte? 3 x 4byte (int) | |

但是上面我們卻發現執行結果是印出許多記憶體位置，而非二維陣列的元素，

這是因為 arr 是一個「二維陣列」，使得 arr 的型態是一個「雙重指標」，所以要把 arr 先轉型成整數指標後，賦值給另一個指標 p，再轉用 p 去印出所有資料。

|  |  |
| --- | --- |
| 用單一 for 迴圈遍歷二維陣列（修正版） | |
| 1  2  3  4  5  6 | int arr[2][3]={1,2,3,4,5,6};  int \*p = (int\*)arr // arr 是雙重指標 int\*\*，轉型後變成整數指標int\* p  for(int i=0;i<6;i++){  cout << \*(p+i) << " ";  } |
| 執行結果 | |
| 1 2 3 4 5 6 | |

(3) 比較兩種遍歷方法的效能

為什麼要把多個 for 迴圈的程式碼轉為單一 for 迴圈？因為這樣可以提高效能、節省時間。

|  |  |
| --- | --- |
| 比較多重 for 迴圈與單一 for 迴圈的速度 | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | #include <iostream>  #include <time.h> // 計時使用  using namespace std;  int main(){  clock\_t s, f;  const int n = 600;  auto arr\_3d = new int[n][n][n]; // 宣告測試用的三維陣列，大小是  int counts = 0; // 600 x 600 x 600  // 用三層迴圈尋訪多維陣列元素  s = clock();  for(int i=0 ; i<n ; i++){  for(int j=0 ; j<n ; j++){  for(int k=0 ; k<n ; k++){  arr\_3d[i][j][k]=counts;  counts++;  }  }  }  f = clock();  cout << "Time consumed:" << (f-s)/(double)CLOCKS\_PER\_SEC << "s" << endl;  // 用單一迴圈尋訪多維陣列元素  s = clock();  counts = 0;  int \*p = (int\*) arr\_3d;  for(int i=0 ; i<n\*n\*n ; i++){ // 共 n\*n\*n 筆資料  \*p=counts;  counts++;  }  f = clock();  cout << "Time consumed:" << (f-s)/(double)CLOCKS\_PER\_SEC << "s" << endl;  } |
| 執行結果 | |
| Time consumed:1.064s  Time consumed:0.629s | |

三層 for 迴圈跑遍所有元素所花的時間，和單一 for 迴圈跑遍元素需要的時間相比，速度約相差 30% 至 40%。

然而非必要時，因為多層迴圈的可讀性較佳，仍應使用多層 for 迴圈。除非資料量非常大，或該操作比較底層，才考慮使用單一 for 迴圈撰寫。

5. 多維陣列記憶體位置的計算練習

若變數大小為 S，陣列起始位置為 L，陣列長寬高為 ，則該三維陣列中，第 (i , j , k) 個元素（從 1 開始計數）的記憶體位置為何（Row-First）?

請用 L、i、j、k、x、y、z、S 表示。

// i 是在第幾個 yz 平面上？ 第i - 1個

// 在第幾個列上？ 第j - 1個

// 還要再移動 k - 1 個位置

6. 宣告 3 x 3 x 3 陣列，並試著在「單一 for 迴圈」內把值初始化成 1 - 27

|  |  |
| --- | --- |
| 用單一 for 迴圈初始化三維陣列 | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | #include <iostream>  using namespace std;  int main(){  int data[3][3][3];  int \*p = (int\*) data; // 記得把三重指標 data 轉為整數指標  // 用單一 for 迴圈初始化  for(int i=0;i<27;i++){  \*(p+i) = i+1 // \*(p+i) 也可寫成 p[i]  }  // 用三層 for 迴圈印出元素（也可使用單一 for 迴圈）  for(int i=0;i<3;i++){  for(int j=0;i<3;j++){  for(int k=0;i<3;k++){  cout << data[i][j][k] << " ";  }  }  }  return 0;  } |
| 執行結果 | |
| 1 2 3 ... 27 | |

如果加上 & 運算子，改輸出記憶體位置，如

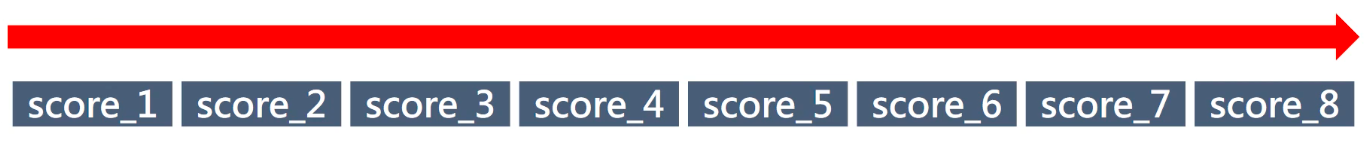
cout << &data[i][j][k] << " ";

執行後會發現 27 筆資料被放在連續的記憶體空間裡面。

**第四節：陣列複雜度與使用時機**

本章的最後，要介紹操作陣列的複雜度，以及什麼情況下應該使用陣列。

1. 陣列的效能分析



(1) 搜尋：O(N)

A. Upper bound：N

B. Average bound：

C. Lower bound：1

在陣列中進行搜尋時，最好的情況是該筆資料正好位在陣列開頭，從第一筆開始向後搜尋時，只需要檢查一筆資料即告完成；最壞的情況下，則是該筆資料正好位在陣列結尾，需要檢查完全部 N（陣列長度）筆資料才能得知；平均而言，需要搜尋 次。

在討論一種操作的複雜度時，在意的是最差時的狀況，因此進行陣列搜尋的複雜度為 O(N)。

(2) 新增與刪除元素

新增 / 刪除陣列第一個元素：把所有的資料往後移，O(N)

新增 / 刪除最後一個元素：新增 / 刪除後不需移動資料，O(1)

新增 / 刪除特定引數：需要移動該引數後所有資料，O(N)

如上所示，當常常需要對前面的資料做增修時，會耗費許多時間，這時就不適合使用陣列。

另外，走訪陣列中的所有資料也需要 O(N)。

(4) 陣列的記憶體需求

陣列是所有資料結構裡最節省空間的，因為只需要儲存資料本身，資料與資料間的次序關係由記憶體位址標示。

2. 陣列的優缺點

(1) 優點 Pros

A. 利用記憶體的連續位置來記錄次序資訊

B. 利用 index 即可在 O(1) 時間對 Array 的資料做存取

C. 是「最節省記憶體空間」的資料結構

(2) 缺點 Cons

A. 只能儲存順序、無法儲存其餘資料間的對應關係

B. 搜尋需要 O(N) 的時間

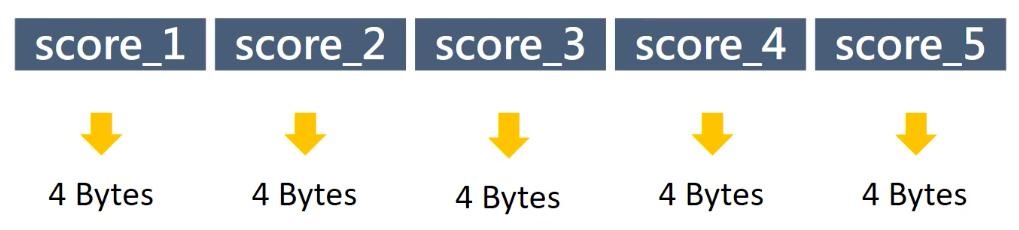
C. 新增 / 刪除特定位置，需要 O(N) 時間作搬移資料

D. 若長度改變，也會花費 O(N) 的時間遷移資料（vector 章節討論）

3. 陣列的使用時機

A. 希望透過快速存取資料

B. 資料數量是「固定」的，執行後不會改變長度

 C. 希望使用的記憶體空間越少越好

|  |  |
| --- | --- |
| 常見陣列資料操作的複雜度 | |
| access：O(1)  search：O(n)  delete：O(n)  insert：O(n) | 取出第 i 個資料  搜尋（未事先排序）  刪除特定元素  插入特定元素 |

預設的靜態陣列無法對長度做修改，每次插入都會有資料遺失，使用動態記憶體配置時才可以。

4. 陣列與編譯

A. 陣列長度必須在編譯時就決定好

a. C99 後支援 VLA(Variable Length Array)，陣列長度由變數指定

b. 但 VLA 在 C++11 中為非必要功能

C. 盡量讓陣列長度為定值（define、const）

B. 程式開始執行後無法改變陣列長度

5. 靜態陣列與動態陣列

A. 靜態陣列 Static array

a. 直接宣告出的陣列：int data[5];

b. 可用 sizeof 來取得陣列大小

B. 動態陣列 Dynamic array

a. 使用 malloc 或 new 開出的陣列

b. 無法使用 sizeof 得知陣列大小，需另外紀錄

動態陣列的好處是可以修改長度，靜態陣列則在宣告後就不能再行修改。

6. 陣列的特性

A. 靜態陣列的長度固定，無法彈性擴充：適合少增修移動、多查找的程式

B. 可以直接透過索引值查詢資料：查找快速，為 O(1)

C. 耗費的記憶體空間最小

在 C++ 中，應盡量以向量 vector 取代陣列使用，因為傳統指標沒有自動化垃圾回收機制，一旦用 new 或 malloc 來動態配置記憶體空間，之後要手動利用 free 或 delete 釋放掉，但撰寫程式時容易忘記，造成記憶體洩漏，向量則會自動釋放記憶體空間，解決了這個問題。

7. 陣列相關的解題技巧（程式競試限定）

當題目有給測資的大小上限，可以直接把陣列大小開滿到該上限：

A. 評分看執行時間，記憶體消耗不影響

B. 記得宣告成「全域變數」

a. 執行前就配置好空間

b. 能容納的空間較大

不過實務上，如面試時則不要使用這種做法，屬於競試限定的技巧。

(1) 競試說明格式

評分說明：輸入包含若干筆測試資料，每一筆測試資料的執行時間限制（time limit）均為 1 秒，依正確通過測資筆數給分

第 1 子題組 10 分，N=2，且取用次數 f(1) = f(2) = 1

第 2 子題組 20 分，N=3

第 3 子題組 45 分，N<1,000，且每一個物品 i 的取用次數 f(i)=1

第 4 子題組 25 分，N<100,000

|  |  |
| --- | --- |
| 使用陣列解題技巧 | |
| 1  2  3  4  5  6  7  8  9 | #include <iostream>  using namespace std;  int data[100000]; // 宣告在 main 函式之外，屬於全域變數  int main(){  …  return 0;  } |