3. Arrays in C++ and Java

3.1 Arrays in C++

說明:

陣列為一組相同資料型態的資料所組成的結構，使用陣列可於程式中儲存多筆同資料型態的資料，這些資料稱為陣列元素(array elements)，有組織地排列於「連續的」記憶體區塊。

陣列產生(於宣告或 runtime)後，其長度(陣列元素個數)即為固定。如以靜態的方式產生陣列，指定之陣列長度須有明確的整數值，故需為一整數值或常數，不可為變數。

一陣列元素(array element)為構成陣列的資料之一，其可由陣列名後之[ ]中指定其在陣列中排列的位置順序，亦即索引值(index)，來對其個別做存取。一陣列的索引(indices)，為正整數，由 0 開始，直到陣列之長度減一為止。對於陣列元素之操作、運算，與一般變數相同。

3.1.1 Creating/Declaring an Array and Accessing an Array Element

格式:

資料型態　陣列名[陣列長度];

以上指令所產生的 array elements 為：

陣列名[0]，陣列名[1]，陣列名[2], ……，陣列名[陣列長度 - 1]

例: 以下程式命令將產生十個整數的陣列，之後以迴圈將陣列元素之值設成等於其索引值。

```
int num[10];          //creating an array

for(int i=0; i<10; i++)
{
```

```
        num[i] = i;             // accessing an array element
        cout<<num[i];
    }
```

程式範例: oop_ex18.cpp

3.1.2 Array Initializers

1. 用於初值設定之資料數目可等於或小於陣列長度，但不可大於陣列長度。
2. 未被指定初值的陣列元其初值為零。
3. 宣告陣列時陣列之長度可以不指定，所產生的陣列長度等於所給的初值數目。

格式:

資料型態　陣列名[陣列長度] = { 資料一 , 資料二 , ……. };

例: int num[5] = { 1, 2, 3, 4, 5 };

程式範例: oop_ex19.cpp

3.1.3 Array and pointer

一維陣列的陣列名為一指標，但其固定指到陣列的第一個元素，不能對其作運算以改變其所指的位址(定址指標)。

程式範例: oop_ex20.cpp

3.1.4 String in C++

說明:

字串為字元(char)之陣列，設定字串之初值其格式如下。比一般陣列方便的是，可使用 cin 及 cout 來直接對陣列名進行輸入及輸出(不需用迴圈)，且可使用"string.h"中的函式(strcpy, strcat 等)對字串做操作。

char 陣列名[ ] = "欲輸入字串";

例: char string1[ ] = “This is a test”;
cin >> string1;
cout<<string1;

3.1.5 Multiple Dimensional Arrays
Self studying!

3.2 Dynamic Memory Allocation in C++

之前，我們所宣告的變數或陣列，其所需的記憶體是於程式編譯時所決定，並於程式執行一開始做配置，其空間大小需明確地指定，且固定不可更改(例如宣告一陣列時必須指定其長度，此長度不可為變數)，此稱為靜態(static)記憶體配置。

除了靜態記憶體配置，C++允許程式設計者做動態(dynamic)記憶體配置。動態記憶體配置是指記憶體配置是於程式執行之中進行，程式設計者可於程式中指定產生未定大小的陣列(使用變數作為指定陣列長度)，以及釋放不需使用之配置空間。我們寫程式時通常預先並不能確定記憶體之使用量，例如一成績處理的程式，需能處理不同筆數之成績，動態記憶體配置可讓程式設計者彈性地使用適當的記憶體，不至浪費或不足。

程式設計者可對所有的基本變數型態之變數或陣列做動態記憶體配置，但通常用於產生陣列居多。此外，對產生即將介紹，由程式設計者自訂之結構體(structure)以及類別(class)型態之物件，或物件之陣列亦很常用。

動態記憶體配置之資料的一大特性，是其生命週期，並不像靜態產生之資料，遵守 scope rule，而是完全由程式指令控制其配置與釋放，此特性可讓我們於函式中產生新資料，並於函式結束後仍能保有並繼續使用此資料，但缺點是程式設計者必須負責記憶體之管理。

每一個系統都會預留一塊記憶體空間，以供 runtime 時之動態記憶體配置之用，但動態記憶體配置不一定保證成功，如果配置時記憶體空間不夠，則會配置失敗導致程式錯誤，故對於記憶體需求大的程式，程式設計者應適時釋放不用的記憶體，避免記憶體不足。

3.2.1 Dynamic Memory Allocation using new

new 指令說明:

C++之 new 指令是用來動態配置記憶體的，其格式如下。new 會依所指定之型別與陣列大小配置一塊等大的記憶體，如果有指定的話，並為其設定初值，最後傳回該塊記憶體的起始位址。由於傳回位址值，故通常需用一此指定資料類型的指標來接收並儲存 new 所傳回的記憶體位址，以定位並能透過此指標來存取資料至所配置的記憶體空間。

動態配置陣列之傳回值即為陣列第一個元素的起始位址，故如用指標來接收，此指標即相當於靜態配置時之陣列名，使用上可如同靜態配置之陣列，例如用[ ]來存取 array elements。

配置單一變數(物件)格式:

```
變數型別*   指標 = new 變數型別;   //不指定初值
變數型別*   指標 = new 變數型別(初始值);   //指定初值
```

例:

```
int *p1 = new int;
double *p2;
p2 = new double(3.14159);
```

配置陣列格式:

```
變數型別 *指標名 = new 變數型別[陣列長度];     //一維陣列

//多維陣列
變數型別 (*指標名)[陣列長度 2] = new 變數型別[陣列長度 1] [陣列長度 2];
 變數型別 (*指標名)[陣列長度 2] [陣列長度 3]= new 變數型別[陣列長度 1]
                                              [陣列長度 2] [陣列長度 3];
```

例:

```
int *p3 = new int[10];
double (*p4)[4] = new double[3][4];
double (*p5)[4][5] = new double[3][4][5];
```

程式範例: oop_ex21.cpp　oop_ex22.cpp

注意要點:

1. 如果記憶體空間不夠而導致配置失敗，則 new 會傳回 NULL 值(0)，以表示配置失敗。配置失敗通常會造成程式執行時的錯誤，我們應養成檢查是否配置失敗之習慣。

2. 使用動態記憶體配置一維陣列，其長度可以使用一含未知值之變數，所產生陣列之長度是程式執行到此記憶體配置命令時，此變數的值而定。

3. 動態配置多維陣列時，第一維的長度可以是一含未知值之變數，但二維以上的長度必須是已知的數值(或常數)，且每一維的長度均不可省略。

3.2.2 Release the Allocated Memory using delete

delete 指令說明:

動態配置的記憶體與靜態宣告的變數或陣列不同，一旦動態配置了某塊記憶體，該配置記憶體將一直存在保留著，不會遵守 scope rule，一直到程式結束為止，除非我們使用 delete 指令來將該配置記憶體釋放。

delete 是針對指標內所存的記憶體位址，而非指標。當以 new 做記憶體配置時，系統會將此配置記憶體的初始位址，以及配置的相關資訊，如配置空間大小，紀錄於一配置表中，當以 delete 欲對某記憶體位址做釋放時，系統會至配置表中尋找此位址，並依配置的相關資訊釋放此配置空間，但如果欲 delete 的位址並沒有紀錄於配置表中，則會產生 runtime 時的錯誤(異常終止)。

格式:

```
delete　指標;　//釋放非陣列的記憶體配置空間
delete [ ]　指標;　//釋放陣列的記憶體配置空間
```

例:

```
delete p1;
delete []　p3;
delete []　p4;
```

程式範例: oop_ex21.cpp , oop_ex23.cpp

注意要點:

1. 不可以 delete 未經記憶體配置的位址(不在配置表中的位址)。

2. 一被釋放記憶體的指標，只是其所指之記憶體空間被釋放，此指標於程式中依然存在，可對其再進行記憶體配置。

3. 一經 delete 指令釋放記憶體的指標，其值並不是 NULL(與記憶體配置失敗不同) ，而是指到相同位址，指不過該位址之記憶體配置已被釋放，所存資料消失。

4. 如一指到有記憶體配置的指標，未經記憶體釋放，就對其進行新的記憶體配置，則此指標將會指到新配置的記憶體空間，且之前所指之記憶體空間，雖已無指標定其位，但依然存在，程式結束前不會消失。

5. 如寫一大型程式，對不再使用的記憶體空間應使用 delete 來釋放出來供其他程式部分使用，要點 4 之情形應避免，以免造成 Memory Leak。

3.2.2 Dynamic 2D array using new and delete

程式範例: oop_ex24.cpp

3.3 Arrays in Java

Declaring a Reference to Refer to an Array

與宣告變數相同，宣告陣列主要分成兩部分：宣告陣列的 type 以及陣列參考(reference)的 name。其格式如下：

type[ ] name;

type 為陣列元素(array elements)之資料型態，可為 primitive type 或 object type，[ ]

表示所產生的是陣列之參考，name 則為您所指定之參考名稱，以供識別。

例：

```
int[ ] intArray;
float[ ] floatArray;
```

以上宣告所產生的是「陣列之參考」，而非陣列本身，陣列之參考有如 C++ 中之指標，其儲存值為記憶體位址，可定位出陣列於記憶體中之所在位置。

Creating an Array

Java 陣列是使用 new 指令所產生，其格式如下：

```
new elementType[arrayLength];
```

elementType 為所產生陣列之陣列元素的資料型態，可為 primitive type 或 object type，arrayLength 為指定的陣列長度，可為整數或變數。new 將配置(elementSize * arrayLength)大小之記憶體空間，並將其初始位址傳回，存於等號左邊之參考。

例：

```
int[ ] intArr;              //declare an int array (a reference for int array)
intArr = new int[100];      //create/allocate an array of int
```

以上之陣列宣告與產生可寫為單一 statement：

```
double[ ] doubleArr = new double[200];
```

3.3.3 Array Initializers

我們可使用以下的語法捷徑，產生一陣列並設定其陣列元素的初值。所產生陣列的長度，等於 { } 間初值的數目。

例：

```
int[ ] anArray = {0,10,20,30,40};   //Create and initialize an array called anArray.
                                     //The length of n array is “5”
```

3.3.4 Accessing an Array Element

當參考所指之記憶體位址配置了陣列後，參考可以[ ]算符配合索引值(index)來存取個別陣列元素。

例：

```
intArray[3] = 10;
float val = floatArray[3];
```

與 C++陣列不同的是：(1)索引值不可以超過陣列之長度 – 1，如超出索引值範圍會發生錯誤：java.lang.ArrayIndexOutOfBoundsException。(2)陣列之長度，可由以下方式得到：

```
arrayReference.length
```

例：

```
int[ ] anArray = new int[5];
for(int i = 0; i < anArray.length ; i++)
    anArray[i] = i*10;          //This is equivalent to the creating and initializing
                                //of the anArray above
```

程式範例: oop_ex2.cpp

3.3.4 Sorting/Searching an Array

程式範例: oop_ex25.java (bubble sort)
oop_ex26.java (binary search on a sorted array)

3.3.5 Multiple-Subscripted Arrays

Java 可使用以下方式產生多為陣列，其中 x、y 均可為變數，或整數。

```
int[ ][ ] a;
a = new int[x][y];
```

亦可使用 Array Initializers，其中每一列之構成元素長度可以不同。

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

程式範例: oop_ex27.java

由以上程式範例可知，多維陣列，以二為陣列為例，其實是陣列的陣列(Array of Arrays)，每一列之構成陣列的長度可以不同。

了解了多維陣列之構成結構，我們可將一多維陣列明確地宣告建立。

程式範例: oop_ex28.java

3.3.6 Garbage Collection

Java 提供了 Garbage Collection 之機制，免除了程式設計者需實際以指令釋放不再使用的記憶體配置。簡單的說，Java 以 new 產生陣列或物件，但並沒有對應的 delete 指令，以 new 所配置之記憶體其釋放規則，是當某陣列或物件不再被任何參考參照到時，其即不可能再被程式所引用到，故其所用空間可以自記憶體中被收回。Java 之 Garbage Collection 機制即是自動偵測未被任何參考參照到的陣列或物件，「必要時」會將其所佔記憶體空間釋放。

以下配置給 a 的長度 100 之整數陣列，會由 Garbage Collection 機制給釋放。

```
int[ ] a = new int[100];
int[ ] b = new int[200];
a = b;
```

如欲釋放某物件或陣列，只需要停止參照他，例如將其參考參照到其他同類物件或陣列，如上例，或者 null，由 Garbage Collection 機制將其釋放。

```
int[ ] c = new int[300];
……
c = null;
```

由 Garbage Collection 機制管理記憶體可避免人為的錯誤(memory leak、誤 delete 仍被參照到之物件造成 dangling references、或 delete 未經配置的位址)，但未被參照到的物件之空間並非馬上被釋放回收，且須時間運作，效率較差。