

# 程式設計

## Ch07. Pointers

Chuan-Chi Lai 賴傳淇

Department of Communications Engineering  
National Chung Cheng University

Spring Semester, 2024

- 1 變數位址與指標變數 (Address and Pointer Variables)
- 2 反參考運算子 (Dereferencing Operator)
- 3 傳址呼叫 (Call by Address)
- 4 指標的運算及與陣列的關係 (Pointer Arithmetic and Relationship with Arrays)
- 5 const 修飾詞 (const Qualifier)
- 6 函式指標 (Function Pointer)

## 變數位址與指標變數 Address and Pointer Variables

# 變數位址與指標變數 (Address and Pointer Variables)

- 在第二章曾經提過，C 語言中宣告的每個變數都實際占用於記憶體中的某段空間。記憶體中的每個位元組 (byte) 都具有一個編號，稱之為記憶體位址 (memory address)。
- 變數的記憶體位址表示其佔用於記憶體區段的起點位置，使用「&」取址運算子可以取得之。

# 變數位址與指標變數 (Address and Pointer Variables)

- 範例：

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main() // main function
5  {
6      int foo = 500;
7      printf("foo is %d\n", foo);
8      printf("foo is at %p\n", &foo);
9      return 0; // end of program
10 }
```

C:\Projects\ch07\_code\var\_ad x + v

```
foo is 500
foo is at 00000000000061FE1C
```

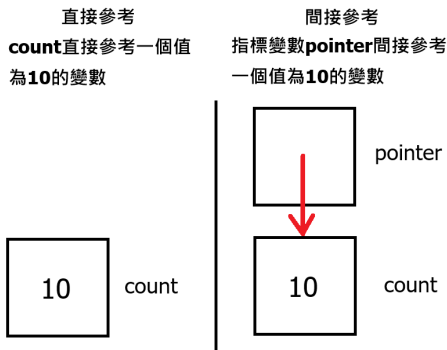
address	61FE1F	61FE1E	61FE1D	61FE1C
value	00000000	00000000	00000001	11110100

## 指標變數

- 指標變數 (pointer variable)，或簡稱指標 (pointer) 是存放記憶體位址的變數，是 C 語言最強大的功能之一。
- 藉由指標，能讓函式之間能互相傳遞變數的位址，模擬傳參考呼叫 (call by reference)。
- 指標可以配合結構 (struct) 產生與操作動態資料結構。

# 變數位址與指標變數 (Address and Pointer Variables)

- 以往我們學到的變數存放的是某個特定資料型態的數值，而指標變數所存放的卻是變數的位址。
- 如果有一指標變數 `pointer` 所存的數值是另一個變數 `count` 的位址，我們可以說 `count` 這個變數名稱直接 (directly) 參考了一個值，而 `pointer` 這個名稱則間接 (indirectly) 參考了同一個值。如下圖。



# 變數位址與指標變數 (Address and Pointer Variables)

- 以下方程式中的 `int b = 2` 為例，在宣告好變數、程式開始執行後，會去記憶體中要一塊儲存空間，然後把 2 這個資料放進去。
- C 語言中一個 `int`（整數型）的大小就占了 4 個 `byte`。

記憶體位址

0X0012FF70	15
0X0012FF74	2
0X0012FF78	39
0X0012FF7C	180
0X0012FF80	67
0X0012FF84	...

從此位址連續寫入 4 個 `byte`  
(因為整數型佔 4 個 `byte`)

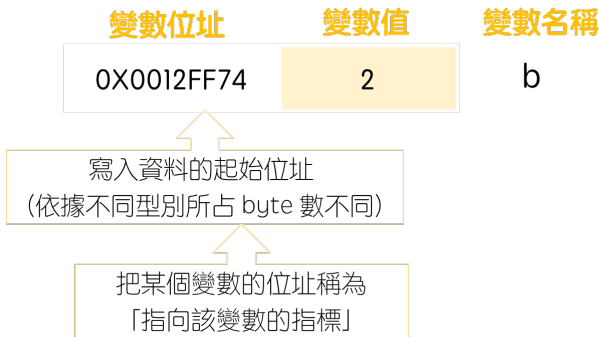
跟記憶體要一塊空間

```
void main(){  
    int a = 15;  
    int b = 2;  
    int c = 39;  
    int d = 180;  
    int e = 67;  
}
```



## 變數三要素

- 當我們宣告一個變數時，總共會有三個要素：



# 變數位址與指標變數 (Address and Pointer Variables)

- 指標 (Pointer) 就是某變數的位址。而這邊的指標變數 (Pointer Variable)，則是用來存放指標的變數。



- 案例中的 `pointer` 就是一個指標變數。變數都是用來存放「值」的，而整數型變數 `int` 就是存整數、字元型變數 `char` 就是存字元。所以這個指標變數就是用來存「地址」的變數。
- 也就是說，宣告一個指標變數，和一般宣告變數一樣，是跟記憶體要一個區域、存放這個變數的值。只是這個變數的型別是指標。
- 另外，由於 `pointer` 中存的地址是變數 `b` 的值，因此我們又把 `pointer` 稱為「變數 `b` 的指標變數」。

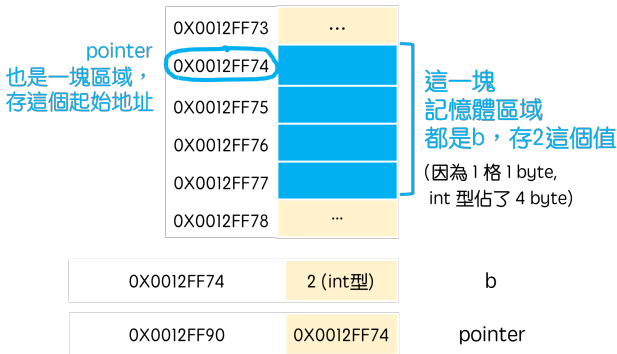
# 變數位址與指標變數 (Address and Pointer Variables)

- 欲宣告指標變數，只要在宣告時將變數名稱前面加上 \* 即可。
- 如下段程式碼宣告一個 int 變數 b 與一個 int 指標變數 pointer，將 b 設為 2，並將 pointer 指向 b：

```
1 int b;  
2 //跟記憶體要一塊區域稱為b,這塊區域專門放int型變數值  
3 b = 2;  
4 //把2這個值給變數b  
5 int *pointer; //也可以打成int* pointer  
6 //跟記憶體要一塊區域稱為pointer,這塊區域專門放指向int型變數的指標（地址）  
7 pointer = &b;  
8 //把變數b的地址值給pointer，注意不能寫成 pointer = b;
```

# 變數位址與指標變數 (Address and Pointer Variables)

- 當我們跑完這個程式碼之後，會發生這件事：



- 也就是說變數 b 在記憶體中對應了一塊儲存空間，而這塊儲存空間總有一個起始的地址。所以 pointer 對應到的就是這個起始地址。
- 在這種狀況下，就可以用「\*pointer」來拿到這個變數。

## 將指標初始化為 NULL

- 在宣告指標變數時，建議直接將指標初始化為某個存在的變數位址或是初始化為 NULL。
- NULL 的指標表示不指向任何東西。(NULL 是定義在 `<stddef.h>` 標頭檔中的符號常數，其值為指標型態的 0。)

```
1 int *pointer = NULL;
```

## 反參考運算子 Dereferencing Operator

# 反參考運算子 (Dereferencing Operator)

- 「\*」反參考運算子 (dereferencing operator) 或稱間接運算子 (indirection operator)，會傳回其運算元 (即指標變數) 所指向的物件的數值。

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

C:\Projects\ch07\_code\derefe X + v

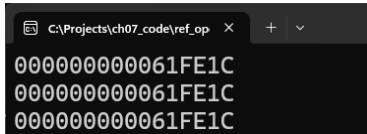
```
10 0000000000061FE14
10 0000000000061FE14
```

# 反參考運算子 (Dereferencing Operator)

## \* 與 & 的互補關係

- \* 會取記憶體位址對應的值，而 & 則是取變數的記憶體位址，兩者為相反的動作。

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



```
C:\Projects\ch07_code\ref_op  X  +  v
000000000061FE1C
000000000061FE1C
000000000061FE1C
```



## 傳址呼叫 Call by Address

# 傳址呼叫 (Call by Address)

- 利用指標和 \* 運算子可以模擬傳參考的動作。若傳給某個函式的引數應該要更改的話，可以傳遞引數的「位址」給函式。
- 當傳遞變數的位址給函式時，函式可以利用 \* 運算子存取位於呼叫者記憶體內的數值。
- scanf 就是 call by address 函式。

# 傳址呼叫 (Call by Address)

- 使用 call by value 與 call by address 來達成將數值乘 2 的函式：

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int callByValue(int a)
5  {
6      return a * 2;
7  }
8
9  int callByAddress(int *a)
10 {
11     *a = *a * 2;
12 }
```

```
13
14 int main() // main function
15 {
16     int foo = 5, bar = 6;
17     foo = callByValue(foo);
18     callByAddress(&bar);
19     printf("%d %d\n", foo, bar);
20 }
```

C:\Projects\ch07\_code\call by

10 12

# 指標的運算及與陣列的關係 (Pointer Arithmetic and Relationship with Arrays)

指標的運算及與陣列的關係

Pointer Arithmetic and Relationship with Arrays

# 指標的運算及與陣列的關係 (Pointer Arithmetic and Relationship with Arrays)

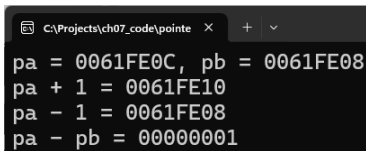
## 指標的運算 (Pointer Arithmetic)

- 指標可使用關係運算子 ( $>$ 、 $=$ 、 $<$  等) 比較大小, 也可使用  $+$ 、 $-$ 、 $++$ 、 $--$  運算子進行運算, 並可以接受下列組合:
  - 指標  $+$  整數
  - 指標  $-$  整數
  - 指標  $-$  指標 (需為同型態的指標)

# 指標的運算及與陣列的關係 (Pointer Arithmetic and Relationship with Arrays)

- 當指標進行加減法時，其單位會是指標所指向的資料型態的大小。
- 例如指標  $p$  的型態是  $\text{int}^*$ ，若計算  $p+1$  則實際上是  $+1 * \text{sizeof}(\text{int})$ 。
- 若  $pa$  與  $pb$  的型態同為  $\text{int}^*$ ，則  $pa - pb$  的值也以  $\text{sizeof}(\text{int})$  為單位之位移量 (offset)。

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main() // main function
5  {
6      int a = 10, b = 20;
7      int *pa = &a, *pb = &b;
8      printf("pa = %08X, pb = %08X\n", pa, pb);
9      printf("pa + 1 = %08X\n", pa + 1);
10     printf("pa - 1 = %08X\n", pa - 1);
11     printf("pa - pb = %08X\n", pa - pb);
12 }
```



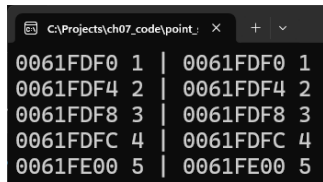
```
C:\Projects\ch07_code\pointe X + v
pa = 0061FE0C, pb = 0061FE08
pa + 1 = 0061FE10
pa - 1 = 0061FE08
pa - pb = 00000001
```

# 指標的運算及與陣列的關係 (Pointer Arithmetic and Relationship with Arrays)

## 指標與陣列的關係

- 將 int 指標 aPtr 指向 int 陣列 a 的第 0 個元素。對指標 aPtr 做加法後使用 \* 運算子取值，可以與陣列下標取值有一樣的作用。

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main() // main function
5  {
6      int a[] = {1, 2, 3, 4, 5}, i;
7      int *aPtr = a; // or int *aPtr = &a[0];
8      for (i = 0; i < 5; i++)
9      {
10         printf("%08X %d | %08X %d\n",
11             aPtr + i, *(aPtr + i), &a[i], a[i]);
12     }
13 }
```



0061FDF0	1	0061FDF0	1
0061FDF4	2	0061FDF4	2
0061FDF8	3	0061FDF8	3
0061FD FC	4	0061FD FC	4
0061FE00	5	0061FE00	5

# 指標的運算及與陣列的關係 (Pointer Arithmetic and Relationship with Arrays)

## 下標運算子 []

- 存取陣列時所使用的中括號被稱為下標運算子 (subscript operator)。
- 運算式  $\text{EXPR1}[\text{EXPR2}]$  等價於  $*((\text{EXPR1}) + (\text{EXPR2}))$ 。
- 因此不只陣列名稱可使用中括號，指標變數也可使用中括號以方便存取陣列。此方法稱為指標下標法 (pointer subscripting)。
- 甚至，若  $a$  為陣列名稱或指標變數，則  $a[2]$  與  $2[a]$  是相等的，只是  $2[a]$  這種形式不符合寫作習慣所以請不要如此編寫。



# 指標的運算及與陣列的關係 (Pointer Arithmetic and Relationship with Arrays)

## 使用指標參數傳遞陣列

- 因為指標與陣列名稱都可使用下標運算子存取陣列元素，因此也可使用指標傳遞一維陣列。

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int sumOfArray(int *arr, int size)
5 {
6     int sum = 0;
7     for (int i = 0; i < size; i++)
8         sum += arr[i];
9     return sum; // return sum
10 }
```

```
11
12 int main() // main function
13 {
14     int foo[5] = {1, 2, 3, 4, 5};
15     printf("Sum = %d\n", sumOfArray(foo, 5));
16 }
```

C:\Projects\ch07\_code\point\_ x + v

Sum = 15

# 指標的運算及與陣列的關係 (Pointer Arithmetic and Relationship with Arrays)

## 多維陣列的下標運算子

- 下標運算子的關聯性為由左至右，因此運算式  $a[2][3]$  會先運算其中的  $a[2]$ 。
- 假設  $a$  為  $\text{int}[5][10]$  的陣列，也就是大小為 5 的  $\text{int}[10]$  陣列。陣列  $a$  的元素單位大小為  $10 * \text{sizeof}(\text{int})$ 。而陣列  $a[2]$  則為一個大小為 10 的  $\text{int}$  陣列，其元素單位大小為  $\text{sizeof}(\text{int})$ 。
- 因此  $a[2][3]$  等價於  $*(*(a + 2) + 3)$  等價於  $*((\text{int}*)a + 2*10 + 3)$ 。

# 指標的運算及與陣列的關係 (Pointer Arithmetic and Relationship with Arrays)

## 思考練習

- 由上頁可推知，若 `arr` 為 `int[M][N]` 的陣列，運算式 `arr[x][y]` 等價於  $((\text{int}^*)\text{arr} + x * N + y)$ 。
- 若 `a` 為 `int[3][4]` 的陣列，`b` 為 `int[2][3][4]` 的陣列，問：
  1. `a[1][3]` 的位址與 `a[2][0]` 的位址相差幾個 bytes?
  2. `b[1][2][3]` 的位址與 `b[0][0][0]` 的位址相差幾個 bytes?

# 指標的運算及與陣列的關係 (Pointer Arithmetic and Relationship with Arrays)

- 運算優先度由高而低排序，以分隔線表示不同優先度：

運算子	關聯性	形式
[] (下標)    () (呼叫函式)    ++ (後置)    -- (後置)	由左至右	後置
sizeof()    ++ (前置)    -- (前置)    ! (邏輯 NOT)	由右至左	單元性
- (負號)    + (正號)    & (取址)    * (間接)    (type) (轉型)		
* (乘法)    / (除法)    % (模數)	由左至右	乘法
+ (加法)    - (減法)	由左至右	加法
<    <=    >    >=	由左至右	關係
==    !=	由左至右	相等
&&	由左至右	邏輯 AND
	由左至右	邏輯 OR
?:	由右至左	條件
=    +=    -=    *=    /=    %=	由右至左	指派

const 修飾詞  
const Qualifier

# const 修飾詞 (const Qualifier)

- const 修飾詞能讓你在宣告時告訴編譯器，該變數的值不應該更改。

```
1 void foo(const int a)
2 {
3     a += 10; // ERROR, const parameter is read-only
4 }
5 int main()
6 {
7     const int k = 20;
8     int x = 10;
9     foo(x);
10    k += 10; // ERROR, const variable is read-only
11 }
```

# const 修飾詞 (const Qualifier)

- 在傳址呼叫時，爲了確保陣列的內容不被更動，可使用 const 修飾指標參數。
- const 與指標變數的搭配一共四種：

指標 \ 指向的資料	非常數	常數
	非常數	常數
非常數	int *ptr	const int *ptr
常數	int *const ptr	const int *const ptr

- 可使用最小權限原則 (principle of least privilege) 來做爲使用原則。

## 指向的資料為常數

- 所指向的資料卻不能夠進行更改，若傳入陣列，則在函式中該陣列內容無法被修改。

```
1 int sumOfArray(const int *a, int len)
2 {
3     int i;
4     for (i = 1; i < len; i++)
5         a[0] += a[i]; // ERROR, location "a" is read-only
6     return a[0];
7 }
```



## 指標變數為常數

- 所指向的位址不能進行更改。

```
1 int sumOfArray(int *const a, int len)
2 {
3     int i, sum = 0;
4     for (i = 1; i < len; i++)
5         sum += *a++; // ERROR, "a" is read-only
6     return sum;
7 }
```


## 函式指標 Function Pointer

# 函式指標 (Function Pointer)

- 函式當程式執行時會存在於記憶體中，函式的名稱即為其記憶體位址。
- 使用函式指標可以存入函式的位址，並使用函式指標進行呼叫，C語法和範例如下：

```
1 RETURN_TYPE (*pointerName)(PARAMETER_LIST);
```

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int maximum(int a, int b) // max function
5  {
6      return a > b ? a : b;
7  }
8
9  int main() // main function
10 {
11     int (*funcPtr)(int, int) = maximum;
12     printf("max(3, 5) = %d\n", funcPtr(3, 5));
13 }
```



C:\Projects\ch07\_code\funcintc X + v  
max(3, 5) = 5

# 函式指標 (Function Pointer)

- 利用函式指標實現遞增或遞減的氣泡排序函式：

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int bubbleSort(int *arr, int size, int (*compare)(int, int))
5 {
6     int i, j, temp;
7     for (i = 0; i < size - 1; i++)
8         for (j = 0; j < size - 1 - i; j++)
9             if (compare(arr[j], arr[j + 1]) > 0)
10                {
11                    temp = arr[j];
12                    arr[j] = arr[j + 1];
13                    arr[j + 1] = temp;
14                }
15 }
16
17 int inc(int a, int b)
18 {
19     return a > b;
20 }
21
22 int dec(int a, int b)
23 {
24     return a < b;
25 }
```

```
27 void printArray(int *arr, int size)
28 {
29     int i;
30     for (i = 0; i < size; i++)
31         printf("%d ", arr[i]);
32     printf("\n");
33 }
34
35 int main() // main function
36 {
37     int arr[5] = {3, 5, 1, 2, 4};
38     bubbleSort(arr, 5, inc);
39     printArray(arr, 5);
40     bubbleSort(arr, 5, dec);
41     printArray(arr, 5);
42 }
```

C:\Projects\ch07\_code\bubble x + v

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

# Q & A