

程式設計 (一)

CH12. 動態記憶體配置

Ming-Hung Wang 王銘宏

tonymhwang@cs.ccu.edu.tw

Department of Computer Science and Information Engineering
National Chung Cheng University

Fall Semester, 2021

本章目錄

1. 動態記憶體配置
2. 一維動態陣列
3. 雙重指標與二維動態陣列
4. 字串與記憶體操作

動態記憶體配置

為什麼動態記憶體配置？

以前我們寫的程式，都是先將變數宣告好，在開始執行時，系統配置記憶體空間給這些變數，並在程式結束後釋放記憶體空間。

動態記憶體配置

而如果有以下需求，則要使用「動態記憶體配置」

- 宣告陣列的時候還不確定所需陣列大小。
- 希望不再使用變數時，釋放記憶體空間。
- 希望能寫一個能產生新的陣列的副程式，且該陣列不會隨著副程式結束而釋放。

動態記憶體配置函式


<stdlib.h> 中提供了動態記憶體相關函式

- `void *malloc(size_t size)`
 - (memory allocation) 配置指定大小的一段記憶體空間，並傳回空間的起始位置 (配置的空間未初始化)。
- `void free(void *ptr)`
 - 釋放動態記憶體空間。

動態記憶體配置

以下程式碼，malloc 配置了一段記憶體，起始位址為 00BB13B8，將此位址轉型為 int 指標並存入指標變數 ptr。

```
1 // Memory allocation
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 int main() {
6     int *ptr = (int *)malloc(sizeof(int));
7     if (ptr != NULL) {
8         *ptr = 100;
9         printf("%p\n%d\n", ptr, *ptr);
10        free(ptr);
11    }
12    else
13        printf("Not enough memory");
14 }
```

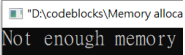


The screenshot shows a debugger window with the title '選擇 "D:\codeblock"'. It displays two memory locations: the first is at address 00BB13B8 with a value of 100, and the second is at address 100 with a value of 100.

動態記憶體配置

如果欲配置的記憶體空間太大而無法配置，
malloc 會回傳 NULL。

```
1 // Memory allocation
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 int main() {
6     int *ptr = (int *)malloc(1000000000000000000);
7     if (ptr != NULL) {
8         *ptr = 100;
9         printf("%p\n%d\n", ptr, *ptr);
10        free(ptr);
11    }
12    else
13        printf("Not enough memory\n");
14 }
```



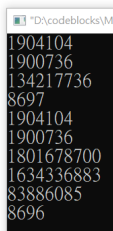
一維動態陣列

一維動態陣列

使用 malloc 配置動態陣列

使用 malloc 配置的記憶體空間並不會初始化為 0。

```
1 // Memory allocation
2 #include <stdio.h>
3 #include <stdlib.h>
4 #define SIZE 10
5
6 int main() {
7     int *ptr = (int *)malloc(sizeof(int) * SIZE);
8     if (ptr != NULL) {
9         for (int i = 0; i < SIZE; i++)
10             printf("%d\n", ptr[i]);
11         free(ptr);
12     }
13     else
14         printf("Not enough memory\n");
15 }
```



*D:\codeblocks\M

1904104
1900736
134217736
8697
1904104
1900736
1801678700
1634336883
83886085
8696

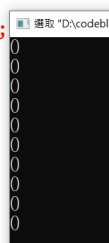
動態陣列配置函式

- `void *calloc(size_t nitems, size_t size)`
 - (contiguous allocation) 配置陣列用的記憶體空間，並傳回空間的起始位址，配置的空間會被初始化為 0。

一維動態陣列

使用 calloc 配置的記憶體空間會初始化為 0

```
1 // Memory allocation
2 #include <stdio.h>
3 #include <stdlib.h>
4 #define SIZE 10
5
6 int main() {
7     int *ptr = (int *)calloc(SIZE, sizeof(int));
8     if (ptr != NULL) {
9         for (int i = 0; i < SIZE; i++)
10             printf("%d\n", ptr[i]);
11         free(ptr);
12     }
13     else
14         printf("Not enough memory\n");
15 }
```



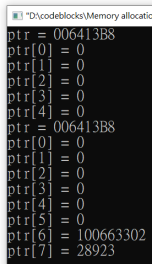
改變動態配置記憶體空間大小

- `void *realloc(void *ptr, size_t size)`
 - 改變已配置的動態記憶體空間的大小為 `size`，並回傳空間的起始位址，新配置的空間未初始化。

一維動態陣列

使用 realloc 新配置的空間不會初始化

```
12 int main() {  
13     int *ptr = (int *)calloc(SIZE, sizeof(int));  
14     int *tmp, size = SIZE;  
15     if (ptr == NULL) {  
16         printf("Not enough memory\n");  
17         return 0;  
18     }  
19     printArray(ptr, size);  
20     tmp = (int *)realloc(ptr, sizeof(int) * (SIZE + 3));  
21     if (tmp != NULL) {  
22         ptr = tmp;  
23         size = SIZE + 3;  
24     }  
25     else  
26         printf("Not enough memory to realloc\n");  
27     printArray(ptr, size);  
28     free(ptr);  
29 }
```

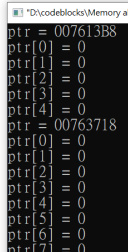


```
"D:\codeblocks\Memory allocatic  
ptr = 006413B8  
ptr[0] = 0  
ptr[1] = 0  
ptr[2] = 0  
ptr[3] = 0  
ptr[4] = 0  
ptr = 006413B8  
ptr[0] = 0  
ptr[1] = 0  
ptr[2] = 0  
ptr[3] = 0  
ptr[4] = 0  
ptr[5] = 0  
ptr[6] = 100663302  
ptr[7] = 28923
```

一維動態陣列

若原本配置的位置的空位不足以配置新的指定大小，則會重新配置到新的位址。

```
12 int main() {  
13     int *ptr = (int *)calloc(SIZE, sizeof(int));  
14     int *tmp, size = SIZE;  
15     if (ptr == NULL) {  
16         printf("Not enough memory\n");  
17         return 0;  
18     }  
19     printArray(ptr, size);  
20     tmp = (int *)realloc(ptr, sizeof(int) * (SIZE * 100));  
21     if (tmp != NULL) {  
22         ptr = tmp;  
23         size = SIZE * 100;  
24     }  
25     else  
26         printf("Not enough memory to realloc\n");  
27     printArray(ptr, size);  
28     free(ptr);  
29 }
```

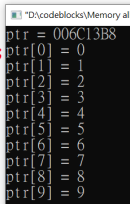


```
*D:\codeblocks\Memory al  
ptr = 007613B8  
ptr[0] = 0  
ptr[1] = 0  
ptr[2] = 0  
ptr[3] = 0  
ptr[4] = 0  
ptr = 00763718  
ptr[0] = 0  
ptr[1] = 0  
ptr[2] = 0  
ptr[3] = 0  
ptr[4] = 0  
ptr[5] = 0  
ptr[6] = 0  
ptr[7] = 0
```

一維動態陣列

函式回傳動態陣列

```
1 // Memory allocation
2 #include <stdio.h>
3 #include <stdlib.h>
4 #define SIZE 10
5
6 int *createArray(int n) {
7     int* arr = (int*)calloc(n, sizeof(int));
8     if (arr != NULL)
9         for (int i = 0; i < n; i++)
10             arr[i] = i;
11     return arr;
12 }
13
14 void printArray(int* ptr, int n) {
15     printf("ptr = %p\n", ptr);
16     for (int i = 0; i < n; i++)
17         printf("ptr[%d] = %d\n", i, ptr[i]);
18 }
19
20 int main() {
21     int* ptr = createArray(10);
22     if (ptr != NULL)
23         printArray(ptr, 10);
24 }
```



```
ptr = 006C13B8
ptr[0] = 0
ptr[1] = 1
ptr[2] = 2
ptr[3] = 3
ptr[4] = 4
ptr[5] = 5
ptr[6] = 6
ptr[7] = 7
ptr[8] = 8
ptr[9] = 9
```


雙重指標與二維動態陣列

雙重指標與二維動態陣列

雙重指標 (pointer to pointer) 型態所儲存的值為指標的位址，可以說是指向指標的指標。

```
1 // Ptr to Ptr
2 #include <stdio.h>
3
4 int main() {
5     int a = 10;
6     int *ptr = &a;
7     int** ptrPtr = &ptr;
8     printf("a      = %d\t\t&a      = %p\n", a, &a);
9     printf("ptr     = %p\t\t&ptr     = %p\n", ptr, &ptr);
10    printf("ptrPtr  = %p\t\t&ptrPtr  = %p\n", ptrPtr, &ptrPtr);
11 }
```

"D:\codeblocks\Ptr to Ptr.exe"

a	= 10	&a	= 0061FF1C
ptr	= 0061FF1C	&ptr	= 0061FF18
ptrPtr	= 0061FF18	&ptrPtr	= 0061FF14

二維動態陣列

上一章 (CH11 p.34) 我們使用指標陣列來儲存許多靜態陣列的位址，來達成二維陣列的效果，二維動態陣列則是將上一章的靜態陣列部分全部替換成動態陣列。

雙重指標與二維動態陣列

建立一個 3*5 的動態二維陣列

```
1 // 2D dynamic array
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 int main() {
6     //create
7     int** arr = (int**)calloc(3, sizeof(int *));
8     if (arr != NULL) {
9         for (int i = 0; i < 3; i++)
10             arr[i] = (int*)calloc(5, sizeof(int));
11         //free
12         for (int i = 0; i < 3; i++)
13             free(arr[i]);
14         free(arr);
15     }
16 }
```

練習

使用二維動態陣列計算矩陣乘法，實作以下副程式

```
int** createMatrix(int x, int y);  
int** multiMatrix(int** a, int** b, int x, int y, int z);  
void printMatrix(int** a, int x, int y);  
void freeMatrix(int** a, int x);
```

createMatrix 會讓使用者輸入矩陣內容並建立二維動態陣列，
multiMatrix 會使用兩個矩陣相乘並回傳一個新的二維動態陣列，
printMatrix 會將矩陣內容印出，freeMatrix 會將二維動態陣列釋放。

主程式

```
int main() {  
    int x, y, z, **m1, **m2, **m3;  
    scanf("%d%d%d", &x, &y, &z);  
    m1 = createMatrix(x, y);  
    m2 = createMatrix(y, z);  
    m3 = multiMatrix(m1, m2, x, y, z);  
    printMatrix(m3, x, z);  
    freeMatrix(m1, x);  
    freeMatrix(m2, y);  
    freeMatrix(m3, x);  
}
```

範例輸入

```
2 3 4
3 5 4
1 2 3

2 0 3 6
1 2 3 2
4 2 3 0
```

範例輸出

```
27 18 36 28
16 10 18 10
```


字串與記憶體操作

字串與記憶體操作

在第 9 章時，我們介紹了一些 `<string.h>` 提供的函式

- 計算字串長度： `strlen`
- 複製字串： `strcpy`、`strncpy`
- 連接字串： `strcat`、`strncat`
- 比較字串： `strcmp`、`strncmp`

字串與記憶體操作

在本節，我們來認識更多 `<string.h>` 提供的函式

- 搜尋相關
 - 搜尋字元：strchr、strrchr
 - 搜尋字串：strstr
 - 搜尋字元集：strpbrk、strspn、strcspn
 - 切割字串：strtok
- 記憶體函式
 - memcpy、memmove、memcmp、memchr、memset

搜尋字元： strchr、strrchr

```
char * strchr ( const char * str, int character );  
char * strrchr ( const char * str, int character );
```

函式 strchr 尋找字串中某個字元第一次出現的位置，
若沒找到則回傳 NULL。

函式 strrchr 會尋找字串中某個字元最後一次出現的位置，
若沒找到則回傳 NULL。

字串與記憶體操作

尋找字串中第一個與最後一個出現的字元 'a'

```
1 // string.h
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 int main() {
6     char text[] = "I ate an apple.";
7     char c = 'a', * tmp;
8     tmp = strchr(text, c);
9     printf("First '%c' at address %p, index %d.\n", c, tmp, tmp - text);
10    tmp = strrchr(text, c);
11    printf("Last '%c' at address %p, index %d.\n", c, tmp, tmp - text);
12 }
```

D:\codeblocks\string.h.exe

```
First 'a' at address 0061FF0A, index 2.
Last 'a' at address 0061FF11, index 9.
```

搜尋字串： strstr

```
char * strstr ( char * str1, const char * str2 );
```

函式 strstr 會搜尋它的第一個字串引數，找出第二個字串引數第一次出現的位置。

字串與記憶體操作

尋找字串中第一個出現的字串 “ate”

```
1 // string.h
2 #include <stdio.h>
3 #include <string.h>
4
5 int main() {
6     char text[] = "I ate an apple.";
7     char target[] = "ate";
8     char* tmp;
9     tmp = strstr(text, target);
10    printf("First '%s' at address %p, index %d.\n", target, tmp, tmp - text);
11 }
```

D:\codeblocks\string.h.exe

First 'ate' at address 0061FF0E, index 2.

搜尋字元集：strpbrk

```
char * strpbrk ( char * str1, const char * str2 );
```

函式 strpbrk 會尋找第一個字串中第一次出現
屬於第二個字串之字元的位置。

字串與記憶體操作

使用迴圈呼叫 strpbrk 找出母音

```
1 // string.h
2 #include <stdio.h>
3 #include <string.h>
4
5 int main() {
6     char text[] = "I ate an apple.";
7     char target[] = "AEIOUaeiou";
8     char* tmp;
9     printf("Vowel in '%s' : ", text);
10    tmp = strpbrk(text, target);
11    while (tmp != NULL) {
12        printf("%c ", *tmp);
13        tmp = strpbrk(tmp + 1, target);
14    }
15    puts("");
16 }
```

D:\codeblocks\string.h.exe

Vowel in 'I ate an apple.' : I a e a a e

搜尋字元集：strspn、strcspn

```
size_t strspn ( const char * str1, const char * str2 );  
size_t strcspn ( const char * str1, const char * str2 );
```

函式 `strspn` 會判斷第一個字串從頭開始到第一個不屬於第二個字串的字元為止，共有多少個字元。此函式會傳回這一段字串的長度。

函式 `strcspn` 會判斷第一個引數的字串從頭開始一直到第一個屬於第二個字串的字元為止，共有多少個字元，並傳回這一段字串的長度。

字串與記憶體操作

使用 `strspn` 與 `strcspn` 抓取字串出現的第一串數字

```
1 // string.h
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <string.h>
5
6 int main() {
7     char text[] = "I have 2500 dollars";
8     char number[] = "1234567890";
9     char* tmp;
10    int index, len;
11    index = strcspn(text, number);
12    printf("Find the first number at index %d\n", index);
13    len = strspn(text + index, number);
14    tmp = (char*)calloc(len + 1, sizeof(char));
15    if (tmp == NULL) return 0;
16    strncpy(tmp, text + index, len);
17    tmp[len] = '\0';
18    printf("The first appeared number: %s\n", tmp);
19 }
```

D:\codeblocks\string.exe

Find the first number at index 7
The first appeared number: 2500

切割字串： strtok

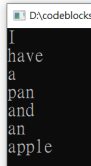
```
char * strtok ( char * str, const char * delimiters );
```

函式 strtok 用來將字串切成數個字符 (token)。
字符是由分界字元 (delimiter) 所分隔出的一連串字元。

字串與記憶體操作

將句子的每個單字分割出來

```
1 // string.h
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <string.h>
5
6 int main() {
7     char str[] = "I have a pan, and an apple.";
8     char* pch;
9     pch = strtok(str, " ,."); // begin tokenizing sentence
10    while (pch != NULL) {
11        printf("%s\n", pch);
12        pch = strtok(NULL, " ,."); // get next token
13    }
14    return 0;
15 }
16
```



```
D:\codeblocks
I
have
a
pan
and
an
apple
```

字串與記憶體操作

複製記憶體區塊：memcpy

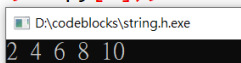
```
void * memcpy ( void * destination, const void * source,  
                size_t num );
```

函式 memcpy 會從第二個引數所指向的物件中複製某個指定數目的字元到第一個引數所指向的物件。此函式的指標引數可以指向任何資料型別的物件。memcpy 函式會將物件的位元組當做字元來處理。

字串與記憶體操作

使用 memcpy 複製陣列內容

```
1 // string.h
2 #include <stdio.h>
3 #include <string.h>
4 #define SIZE 5
5
6 int main() {
7     int arr[SIZE] = {2, 4, 6, 8, 10};
8     int copy[SIZE];
9     memcpy(copy, arr, sizeof(arr));
10    for (int i = 0; i < SIZE; i++)
11        printf("%d ", copy[i]);
12    puts("");
13 }
```



A terminal window titled "D:\codeblocks\string.h.exe" displays the output of the program: "2 4 6 8 10".

複製記憶體區塊：memmove

```
void * memmove ( void * destination, const void * source,  
                 size_t num );
```

函式 memmove 會從第二個引數所指向的物件中複製某個指定數目的字元到第一個引數所指向的物件。複製動作會先複製到一個暫時的陣列，然後再由這個暫時的陣列複製到第一個物件。

字串與記憶體操作

memmove 與 memcpy 和 strncpy 最大的不同是
memmove 可以將複製的來源區塊與貼上的目標區塊重疊。

```
1 // string.h
2 #include <stdio.h>
3 #include <string.h>
4
5
6 int main() {
7     char text[100] = "I have an apple and a pineapple.";
8     char* target = "apple";
9     char* dest = strstr(text, target);
10    int len = strlen(target);
11    memcpy(dest + len, dest, strlen(dest));
12    puts(text);
13 }
```

D:\codeblocks\string.h.exe

I have an appleapple and a pineapple.

比較記憶體區塊：memcmp

```
int memcmp ( const void * ptr1, const void * ptr2,  
             size_t num );
```

函式 memcmp 會比較第一個和第二個引數的某個指定個數的字元。
此函式的指標引數可以指向任何資料型別的物件。
memcmp 函式會將物件的位元組當做字元來處理。

字串與記憶體操作

使用 memcmp 比較陣列內容

```
1 // string.h
2 #include <stdio.h>
3 #include <string.h>
4
5
6 int main() {
7     int arr1[] = {2, 3, 5, 7};
8     int arr2[] = {2, 3, 5, 8};
9     int arr3[] = {2, 3, 5, 7};
10    if (memcmp(arr1, arr2, sizeof(arr1)) == 0)
11        puts("arr1 == arr2");
12    else
13        puts("arr1 != arr2");
14    if (memcmp(arr1, arr3, sizeof(arr1)) == 0)
15        puts("arr1 == arr3");
16    else
17        puts("arr1 != arr3");
18 }
```

D:\codeblocks\string.h.exe

```
arr1 != arr2
arr1 == arr3
```

搜尋位元組：memchr

```
void * memchr ( const void * ptr, int value, size_t num );
```

函式 memchr 會在某物件的指定個數的字元內，尋找某個位元組 (unsigned char) 第一次出現的位置。如果找到的話，memchr 會傳回指向此位元組的指標，否則便會傳回 NULL 指標。

字串與記憶體操作

尋找字串中第一個出現的字元 'u'

```
1 // string.h
2 #include <stdio.h>
3 #include <string.h>
4
5
6 int main() {
7     char text[] = "Cat is cute.";
8     char target = 'u';
9     char* found = (char*)memchr(text, target, sizeof(text));
10    if (found != NULL)
11        printf("%c is at index %d.\n", target, found - text);
12    else
13        printf("%c is not found.\n", target);
14 }
```

D:\codeblocks\string.h.exe

'u' is at index 8.

填充位元組：memset

```
void * memset ( void * ptr, int value, size_t num );
```

函式 `memset` 會將第二個引數當作 `unsigned char` 複製到第一個引數所指向之位址之後指定大小內的所有位元組。
`memset` 常用在將陣列或動態記憶體歸零。

字串與記憶體操作

使用 memset 將動態陣列歸零

```
1 // string.h
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <string.h>
5
6 int printArray(int*a, int n) {
7     for (int i = 0; i < n; i++)
8         printf("%d ", a[i]);
9     puts("");
10 }
11
12 int main() {
13     int* arr = (int*)malloc(sizeof(int) * 5);
14     printArray(arr, 5);
15     memset(arr, 0, sizeof(int) * 5);
16     printArray(arr, 5);
17 }
```

D:\codeblocks\string.exe

```
11210216 11206848 50331651 39492 11210216
0 0 0 0 0
```

參考資料： Deitel, H. M., & Deitel, P. J. (2015). C: How to program.
Upper Saddle River, N.J: Prentice Hall.