

程式設計

Ch11. Dynamic Memory Allocation

Chuan-Chi Lai 賴傳淇

Department of Communications Engineering
National Chung Cheng University

Spring Semester, 2024

Outline

- 1 概念 (Concept of DMA)
- 2 動態記憶體配置 (Dynamic Memory Allocation)
- 3 malloc 函式
- 4 free 函式 – 釋放記憶體空間
- 5 calloc 函式
- 6 realloc 函式 – 一維陣列動態記憶體配置 (Dynamic 1D Array)
- 7 二維陣列動態記憶體配置 (Dynamic 2D Array)
- 8 記憶體洩漏 (Memory Leak)
- 9 常見錯誤 (Common Mistakes)
- 10 總結 (Summary)

概念 Concept of DMA

概念 (Concept of DMA)

- 為什麼我們需要“動態記憶體配置”？
- 其實很多時候根本不知道我們究竟需要多少記憶體空間，如果今天老師請你寫一個計算全班成績的程式碼，你會怎麼做？
 - ① 班上有多少位學生？
 - ② 有幾次成績需要輸入？
 - ③ 需要加權嗎？權重為何？
- 你覺得上面是否已問完所有問題？

立刻開啟你的編輯器：

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
int main(){  
    int numberOfStudent = 30, items = 10;  
    int grade[numberOfStudent][items] = {0};  
    ...  
}
```

概念 (Concept of DMA)

- 爲什麼我們需要“動態記憶體配置”？
- 你有想過老師同一門課開了好幾個班級？
- 每一個班級人數也不一樣？
- 分數的計算方法也可能不一樣？
- 那麼該怎麼辦呢？
- **ANS：動態記憶體配置 (Dynamic Memory Allocation)**

概念 (Concept of DMA)

- 在講動態記憶體配置之前，我們先來看兩個問題。
- 首先是第一個問題，請觀察下方程式碼有何問題呢？

```
1  #include <stdio.h>
2
3  int main(){
4      int length;
5      scanf("%d", &length);
6      int arr[length];
7      scanf("%d", &arr[length-1]);
8      printf("%d\n", arr[length-1]);
9  }
```

概念 (Concept of DMA)

- 雖然語法看似正確，但由於靜態陣列宣告時，編譯器會在編譯時期配置記憶體，如果輸入的 length 太大，程式可能會因為記憶體配置不足而發生錯誤，進而導致程式無法順利執行。

```
1  #include <stdio.h>
2
3  int main(){
4      int length;
5      scanf("%d", &length);
6      int arr[length];
7      scanf("%d", &arr[length-1]);
8      printf("%d\n", arr[length-1]);
9  }
```

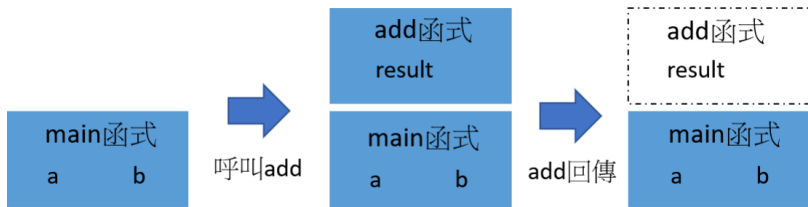
概念 (Concept of DMA)

- 接著是第二個問題，請觀察下方程式碼有何問題呢？

```
1  #include <stdio.h>
2
3  int *add(int x, int y) {
4      int result = x + y;
5      return &result;
6  }
7
8  int main(){
9      int a = 1, b = 2;
10     printf("%d\n", *add(a, b));
11 }
```


概念 (Concept of DMA)

- 根據底下的示意圖，我們可以發現在呼叫 add 函式時，result 變數會被宣告在 add 函式內部的記憶體，當函式結束時，此記憶體空間將被回收。
- 因此即使將 result 的位置回傳出來，也可能無法正確讀取其值，這時候我們會稱呼此為迷途指標 (Dangling Pointer)。



動態記憶體配置 Dynamic Memory Allocation

動態記憶體配置

- 動態陣列不同於靜態陣列，它不會占用系統配置給函式的空間。相反地，它會另外配置一塊記憶體空間，使得動態陣列能夠解決上述靜態陣列遇到的兩個問題。
- 動態記憶體配置是一種在程式執行期間分配記憶體的方法，相較於靜態記憶體配置，靜態記憶體配置是在程式編譯期間就已經完成了所有記憶體的分配。
- 動態記憶體配置可以根據需要在執行期間分配或釋放記憶體空間，避免了靜態記憶體分配的空間限制和浪費。

記憶體配置方法

- 靜態配置：
 - 在編譯時期 (compile-time)，編譯器就必須決定配置多少記憶空間給變數。
 - 記憶體使用較無彈性，配置過多會浪費，配置過少會不夠使用
- 動態配置：
 - 在程式執行時期 (run-time)，根據處理資料的需求才向系統要求記憶空間。
 - 可以有效地使用配置的記憶體。

動態記憶體配置的使用情境

- 動態記憶體配置的用意在於提高程式記憶體的靈活性和使用效率。以下是使用情境：
 - 記憶體需求不確定：當程式執行時，有些地方需要的記憶體大小可能是不確定的。動態記憶體配置可以讓程式在執行時根據需要動態分配記憶體。
 - 提高記憶體使用效率：當程式在執行期間需要使用大量記憶體時，靜態記憶體配置可能會浪費大量的記憶體空間。動態記憶體配置可以只在需要時分配記憶體。

sizeof 函式

- sizeof 函式可以用來計算變數或資料型別在記憶體中所佔的空間大小，單位為位元組 (Byte)。
- 在動態記憶體配置時，經常會搭配 sizeof 函式使用。即使是相同的變數型別，在不同的執行環境下也可能佔用不同的記憶體大小。
- 因此使用 sizeof 可以確保正確計算記憶體大小，並提高程式的可讀性和可維護性。

動態記憶體配置 (Dynamic Memory Allocation)

- 究竟要怎麼做動態記憶體配置呢？
- 首先，利用 `malloc()` or `calloc()` 來動態配置所需要的記憶體空間。
- 使用完畢記得用 `free()` 回收掉剛剛配置的記憶體空間。

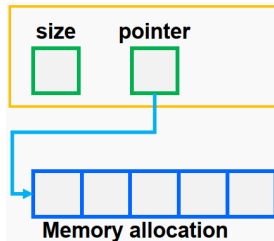
Function	Meanings
<code>void* malloc(size_t size)</code>	配置一塊指定空間大小 (<code>size_t</code>) 的記憶體，且不進行初始化，並回傳一個指標。
<code>void* calloc(size_t nitems, size_t size)</code>	配置一塊指定空間大小 (<code>size_t</code>) 的記憶體，且初始化成 0，並回傳一個指標。
<code>void* realloc(void *ptr, size_t size)</code>	調整原先指標 <code>ptr</code> 指向已安排好的記憶體空間，並回傳一個指標。
<code>void free(void *ptr)</code>	釋放 <code>malloc</code> 、 <code>calloc</code> 、或是 <code>realloc</code> 所配置的記憶體。

malloc 函式

malloc 函式

- 程式設計者透過 malloc 函式進行動態記憶體的配置，以下是 malloc 函式的原型，size_t 是無號整數，表示要分配的記憶體大小 (單位是 Byte)。
- 函式會返回一個 void* 指標，指向分配的記憶體起始地址。假如分配失敗，則會回傳 NULL。

```
void* malloc(size_t size)
```



malloc 函式

- 配置可存放 3 個整數的記憶體空間：

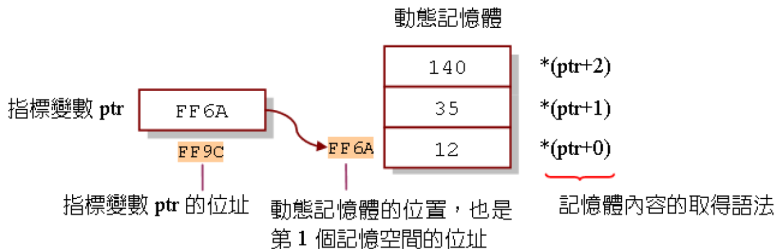
```
1 int *ptr; /* 宣告指向整數的指標 ptr */
2 ptr = (int *) malloc(12); /* 較不好的寫法 */
3 ptr = (int *) malloc(3*sizeof(int)); /* 較好的寫法 */
```

- 需要注意的是，這邊所配置的記憶體，不會自動將所有元素變成 0，取而代之的是隨機亂數。
- 此外，如果在 block 中做動態記憶體配置的時候，配置的記憶體會隨著 block 結束，而結束。
- malloc 函式返回的是 void* 型別的指標，表示分配記憶體的起始地址，並不屬於任何一種具體的型別。而在 C 語言中，對於指標變量需要明確指定其指向的型別，因此需要將 malloc 函式返回的指標進行強制型別轉換，以將其指向具體的型別。

設定與取得記憶體的内容

- 第 k 個記憶空間的内容可藉由 $*(ptr+k-1)$ 來存取：

```
1 int *ptr;  
2 ptr=(int *) malloc(3*sizeof(int));  
3 *ptr=12; /* 將 ptr 所指向的第 1 個記憶空間設值為 12 */  
4 *(ptr+1)=35; /* 將第 2 個記憶空間設值為 35 */  
5 *(ptr+2)=140; /* 將第 3 個記憶空間設值為 140 */
```

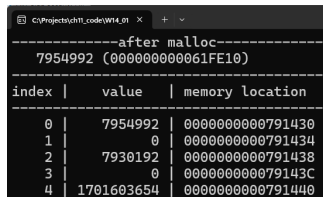


- 因為動態記憶體配置得到的空間是連續的，本來就可以當成陣列使用。
- 建議使用 `ptr[k]` 取代 `*(ptr+k)`

malloc 範例

- 透過以下程式，可以得知 malloc 分配出的空間內容不會被初始化。

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(){
5      int size = 5, i;
6      int *p = (int*) malloc(sizeof(int)*size);
7
8      printf("-----after malloc-----\n");
9      printf("%10d (%p)\n", *p, &p);
10
11     // print value
12     printf("-----\n");
13     printf("index |    value    | memory location\n");
14     printf("-----\n");
15     for (i=0; i<size; i++){
16         printf("%5d | %10d | %p\n", i, p[i], &p[i]);
17     }
18 }
```

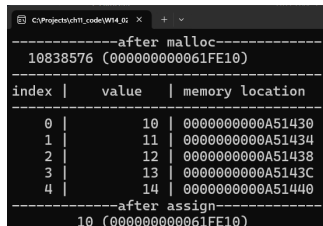


-----after malloc-----
7954992 (0000000000061FE10)

index	value	memory location
0	7954992	0000000000791430
1	0	0000000000791434
2	7930192	0000000000791438
3	0	000000000079143C
4	1701603654	0000000000791440

malloc and assign value 範例

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(){
5     int size = 5, i;
6     int *p = (int*) malloc(sizeof(int)*size);
7
8     printf("-----after malloc-----\n");
9     printf("%10d (%p)\n", *p, &p);
10
11     // assign value
12     printf("-----\n");
13     printf("index |    value    | memory location\n");
14     printf("-----\n");
15     for (i=0; i<size; i++){
16         p[i] = i+10;
17         printf("%5d | %10d | %p\n", i, p[i], &p[i]);
18     }
19
20     printf("-----after assign-----\n");
21     printf("%10d (%p)\n", *p, &p);
22 }
```



```
C:\Projects\ch11_code\W14_02  x  +  v
-----after malloc-----
10838576 (0000000000061FE10)
-----
index |    value    | memory location
-----
0 |          10 | 0000000000A51430
1 |          11 | 0000000000A51434
2 |          12 | 0000000000A51438
3 |          13 | 0000000000A5143C
4 |          14 | 0000000000A51440
-----after assign-----
10 (0000000000061FE10)
```

課後練習

- 前一個範例中，利用 for loop 做改 0 的動作十分沒有效率。
- 可以利用 memset 函式，指標變數的特定範圍內，全部變成同一個特定字元。
- 請利用 memset 就可以一次改完全部 element 內的數值，而且還省去一個 for loop。

index	value	memory location
0	0	00000000001B1550
1	0	00000000001B1554
2	0	00000000001B1558
3	0	00000000001B155C
4	0	00000000001B1560

malloc in block 範例

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(){
5      int size = 5, i;
6      if(1){ // if else block
7          int *p = (int*) malloc(sizeof(int)*size);
8
9          printf("-----after malloc-----\n");
10         printf("%10d (%p)\n", *p, &p);
11
12         // assign value
13         printf("-----after assign-----\n");
14         printf("index | value | memory location\n");
15         printf("-----\n");
16         for (i=0; i<size; i++){
17             p[i] = i+100;
18             printf("%5d | %10d | %p\n", i, p[i], &p[i]);
19         }
20
21         printf("-----after assign-----\n");
22         printf("%10d (%p)\n", *p, &p);
23     }
24     // printf("%10d (%p)\n", *p, &p); // error: 'p' undeclared (first use in this function)
25 }

```

Ex 12-3: malloc in block

-----after malloc-----		
1668416 (000000000061FE10)		
index	value	memory location
0	100	0000000000191540
1	101	0000000000191544
2	102	0000000000191548
3	103	000000000019154C
4	104	0000000000191550
-----after assign-----		
100 (000000000061FE10)		

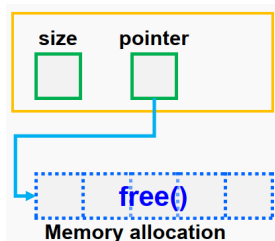
Block 內配置的記憶體：離開block 的時候，內部的變數就會被釋放。但是記憶體的數值並沒有reset。

free 函式 – 釋放記憶體空間

釋放記憶體空間 (free memory space)

- 一旦做了動態記憶體配置，工作完成後就一定要記得釋放掉!
- 在 C 語言中，記憶體釋放的方法就是使用 free 函式，free 函式的原型如下。

```
void free(void *ptr)
```



free 函式 – 釋放記憶體空間

- 要注意 ptr 必須是以 malloc、calloc 或 realloc 函式分配出來的記憶體空間，否則屬於未定義行為。
- 另外釋放完記憶體空間後，為避免重複釋放記憶體，應該將指標賦值為 NULL，也可以避免產生迷途指標 (Dangling Pointer)。

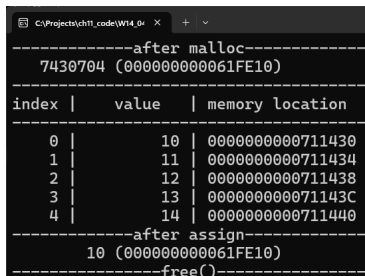
記憶體空間管理不當

- 記憶體洩漏 (memory leak) 問題
 - 動態配置的記憶體沒有用 free() 歸還系統。
 - 後續會詳細介紹。
- 記憶體區段存取失敗 (segmentation fault) 或非法記憶體存取 (illegal memory access)
 - 如果記憶空間已歸還了，卻還嘗試著去使用那塊記憶空間。

free 函式 – 釋放記憶體空間

free memory space 範例

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     int size = 5, i;
7     int *p = (int*) malloc(sizeof(int)*size);
8
9     printf("-----after malloc-----\n");
10    printf("%10d (%p)\n", *p, &p);
11
12    // assign value
13    printf("-----\n");
14    printf("index | value | memory location\n");
15    printf("-----\n");
16    for (i=0; i<size; i++){
17        p[i] = i+10;
18        printf("%5d | %10d | %p\n", i, p[i], &p[i]);
19    }
20
21    printf("-----after assign-----\n");
22    printf("%10d (%p)\n", *p, &p);
23
24    printf("-----free()-----\n");
25    free(p); // safe and okay
26 }
```

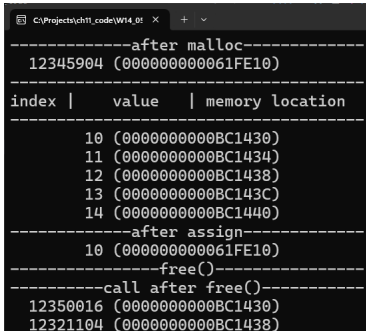


```
C:\Projects\ch11_code\W14_04 X +
-----after malloc-----
7430704 (000000000061FE10)
-----
index | value | memory location
-----
0 | 10 | 0000000000711430
1 | 11 | 0000000000711434
2 | 12 | 0000000000711438
3 | 13 | 000000000071143C
4 | 14 | 0000000000711440
-----after assign-----
10 (000000000061FE10)
-----free()-----
```

free 函式 – 釋放記憶體空間

free memory and call again 範例

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     int size = 5, i;
5     int *p = (int*) malloc(sizeof(int)*size);
6     printf("-----after malloc-----\n");
7     printf("%10d (%p)\n", *p, &p);
8     printf("-----\n");
9     printf("index |    value    | memory location\n");
10    printf("-----\n");
11    for (i=0; i<size; i++){
12        p[i] = i+10;
13        printf("%10d (%p)\n", p[i], &p[i]);
14    }
15    printf("-----after assign-----\n");
16    printf("%10d (%p)\n", *p, &p);
17    printf("-----free()-----\n");
18    free(p); // safe and okay
19    printf("-----call after free()-----\n");
20    printf("%10d (%p)\n", p[0], &p[0]);
21    printf("%10d (%p)\n", p[2], &p[2]);
22 }
```

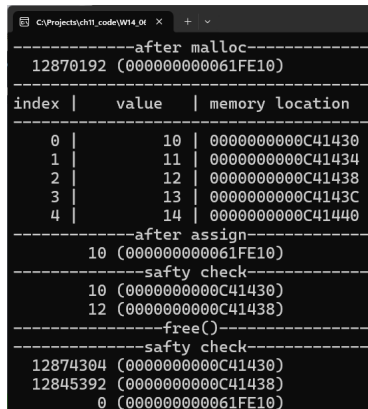


```
C:\Projects\ch11_code\W14_01 x + v
-----after malloc-----
12345904 (000000000061FE10)
-----
index |    value    | memory location
-----
10 (0000000000BC1430)
11 (0000000000BC1434)
12 (0000000000BC1438)
13 (0000000000BC143C)
14 (0000000000BC1440)
-----after assign-----
10 (000000000061FE10)
-----free()-----
-----call after free()-----
12350016 (0000000000BC1430)
12321104 (0000000000BC1438)
```

free 函式 – 釋放記憶體空間

Free & Set to 0 範例

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     int size = 5, i;
5     int *p = (int*) malloc(sizeof(int)*size);
6     printf("-----after malloc-----\n");
7     printf("%10d (%p)\n", *p, &p);
8     printf("-----\n");
9     printf("index | value | memory location\n");
10    printf("-----\n");
11    for (i=0; i<size; i++){
12        p[i] = i+10;
13        printf("%5d | %10d | %p\n", i, p[i], &p[i]);
14    }
15    printf("-----after assign-----\n");
16    printf("%10d (%p)\n", *p, &p);
17    printf("-----safty check-----\n");
18    printf("%10d (%p)\n", p[0], &p[0]);
19    printf("%10d (%p)\n", p[2], &p[2]);
20    printf("-----free()-----\n");
21    free(p); // safe and okay
22    printf("-----safty check-----\n");
23    printf("%10d (%p)\n", p[0], &p[0]);
24    printf("%10d (%p)\n", p[2], &p[2]);
25    p = 0;
26    // printf("%10d (%p)\n", p[0], &p[0]); // cannot use anymore
27    printf("%10d (%p)\n", p, &p);
28 }
```



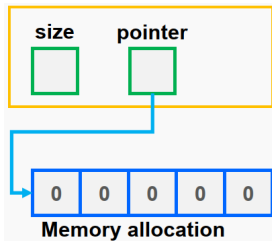
```
C:\Projects\ch11_code\W14_04 x + v
-----after malloc-----
12870192 (000000000061FE10)
-----
index | value | memory location
-----
0 | 10 | 0000000000C41430
1 | 11 | 0000000000C41434
2 | 12 | 0000000000C41438
3 | 13 | 0000000000C4143C
4 | 14 | 0000000000C41440
-----after assign-----
10 (000000000061FE10)
-----safty check-----
10 (0000000000C41430)
12 (0000000000C41438)
-----free()-----
-----safty check-----
12874304 (0000000000C41430)
12845392 (0000000000C41438)
0 (000000000061FE10)
```

calloc 函式

calloc 函式

- 在前面的範例中，會不會覺得用 malloc 配置記憶體完還需要再寫一個程式，將數值設為 0，不覺得很麻煩嗎？
- 這個時候你就可以用 calloc 函數，以下是 calloc 函式的原型，用法和 malloc 相似，但他會將分配的記憶體空間初始化成 0。

```
void *calloc(size_t nitems, size_t size)
```



calloc 範例

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(){
5     int size = 5, i;
6     int *p = (int*) calloc(size, sizeof(int));
7     printf("-----after calloc-----\n");
8     printf("%10d (%p)\n", *p, &p);
9     printf("-----\n");
10    printf("index |    value    | memory location\n");
11    printf("-----\n");
12    for (i=0; i<size; i++){
13        printf("%5d | %10d | %p\n", i, p[i], &p[i]);
14    }
15    printf("-----value check-----\n");
16    printf("%10d (%p)\n", *p, &p);
17    printf("-----value check-----\n");
18    printf("%10d (%p)\n", p[0], &p[0]);
19    printf("%10d (%p)\n", p[2], &p[2]);
20    free(p); // safe and okay
21    printf("-----safty check-----\n");
22    printf("%10d (%p)\n", p[0], &p[0]);
23    printf("%10d (%p)\n", p[2], &p[2]);
24    p = 0;
25    printf("%10d (%p)\n", p, &p);
26 }

```

```

C:\Projects\ch11_code\W14_07  X  +  v
-----after calloc-----
0 (0000000000061FE10)
-----
index |    value    | memory location
-----
0 | 0 | 00000000000C21430
1 | 0 | 00000000000C21434
2 | 0 | 00000000000C21438
3 | 0 | 00000000000C2143C
4 | 0 | 00000000000C21440
-----value check-----
0 (0000000000061FE10)
-----value check-----
0 (00000000000C21430)
0 (00000000000C21438)
-----safty check-----
12743232 (00000000000C21430)
12714320 (00000000000C21438)
0 (0000000000061FE10)

```

realloc 函式 – 一維陣列動態記憶體配置 realloc() – Dynamic 1D Array

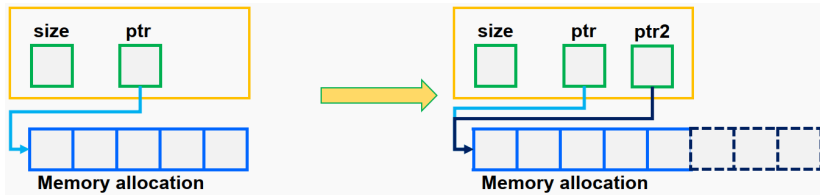
realloc 函式 – 一維陣列動態記憶體配置

- 有時候我們會面臨到，已經配置好的記憶體空間需要被調整大小，甚至需要更多的記憶體位置的時候，就可以使用到 realloc 函式。
- 這個函式主要目的，就是再跟系統要多的記憶體空間配置到指定的 pointer。
- realloc 函式有兩個參數，第一個表示先前分配的記憶體地址，第二個表示新分配需要的記憶體大小。

```
void *realloc(void *ptr, size_t size)
```

realloc 函式 – 一維陣列動態記憶體配置

- 假如原本空間後面的連續記憶體足夠，會直接擴大空間並回傳原本的地址。
- 但如果不夠的話，會尋找一段足夠長的記憶體空間，將原本的記憶體數據複製過去，並回傳新的記憶體空間，最後釋放原本的記憶體空間。



realloc 函式 – 一維陣列動態記憶體配置

- 這邊要特別注意，請避免使用以下寫法，因為假如記憶體分配失敗的話，會造成原本的記憶體洩漏。

```
1 ptr = realloc(ptr, new_size); /* 錯誤寫法 */
```

- 應該使用以下寫法，找一個指標暫存，並檢查是否分配成功。

```
1 new_ptr = realloc(ptr, new_size); /* 正確寫法 */
2 if (new_ptr == null){
3     //錯誤處理
4 }
5 ptr = new_ptr;
```

realloc 函式 – 一維陣列動態記憶體配置

realloc 範例

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(){
5     int size = 5, i;
6     int *arr1 = (int*) malloc(sizeof(int)*size);
7     printf("-----after malloc-----\n");
8     printf("%10d (%p)\n", *arr1, &arr1);
9     printf("-----\n");
10    printf("index |    value    | memory location\n");
11    printf("-----\n");
12    // assign value
13    for (i=0; i<size; i++){
14        arr1[i] = i+10;
15        printf("%5d | %10d | %p\n", i, arr1[i], &arr1[i]);
16    }
17    printf("-----after assign-----\n");
18    printf("%10d (%p)\n", *arr1, &arr1);
```

```
20    printf("-----after realloc()-----\n");
21    int *arr2 = realloc(arr1, sizeof(int)*size*2);
22    // print value
23    printf("-----\n");
24    printf("index |    value    | memory location\n");
25    printf("-----\n");
26    for (int i=0; i<size*2; i++){
27        printf("%5d | %11d | %p\n", i, arr2[i], &arr2[i]);
28    }
29    printf("-----value check-----\n");
30    printf("%10d (%p)\n", arr1[0], &arr1[0]);
31    printf("%10d (%p)\n", arr2[2], &arr2[2]);
32    printf("-----free()-----\n");
33    // free(arr1); <= that is unnecessary
34    free(arr2); // safe and okay
35    printf("-----safty check-----\n");
36    printf("%10d (%p)\n", arr1[0], &arr1[0]);
37    printf("%10d (%p)\n", arr2[2], &arr2[2]);
38 }
```

realloc 函式 – 一維陣列動態記憶體配置

realloc 範例輸出

```
C:\Projects\ch11_code\W14_01 x + v
-----after malloc-----
7823920 (000000000061FE00)
-----
index | value | memory location
-----
0 | 10 | 0000000000771430
1 | 11 | 0000000000771434
2 | 12 | 0000000000771438
3 | 13 | 000000000077143C
4 | 14 | 0000000000771440
-----after assign-----
10 (000000000061FE00)
-----after realloc()-----
index | value | memory location
-----
0 | 10 | 0000000000771430
1 | 11 | 0000000000771434
2 | 12 | 0000000000771438
3 | 13 | 000000000077143C
4 | 14 | 0000000000771440
5 | 1447976051 | 0000000000771444
6 | -1694498661 | 0000000000771448
7 | 48803 | 000000000077144C
8 | 7828032 | 0000000000771450
9 | 0 | 0000000000771454
-----value check-----
10 (0000000000771430)
12 (0000000000771438)
-----free()-----
-----safty check-----
7828032 (0000000000771430)
7799120 (0000000000771438)
```


二維陣列動態記憶體配置 Dynamic 2D Array

二維陣列動態記憶體配置 (Dynamic 2D Array)

二維陣列動態記憶體配置 (Dynamic 2D Array)

- 既然可以做到一維陣列的動態記憶體配置，二維陣列當然也可以做。
- 在介紹動態二維陣列之前，我們先了解一下靜態宣告的二維陣列是如何運作的。
- 實際上，靜態二維陣列在記憶體中的儲存方式與一維陣列相同，透過指定 Column 的大小，我們可以得知一行有多少元素，進而轉換成二維陣列的形式。
- 詳細的圖請參考下一頁。

二維陣列動態記憶體配置 (Dynamic 2D Array)

- 假設一個 element 大小為 1 byte。

二維陣列

ColSize = 2

RowSize = 3

A[0][0]	A[0][1]
A[1][0]	A[1][1]
A[2][0]	A[2][1]



記憶體

A[0][0]
A[0][1]
A[1][0]
A[1][1]
A[2][0]
A[2][1]

$$\text{Address} \leq (\text{Base} + (\text{row} * \text{ColSize}) + \text{col})$$

$$0x100 \leq (0x100 + (0 * 2) + 0)$$

$$0x101 \leq (0x100 + (0 * 2) + 1)$$

$$0x102 \leq (0x100 + (1 * 2) + 0)$$

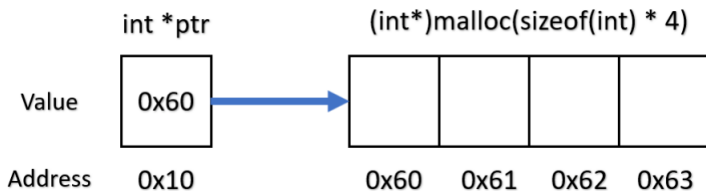
$$0x103 \leq (0x100 + (1 * 2) + 1)$$

$$0x104 \leq (0x100 + (2 * 2) + 0)$$

$$0x105 \leq (0x100 + (2 * 2) + 1)$$

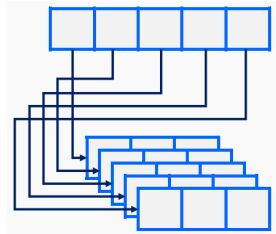
二維陣列動態記憶體配置 (Dynamic 2D Array)

- 當我們宣告靜態陣列時，變數等於第一個元素的記憶體地址。
- 但動態陣列並不是這樣操作的，當我們宣告動態一維陣列時，我們會先宣告一個指標，然後將指標指向動態配置出的一塊記憶體區域。

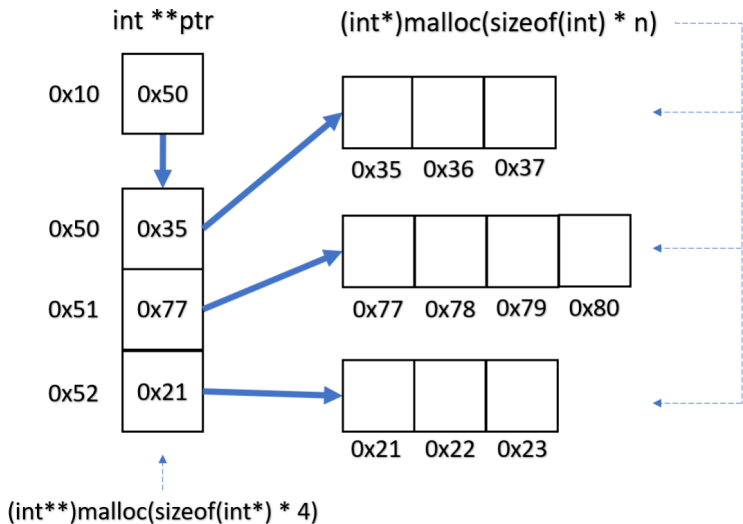


二維陣列動態記憶體配置 (Dynamic 2D Array)

- 接下來談談動態二維陣列的部分。如果我們將它與動態一維陣列做比較，我們會發現動態二維陣列其實就是把原本的指標（也就是指向一維陣列的指標）再開成一個陣列，讓每個元素都指向一個動態配置的一維陣列。
- 簡單來說，動態二維陣列就是一個指向一堆動態一維陣列的指標陣列，需要進行兩次配置，會使用到 pointer to pointer，概念圖如右。
- 詳細的圖請參考下一頁。



二維陣列動態記憶體配置 (Dynamic 2D Array)



二維陣列動態記憶體配置 (Dynamic 2D Array)

- 理解背後記憶體配置的概念後，接著來看看一個 3x4 的陣列動態記憶體配置的實作範例。

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main() {
5      int rows = 3;
6      int cols = 4;
7      // Dynamically allocate memory for the 2D array
8      int **arr = (int **)malloc(sizeof(int *) * rows);
9      for (int i = 0; i < rows; i++) {
10         arr[i] = (int *)malloc(sizeof(int) * cols);
11     }
12 }
```

二維陣列動態記憶體配置 (Dynamic 2D Array)

二維陣列動態記憶體的空間釋放

- 在釋放記憶體時，我們要注意不能直接釋放整個陣列 (arr)，而應該逐步釋放每個元素 (arr[i]) 所佔用的記憶體，最後才能釋放整個陣列 (arr) 所佔用的記憶體。
- 換句話說，我們需要按照配置陣列時的步驟逆向操作，以確保記憶體被正確地釋放。如果我們直接釋放整個陣列，會導致其它部分的記憶體被錯誤地釋放或無法釋放，從而導致記憶體洩漏或其它問題。因此我們應該謹慎對待釋放記憶體的操作，以確保程式運行的穩定性和可靠性。
- 程式碼範例請見下頁。

二維陣列動態記憶體配置 (Dynamic 2D Array)

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main() {
5      int rows = 3;
6      int cols = 4;
7      // Dynamically allocate memory for the 2D array
8      int **arr = (int **)malloc(sizeof(int *) * rows);
9      for (int i = 0; i < rows; i++) {
10         arr[i] = (int *)malloc(sizeof(int) * cols);
11     }
12
13     // Free the dynamically allocated memory
14     for (int i = 0; i < rows; i++) {
15         free(arr[i]);
16     }
17     free(arr);
18     arr = NULL;
19 }
```

記憶體洩漏 Memory Leak

記憶體洩漏 (Memory Leak)

- 當我們使用動態記憶體配置的時候，系統並不會自動回收該記憶體空間，因此如果我們分配記憶體且不回收，就可能發生記憶體洩漏 (memory leak)。
- 記憶體洩漏是指程式進行動態記憶體分配，但在程式結束時沒有正確釋放這些記憶體空間，導致記憶體佔用不斷增加，最終可能導致程式崩潰或系統資源耗盡。

記憶體洩漏 (Memory Leak)

- 在 C/C++ 中，有許多 debug tool 會自動檢查不能存取的記憶體，進而達到避免記憶體流失的問題。這一般會出現兩個議題：
 - ❶ 重要機密資訊沒有隨著程式結束而釋放掉，導致駭客有機會 trace 到這些資料內容。
 - ❷ 因為記憶體不斷的洩漏 (沒有釋放)，所以所剩的記憶體空間會越來越小，最後就沒有空間了。
- 以下的範例，是讓大家了解常見的記憶體洩漏會出現在哪裡。

記憶體洩漏 (Memory Leak)

- Memory Leak – 範例 1 – forget to free

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(){
5      int size = 10;
6      int *p = (int*) malloc(sizeof(int)*size);
7
8      // after malloc
9      printf("%10d (%p)\n", *p, &p);
10
11     // free memory space
12     // ...
13 }
```

記憶體洩漏 (Memory Leak)

- Memory Leak – 範例 2 – allocate too large memory

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(){
5      int size = 100000;
6      int *p = (int*) malloc(sizeof(int)*size);
7
8      // after malloc
9      printf("Memory Leak 02 :: too large memory allocation\n");
10     printf("%10d (%p)\n", *p, &p);
11
12     // free memory space
13     free(p); // safe and okay
14     p = 0;
15 }
```

記憶體洩漏 (Memory Leak)

- Memory Leak – 範例 3 – free correct variable (func)

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int *expand(int size){
5      int *exp = (int*) malloc(sizeof(int)*size);
6      return exp;
7  }
8
9  int main(){
10     /* Memory Leak (function) 03 :: free correct variable? */
11     int *arr = expand(10);
12     free(arr); // Is it safe? Yes.
13     arr = 0;
14 }
```

記憶體洩漏 (Memory Leak)

- Memory Leak – 範例 4 – free correct variable (func)

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int expand(int *exp, int size){
5      exp = (int*) malloc(sizeof(int)*size);
6  }
7
8  int main(){
9      /* Memory Leak (function) 04 :: free correct variable? */
10     int *arr;
11     expand(arr, 10);
12     free(arr); // Is it safe? No.
13 }
```


記憶體洩漏 (Memory Leak)

- Memory Leak – 範例 5 – free correct variable (func)

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int expand(int **exp, int size){
5      *exp = (int*) malloc(sizeof(int)*size);
6  }
7
8  int main(){
9      /* Memory Leak (function) 05 :: free correct variable? */
10     int *arr;
11     expand(&arr, 10);
12     free(arr); // Is it safe? Yes.
13 }
```

常見錯誤 Common Mistakes

常見錯誤 (Common Mistakes)

- 最後我們來談一下常見的錯誤 (Common Mistakes) :
 - ❶ 釋放記憶體之後再次呼叫 (call after free)
 - ❷ 釋放兩次記憶體 (free twice)
 - ❸ 釋放錯誤的變數記憶體空間 (free wrong data type)

“
If you love someone,
set them free.
”

常見錯誤 (Common Mistakes)

- Common Mistakes – 範例 1 – call after free

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(){
5      /* Common Mistake 01 :: call after free */
6      int size = 10;
7      int *p = (int*) malloc(sizeof(int)*size);
8
9      // free memory space
10     free(p); // safe and okay
11     p = 0;
12
13     // call again
14     printf("%d\n", p[1]); // occurs error
15 }
```

常見錯誤 (Common Mistakes)

- Common Mistakes – 範例 2 – free twice

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(){
5      /* Common Mistake 02 :: free twice */
6      int size = 10;
7      int *p = (int*) malloc(sizeof(int)*size);
8
9      int *q = p;
10     printf("%p %p\n", p, q);
11
12     // free memory space
13     free(q); // safe and okay
14     free(p); // that is redundant!
15 }
```

常見錯誤 (Common Mistakes)

- Common Mistakes – 範例 3 – free wrong data type

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int main(){
4      /* Common Mistake 03 :: free wrong type */
5      int size = 10;
6      int *p = (int*) malloc(sizeof(int)*size);
7      // free memory space
8      free(p); // safe and okay
9      free(size); // => warning: passing argument 1 of
                  // 'free' makes pointer from integer without a cast
                  // [-Wint-conversion]
10 }
```

總結 Summary

總結 (Summary)

- 學會動態配置記憶體後，最後比較一下動態跟靜態的差異吧。

特性	靜態陣列	動態陣列
記憶體配置	編譯時期	執行時期
建立方式	宣告時指定大小	動態分配記憶體
大小	固定	可變
釋放記憶體	系統自動回收	必須手動釋放記憶體

Q & A