Chapter 13 ： 動態記憶體配置(Dynamic Memory Allocation)

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，此長度不可爲變數)，此稱爲靜態(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(使用變數作爲指定陣列長度)，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。

首先，我們來介紹 C++所提供之較方便之動態記憶體配置方法(使用 new 以及 delete 指令)，之後我們也會介紹傳統 C 之方法(使用 malloc 以及 free 指令)。

13.1 C++ style 動態記憶體配置 (使用 new 指令)

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，並爲其設定初值，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爲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: cpp_ex61.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，其長度可以使用一個含未知值之變數，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。

13.2 C++ style 動態記憶體釋放 (使用 delete 指令)

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，一直到程式結束爲止，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: cpp_ex61.cpp , cpp_ex62.cpp , cpp_ex63.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。

13.3 C style 動態記憶體配置 (使用 malloc, free, realloc)

13.3.1 malloc 指令說明:

malloc 之功能與 new 相當，其產生一連續的記憶體空間大小等於所輸入的參數，並將配置的記憶體之起始位址以 void 指標(void*)型態傳回，須對傳回指標由 void 指標強制轉換爲所需的指標型態。

與 new 不同，使用 malloc 不能同時指定初值。配置時，格式中不需特別指定是否爲陣列，只須配置適當的空間大小。

格式:

```
變數型別*   指標 = (變數型別*) malloc(記憶體空間大小);
```

例:

```
int *p1 = (int*) malloc(sizeof(int));          //爲單一變數配置記憶體
int *p3 = (int*) malloc(10*sizeof(int));       //爲一長度爲十之陣列配置記憶體
                                               //(大小爲十個整數的空間)
```

13.3.2 free 指令說明:

free 之功能與 delete 相當，其將一配置的記憶體空間，透過輸入記憶體空間之起始指標來釋放。與 delete 不同，使用 free 對單一變數或陣列並無格式上的區別。

格式:

free(指標);

例:

free(p1);
free(p3);

程式範例: cpp_ex64.cpp

注意要點:

1. malloc, free 定義於 <stdlib.h> 標題檔中。
2. 配置失敗將傳回 NULL。
3. malloc, free 與 new, delete 最好不要混合使用。

13.3.3 realloc 指令說明:

realloc 之功能是將一已存在的記憶體空間進行重新配置。其輸入參數爲已存在的記憶體空間之指標，以及新的空間大小。

新增的空間將配置於原空間之後，如空間不足，則系統將另覓空間配置，並將原空間中之值拷貝到新空間，清除舊空間，最後將新空間之起始位址以 void 指標(void*)的型態傳回。

格式:

變數型別* 新指標 = (變數型別*) realloc(原指標,新記憶體空間大小);

例:

```
int *new_p3 = (int*) realloc(p3, 20*sizeof(int));   //改爲二十個整數的大小
```

程式範例: cpp_ex65.cpp

注意要點:

new 並沒有相同的功能，但同樣的效果可以很容易的用 new 及 delete 來寫出來。

程式範例: cpp_ex66.cpp

13.4 記憶體操作函式

C/C++提供了可對記憶體區塊中資料作操作、比較、與搜尋的函式：

(1) void* memcpy(void* s1, const void* s2, size_t n);
將指標 s2 所指之記憶體拷貝 n 筆資料到 s1 所指之記憶體。

(2) void* memmove(void* s1, const void* s2, size_t n);
將指標 s2 所指之記憶體拷貝 n 筆資料到 s1 所指之記憶體。此函式會將 s2 所指之 n 筆資料先拷貝至一暫存的陣列，再拷貝至 s1 所指之記憶體。

(3) int memcmp(const void* s1, const void* s2, size_t n);
比較 s1 與 s2 所指記憶體的前 n 筆資料。如 s1==s2 則傳回 0，如 s1<s2 則傳回負值，如 s1>s2 則傳回正值。

(4) void* memchr(const void* s, int c, size_t n);
找出 c 於 s 所指記憶體的前 n 筆資料中，首次出現的位置。如有找到則傳回其位址，否則傳回 NULL。

(5) void* memset(void* s, int c, size_t n);
將 s 所指記憶體的前 n 筆資料全設爲 c，並傳回一指向結果之指標。

注意要點:

1. 使用 void 指標。
2. memmove 支援同陣列資料之拷貝。

程式範例: cpp_ex67.cpp