Chapter 7 – Pointers（指標）

Outline

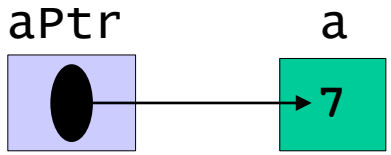
Objectives

- In this chapter, you will learn:
  - To be able to use pointers.
  - To be able to use pointers to pass arguments to functions using call by reference.
  - To understand the close relationships among pointers, arrays and strings.
  - To understand the use of pointers to functions.
  - To be able to define and use arrays of strings.

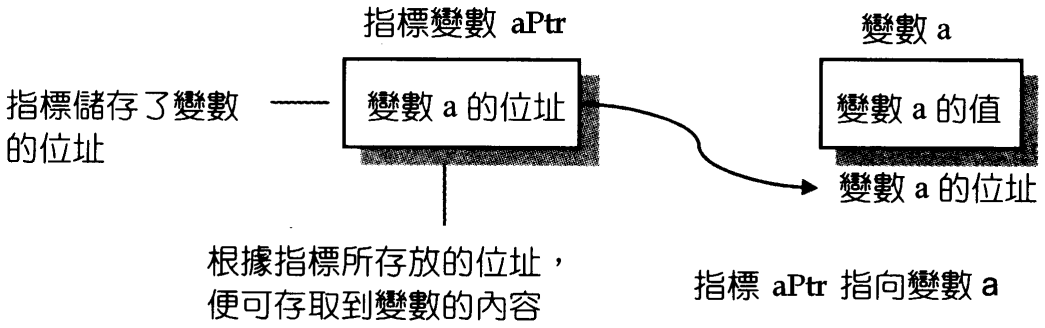
Introduction

- Pointers
  - Powerful, but difficult to master
  - Simulate call-by-reference
  - Close relationship with arrays and strings
- Pointer Variables (指標變數)
  - Pointer variables contain *memory addresses* as their values
  - Normal variables contain a specific value (direct reference), e.g., a = 7;

  - Pointer contains *the address of a normal variable* that has a specific value (indirect reference)
  - Indirection – referencing a pointer value

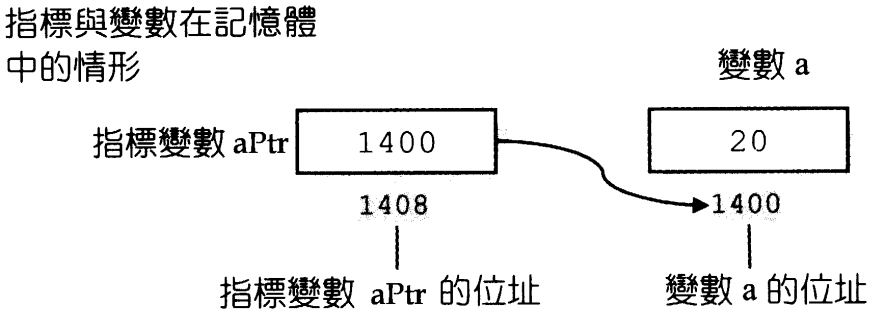


Pointer Variable - Definitions and Initialization



當我們宣告一個變數 a 時，編譯器就會配置一記憶體空間給它，每個記憶體空間都有編號，這就是記憶體位址 (address)。在 C 語言中，指標(指標變數)是用來儲存變數位址的一種特殊變數，如果**指標 aPtr 存放了變數 a 的位址**我們稱「指標 aPtr 指向變數 a (aPtr points to a)」。

因為**指標變數也是變數的一種**，所以它在記憶體中也有屬於它的位址，如右圖所示：



Pointer Variable - Definitions and Initialization

- Pointer Definitions
  - * used with pointer variables (宣告時在名稱前加 *)

```
int *aPtr;
```

  - This means "**aPtr** is a pointer which points to an integer."
  - Defines a pointer to an `int` (pointer of type `int *`)
  - Multiple pointers require using a * before each variable definition (每一個指標變數前都必須加 *)

```
int *myPtr1, *myPtr2;
```

  - Pointers can also be defined to point to objects of any other data type.
  - 注意宣告時指標變數之前的資料型態是代表指標所指向的變數之型態。 指標本身則儲存了所指向變數的地址。
  - Initialize pointers to `0`, `NULL`, or an address
    - `0` or `NULL` – points to nothing (NULL preferred)

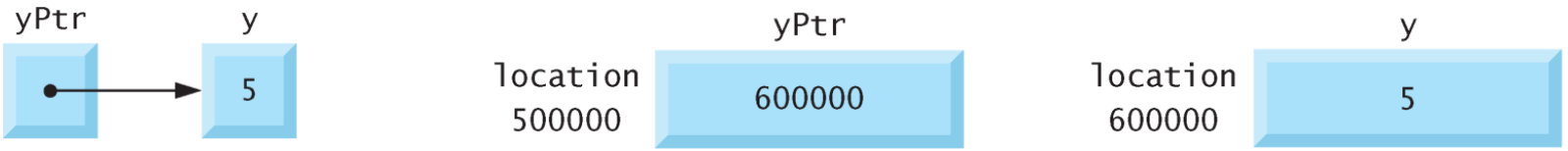
Pointer Operators

- **&** (*Address* Operator ***取址運算子***)
  - Returns **address of operand (which can be either a regular or a pointer variable)**

```
int y = 5;
int *yPtr;
yPtr = &y;      // yPtr gets address of y
```

  - Means `yPtr` "points to" `y` (指標變數 `yPtr` 指向 `y` ，也就是 `yPtr` 儲存的是 `y` 的地址)
  - 可在宣告時便將它指向某個變數

```
int y = 5;
int *yPtr = &y;
```



7.3 Pointer Operators

- * (*Indirection/Dereferencing* Operator
  *依址取值*運算子)
  - Returns a synonym/alias of what its operand (which is a pointer) points to
  - 當 **yPtr** 為指標變數時，**yPtr** 為 **y** 在記憶體中的位址，而 ***yPtr** 則為存放在該位址之值 (即 **y** 值)
  - ***yPtr** returns the value of **y** (because **yPtr** points to **y**)
  - * can be used for assignment
    - Returns alias to an object

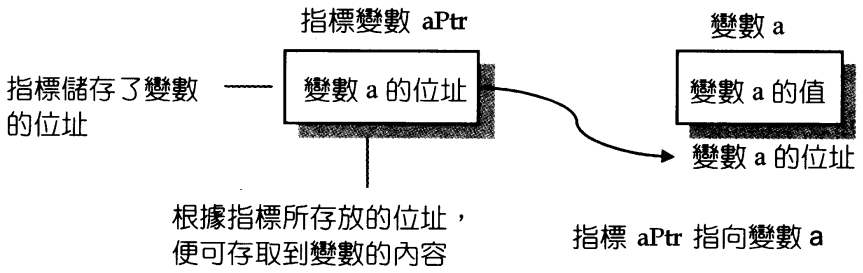
```
*yPtr = 7;  // it also changes y to 7 //
```

  - Dereferenced pointer (operand of *) must be an lvalue (no constants)
- * and & are inverses
  - & 是取記憶體位置，* 則是取某記憶體位置所存放的值，所以：
  - They cancel each other out

Using the & And * Operators

fig07_04.c

```
1  /* Fig. 7.4: fig07_04.c
2     Using the & and * operators */
3  #include <stdio.h>
4
5  int main()
6  {
7     int a;        /* a is an integer */
8     int *aPtr;    /* aPtr is a pointer to an integer */
9
10    a = 7;
11    aPtr = &a;    /* aPtr set to address of a */
12
13    printf( "The address of a is %p"
14            "\nThe value of aPtr is %p", &a, aPtr );
15
16    printf( "\n\nThe value of a is %d"
17            "\nThe value of *aPtr is %d", a, *aPtr );
18
19    printf( "\n\nShowing that * and & are complements of "
20            "each other\n&*aPtr = %p"
21            "\n*&aPtr = %p\n", &*aPtr, *&aPtr );
22
23    return 0; /* indicates successful termination */
24
25 } /* end main */
```



若在宣告時同時定義則用
`int a = 7;`
`int *aPtr = &a;`

The address of `a` is the value of `aPtr`. 注意印地址用 %p

The `*` operator returns an alias to what its operand points to. `aPtr` points to `a`, so `*aPtr` returns `a`.

Notice how `*` and `&` are inverses

```
The address of a is 0012FF7C
The value of aPtr is 0012FF7C

The value of a is 7
The value of *aPtr is 7

Showing that * and & are complements of each other.
&*aPtr = 0012FF7C
*&aPtr = 0012FF7C
```

Program Output

Pointer Operators

Operators	Associativity	Type
() []	left to right	highest
+ - ++ -- ! * & (*type*)	right to left	unary
* / %	left to right	multiplicative
+ -	left to right	additive
< <= > s>=	left to right	relational
== !=	left to right	equality
&&	left to right	logical AND
||	left to right	logical OR
?:	right to left	conditional
= += -= *= /= %=	right to left	assignment
,	left to right	comma

Fig. 7.5 | Operator precedence and associativity.

指標練習

```
/* 指標的操作練習 testPointer.c */
#include <stdio.h>

int main()
{
   int a = 5, b = 10;
   int *ptr1, *ptr2;
   ptr1  = &a;                  /* 將ptr1設為a的位址 */
   ptr2  = &b;                  /* 將ptr2設為b的位址 */
   *ptr1 = 7;                   /* 將ptr1指向的內容設為7 */
   *ptr2 = 32;                  /* 將ptr2指向的內容設為32 */
   a     = 17;                  /* 設定a為17 */
   ptr1  = ptr2;                /* 設定ptr1=ptr2 */
   *ptr1 = 9;                   /* 將ptr1指向的內容設為9 */
   ptr1  = &a;                  /* 將ptr1設為a的位址 */
   a     = 64;                  /* 設定a為64 */
   *ptr2 = *ptr1 + 5;           /* 將ptr2指向的內容設為*ptr1+5*/
   ptr2  = &a;                  /* 將ptr2設為a的位址 */

   printf("a=%2d, b=%2d, *ptr1=%2d, *ptr2=%2d\n",a,b,*ptr1,*ptr2);
   printf("ptr1=%p, ptr2=%p\n",ptr1,ptr2);

   return 0;
}
```

指標練習

執行時，變數變化的情形（&a=FF6C，&b= FF68）

行號	程式碼	a	b	ptr1	*ptr1	ptr2	*ptr2
06	int a=5,b=10;	5	10				
07	int *ptr1,*ptr2;	5	10	殘值	殘值	殘值	殘值
08	ptr1=&a;	5	10	FF6C	5	殘值	殘值
09	ptr2=&b;	5	10	FF6C	5	FF68	10
10	*ptr1=7;	7	10	FF6C	7	FF68	10
11	*ptr2=32;	7	32	FF6C	7	FF68	32
12	a=17;	17	32	FF6C	17	FF68	32
13	ptr1=ptr2;	17	32	FF68	32	FF68	32
14	*ptr1=9;	17	9	FF68	9	FF68	9
15	ptr1=&a;	17	9	FF6C	17	FF68	9
16	a=64;	64	9	FF6C	64	FF68	9
17	*ptr2=*ptr1+5;	64	69	FF6C	64	FF68	69
18	ptr2=&a;	64	69	FF6C	64	FF6C	64

指標練習

	程式碼	a	b	ptr	*ptr
1	int a = 12, b = 7;	12	7		
2	int *ptr;	12	7		
3	ptr = &a;	12	7	FF7C	12
4	*ptr = 19;	19	7	FF7C	19
5	ptr = &b;	19	7	FF78	7
6	b = 16	19	16	FF78	16
7	*ptr = 12;	19	12	FF78	12
8	a = 17;	17	12	FF78	12
9	ptr = &a;	17	12	FF7C	17
10	a = b;	12	12	FF7C	12
11	*ptr = 63;	63	12	FF7C	63

Call by *Value* - Example

```
/* 函數的傳值機制 借自 C 語言教學手冊*/

#include <stdio.h>

void add10(int,int);          /* add10()的原型 */

int main()
{
   int a = 3, b = 5;          /* 宣告區域變數a與b */

   printf ( "呼叫函數add10()之前: " );
   printf ( "a = %2d, b = %2d\n", a, b ); /* 印出a、b的值 */

   add10(a,b);

   printf ( "呼叫函數add10()之後: " );
   printf ( "a = %2d, b = %2d\n", a, b ); /* 印出a、b的值 */

   return 0;
}

void add10(int a,int b)
{
   a = a + 10;                /* 將變數a的值加10之後，設回給a */
   b = b + 10;                /* 將變數b的值加10之後，設回給b */

   printf ( "函數 add10 中 :      " );
   printf ( "a = %2d, b = %2d\n", a, b ); /* 印出a、b的值 */
}
```

```
呼叫函數add10()之前: a =  3, b =  5
函數 add10 中 :      a = 13, b = 15
呼叫函數add10()之後: a =  3, b =  5
```

Call by *Address* - Example

```
/* 函數的傳值機制 借自 C 語言教學手冊*/

#include <stdio.h>

void add10(int *,int *);     /* add10()的原型 */

int main()
{
   int a = 3, b = 5;         /* 宣告區域變數a與b */

   printf ( "呼叫函數add10()之前: " );
   printf ( "a = %2d, b = %2d\n", a, b ); /* 印出a、b的值 */

   add10(&a,&b);

   printf ( "呼叫函數add10()之後: " );
   printf ( "a = %2d, b = %2d\n", a, b ); /* 印出a、b的值 */

   return 0;
}

void add10(int *a,int *b)
{
   *a = *a + 10;
   *b = *b + 10;

   printf ( "函數 add10 中 :      " );
   printf ( "*a = %2d, *b = %2d\n  ", *a, *b ); /* 印出*a、*b的值 */
}
```

```
呼叫函數add10()之前: a =  3, b =  5
函數 add10 中 :      *a = 13, *b = 15
呼叫函數add10()之後: a = 13, b = 15
```

Calling Functions by Reference

- Call by Reference with Pointer Arguments
  - 指標可以在函數間傳遞，也可從函數傳回指標。
  - 呼叫函數時**在函數引數 (argument)前加 &** 即可將該變數的位址傳到函數中。
  - 在函數中接收的引數前加 ***** (即宣告為指標函數) ，就可接受該變數在記憶體位址中之值；
  - 在函數中如果該指標函數所代表的值改變時，表示它在記憶體位址的值也改變了，這時該值就可在函數間傳遞。
  - 在前一章已經提過，如果引數為陣列的話，就不需要加 **&** ；因為陣列名稱本身就是指標了。
- ***** Operator
  - Used as alias/nickname for variable ins[illegible]function

```
void double( int *number )
 {
  *number = 2 * ( *number );
 }
```

  - ***number** used as nickname for the variable passed

宣告 number 是個指標變數，並接收記憶體地址

*number 就是該位址所存的值

Cube a Variable using Call-by-Value

fig07_06.c

```c
/* Fig. 7.6: fig07_06.c
   Cube a variable using call-by-value */
#include <stdio.h>

int cubeByValue( int n ); /* prototype */

int main()
{
   int number = 5; /* initiali

   printf( "The original value of number is %d", number );

   /* pass number by value to cubeByValue */
   number = cubeByValue( number );

   printf( "\nThe new value of number is %d\n", number );

   return 0; /* indicates successful termination */

} /* end main */

/* calculate and return cube of integer argument */
int cubeByValue( int n )
{
   return n * n * n;   /* cube local variable n and return result */

} /* end function cubeByValue */
```

如果改成
number1 = cubeByValue(number);
時，主程式中 **number** 會改變嗎？ **NO!**

```
The original value of number is 5
The new value of number is 125
```

Program Output

由此例可知 C 語言中的函數若用 call-by-value 傳值時，一次只能傳遞一個變數的值，相當不方便！

fig07_07.c

```c
/* Fig. 7.7: fig07_07.c
   Cube a variable using call-by-reference with a pointer argument */

#include <stdio.h>

void cubeByReference( int *nPtr ); /* prototype */

int main()
{
   int number = 5; /* initialize number */

   printf( "The original value of number is %d", number );

   /* pass address of number to cubeByReference */
   cubeByReference( &number );

   printf( "\nThe new value of number is %d\n", number );

   return 0; /* indicates successful termination */

} /* end main */

/* calculate cube of *nPtr; modifies variable number in main */
void cubeByReference( int *nPtr )
{
   *nPtr = *nPtr * *nPtr * *nPtr;  /* cube *nPtr */
} /* end function cubeByReference */
```

Notice that the function prototype takes a pointer to an integer.

在此把 number 這個變數的記憶體位址傳到函數 cubeByReference 中，函數接收的引數必須為 pointer (an address of a variable) 。

因此在此處宣告 **nPtr** 為 pointer (**int *nPtr**) 來接收該位址的值，在函數中就用 ***nPtr** 進行計算。計算完後，該記憶位址的值已經更改了，因為該位址是主程式中 **number** 的位址，所以主程式中 **number** 的值也更改了。

```
The original value of number is 5
The new value of number is 125
```

Program Output

由此例可知 C 語言中的函數若用 call-by-reference 傳值時，可以同時傳遞數個變數的值。

Step 1: Before `main` calls `cubeByValue`:

```
int main( void )
{
   int number = 5;

   number = cubeByValue( number );
}
```

number: 5

```
int cubeByValue( int n )
{
   return n * n * n;
}
```

n: undefined

Step 2: After `cubeByValue` receives the call:

```
int main( void )
{
   int number = 5;

   number = cubeByValue( number );
}
```

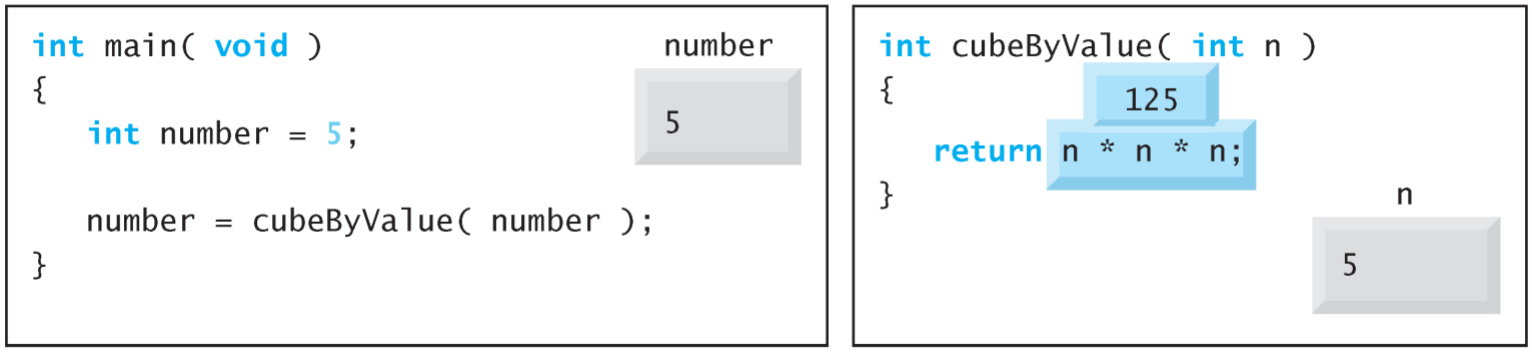
number: 5

```
int cubeByValue( int n )
{
   return n * n * n;
}
```

n: 5

Fig. 7.8 | Analysis of a typical call-by-value. (Part 1 of 3.)

Step 3: After `cubeByValue` cubes parameter `n` and before `cubeByValue` returns to `main`:



Step 4: After `cubeByValue` returns to `main` and before assigning the result to `number`:

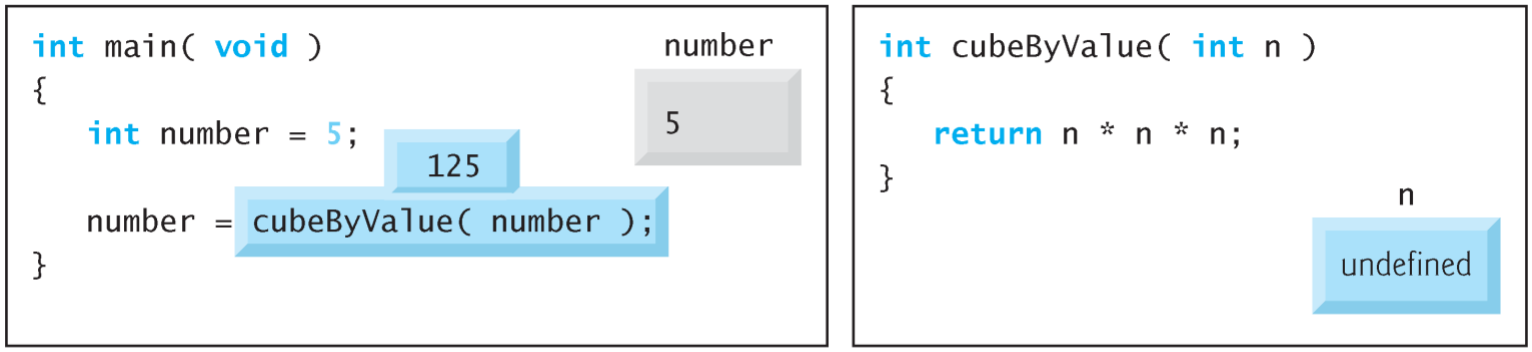


Fig. 7.8 | Analysis of a typical call-by-value. (Part 2 of 3.)

Step 5: After `main` completes the assignment to `number`:

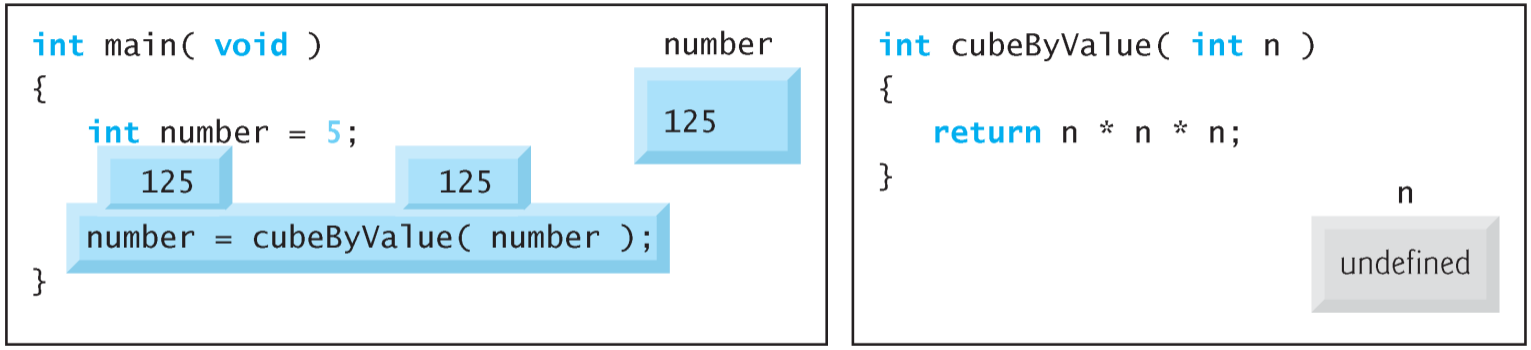


Fig. 7.8 | Analysis of a typical call-by-value. (Part 3 of 3.)

Step 1: Before `main` calls `cubeByReference`:

```
int main( void )
{
   int number = 5;

   cubeByReference( &number );
}
```

number: 5

```
void cubeByReference( int *nPtr )
{
   *nPtr = *nPtr * *nPtr * *nPtr;
}
```

nPtr: undefined

Step 2: After `cubeByReference` receives the call and before `*nPtr` is cubed:

```
int main( void )
{
   int number = 5;

   cubeByReference( &number );
}
```

number: 5

```
void cubeByReference( int *nPtr )
{
   *nPtr = *nPtr * *nPtr * *nPtr;
}
```

call establishes this pointer

nPtr

Step 3: After `*nPtr` is cubed and before program control returns to `main`:

```
int main( void )
{
   int number = 5;

   cubeByReference( &number );
}
```

number: 125

```
void cubeByReference( int *nPtr )
{
   *nPtr = *nPtr * *nPtr * *nPtr;
}
```

125

called function modifies caller's variable

nPtr

Fig. 7.9 | Analysis of a typical call-by-reference with a pointer argument.

`const` Qualifier

- Variable cannot be changed (變數的值不能再改變，即成為常數)
- Use **`const`** if function does not need to change a variable
- Attempting to change a **`const`** variable produces an error

Type Qualifier - **const**

```
1  /* Fig. 6.14: fig06_14.c
2     Demonstrating the const type qualifier with arrays */
3  #include <stdio.h>
4
5  void tryToModifyArray( const int b[] ); /* function prototype */
6
7  /* function main begins program execution */
8  int main()
9  {
10    int a[] = { 10, 20, 30 }; /* initialize a */
11
12    tryToModifyArray( a );
13
14    printf("%d %d %d\n", a[ 0 ], a[ 1 ], a[ 2 ] );
15
16    return 0; /* indicates successful termination */
17
18 } /* end main */
19
```

當宣告變數或陣列時使用 const 這個**修飾詞**後，該變數或陣列就宣告成為常數或常數陣列，它們的內容在程式執行時就不容更改！

當主程式呼叫函式時，如果函式的引數(arguments)是利用陣列傳遞時，只要在函式中陣列的值改變的話，主程式中對應的陣列值也會跟著改變。如果要避免主程式的陣列值改變，這時在函式中可宣告該陣列為常數陣列。

```
20 /* in function tryToModifyArray, array b is const, so it cannot be
21    used to modify the original array a in main. */
22 void tryToModifyArray( const int b[] )
23 {
24    b[ 0 ] /= 2;     /* error */
25    b[ 1 ] /= 2;     /* error */
26    b[ 2 ] /= 2;     /* error */
27 } /* end function tryToModifyArray */
```

```
Compiling...
FIG06_14.C
fig06_14.c(24) : error C2166: l-value specifies const object
fig06_14.c(25) : error C2166: l-value specifies const object
fig06_14.c(26) : error C2166: l-value specifies const object
```

在本函數中， b 已經宣告型態為 const int 的常數型整數陣列，但在程式中又試圖改變 b的內容，因此產生語法錯誤！

Using the `const` Qualifier with Pointers

A `const` pointer points to a ***constant memory location*** (所指地址不能再改變！), which must be initialized when defined (在宣告時就必須給值，也就是某個變數的地址)

- **`int *const myPtr = &x;`**
  - Read from right to left: "myPtr is a constant pointer which points to an integer."
  - Type **`int *const`** – constant pointer to an **`int`**
  - 此處 myPtr 為 const pointer，所以 myPtr (所指向的地址) 不能改變！
  - 故若程式再出現 myPtr = &y時，在編譯時就會出現錯誤。
- **`const int *myPtr = &x;`**
  - "myPtr is a regular pointer which points to a constant integer."
  - 此處 *myPtr (所指向的數值) 不能改變，但 myPtr 為一般 pointer
  - 所以 x 的值不能經由 pointer 改變， *myPtr = 7 在編譯時就會出現錯誤。
  - 但 x 值可改成另外的值，如 x = x+10;
- **`const int *const myPtr = &x;`**
  - “myPtr is a constant pointer which points to a constant integer."
  - 此處 myPtr 與 *myPtr 都不能改變！
  - **`x`** can be changed, but not **`*myPtr`**
  - 所以在程式中若寫 `x = 7;` 時，*myPtr 也跟著變為 7 ；但不可寫成 *myPtr = 7;

Converting Lowercase Letters to Uppercase Letters Using a Non-constant pointer to Non-constant Data

fig07_10.c (Part 1 of 2)

```
1  /* Fig. 7.10: fig07_10.c
2     Converting lowercase letters to uppercase letters
3     using a non-constant pointer to non-constant data */
4
5  #include <stdio.h>
6  #include <ctype.h>
7
8  void convertToUppercase( char *sPtr ); /* prototype */
9
10 int main()
11 {
12    char string[] = "characters and $32.98"; /* initialize char
13
14    printf( "The string before conversion is: %s", string );
15    convertToUppercase( string );
16    printf( "\nThe string after conversion is: %s\n", string );
17
18    return 0; /* indicates successful termination */
19
20 } /* end main */
21
```

本程式是把字串中小寫字元改成大寫字元，程式中會用到兩個與字元相關的內建函數：**islower, toupper** 故需要加 **#include <ctype.h>**

記得 string 是字元陣列，所以作為呼叫函數的引數時不需要加 & ，就已經屬於傳址呼叫！

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <ctype.h>
```

```
22 /* convert string to uppercase letters */
23 void convertToUppercase( char *sPtr )
24 {
25    while ( *sPtr != '\0' ) { /* current character is not '\0' */
26
27       if ( islower( *sPtr ) ) {    /* if character is lowercase, */
28          *sPtr = toupper( *sPtr ); /* convert to uppercase */
29       } /* end if */
30
31       ++sPtr;  /* move sPtr to the next character */
32    } /* end while */
33
34 } /* end function convertToUppercase */
```

++sPtr：更改到下一個字元地址

此處有兩個處理字元的函數：(需要 **#include <ctype.h>**)
islower
toupper

當本函數接受呼叫時，傳進來的引數是主程式中 string 陣列第一個元素的地址，所以在本函數的引數為「宣告 sPtr 為字元型態的指標變數」，亦即「char *sPtr」；此時， sPtr 拿到的是主程式中 string 陣列第一個元素的地址，而程式內 *sPtr 就是該地址所存的字元 (也就是「c」「h」「a」等) 。

```
The string before conversion is: characters and $32.98
The string after conversion is: CHARACTERS AND $32.98
```

Printing a String One Character at a Time Using a Non-Constant Pointer to a Constant Data

```
1  /* Fig. 7.11: fig07_11.c
2     Printing a string one character at a time using
3     a non-constant pointer to constant data */
4
5  #include <stdio.h>
6
7  void printCharacters( const char *sPtr );
8
9  int main()
10 {
11    /* initialize char array */
12    char string[] = "print characters of a string";
13
14    printf( "The string is:\n" );
15    printCharacters( string );
16    printf( "\n" );
17
18    return 0; /* indicates successful termination */
19
20 } /* end main */
21
```

fig07_11.c (Part 1 of 2)

```
22 /* sPtr cannot modify the character to which it points,
23    i.e., sPtr is a "read-only" pointer */
24 void printCharacters( const char *sPtr )
25 {
26    /* loop through entire string */
27    for ( ; *sPtr != '\0'; sPtr++ ) { /* no initialization */
28       printf( "%c", *sPtr );
29    } /* end for */
30
31 } /* end function printCharacters */
```

在此函數中已宣告 sPtr 為一般指標變數，指向到一個 const char，顯然 *sPtr 不可以改變！也就是 在 sPtr 地址的字元 (*sPtr) 不可以改變。

fig07_11.c (Part 2 of 2)

如果程式第一行改成
void printCharacters(char *const sPtr)
會產生什麼結果？

```
The string is:
print characters of a string
```

Program Output

Attempting to Modify Data Through a Non-Constant Pointer to Constant Data

fig07_12.c

```
1  /* Fig. 7.12: fig07_12.c
2     Attempting to modify data through a
3     non-constant pointer to constant data. */
4  #include <stdio.h>
5
6  void f( const int *xPtr ); /* prototype */
7
8  int main()
9  {
10    int y=50;        /* define y */
11
12    f( &y );      /* f attempts illegal modification */
13
14    return 0;    /* indicates successful termination */
15
16 } /* end main */
17
18 /* xPtr cannot be used to modify the
19    value of the variable to which it points */
20 void f( const int *xPtr )
21 {
22    *xPtr = 100;  /* error: cannot modify a const object */
23 } /* end function f */
```

在此函數中已宣告 xPtr 為一般指標變數，指向到一個 const int，因此 *xPtr 不可以改變！故在程式中 *xPtr = 100 很明顯是錯誤的！

```
Compiling...
FIG07_12.c
d:\books\2003\chtp4\examples\ch07\fig07_12.c(22) : error C2166: l-value
   specifies const object
Error executing cl.exe.

FIG07_12.exe - 1 error(s), 0 warning(s)
```

Program Output

Attempting to Modify a Constant Pointer to Nonconstant Data

fig07_13.c

```
1  /* Fig. 7.13: fig07_13.c
2     Attempting to modify a constant pointer to non-constant data */
3  #include <stdio.h>
4
5  int main()
6  {
7     int x; /* define x */
8     int y; /* define y */
9
10    /* ptr is a constant pointer to an integer that can be modified
11       through ptr, but ptr always points to the same memory location */
12    int *const ptr = &x;
13
14    *ptr = 7; /* allowed: *ptr is not const */
15    ptr = &y; /* error: ptr is const; cannot assign new address */
16
17    return 0; /* indicates successful termination */
18
19 } /* end main */
```

宣告 "ptr is a constant pointer which points to a integer." Changing *ptr is allowed – x doesn't need to be a constant.

Changing ptr is an error – ptr is a constant pointer.

Program Output

```
Compiling...
FIG07_13.c
D:\books\2003\chtp4\Examples\ch07\FIG07_13.c(15) : error C2166: l-value
   specifies const object
Error executing cl.exe.

FIG07_13.exe - 1 error(s), 0 warning(s)
```

Attempting to Modify a Constant Pointer to Constant Data

fig07_14.c

```
1  /* Fig. 7.14: fig07_14.c
2     Attempting to modify a constant pointer to constant data. */
3  #include <stdio.h>
4
5  int main()
6  {
7     int x = 5; /* initialize x */
8     int y;     /* define y */
9
10    /* ptr is a constant pointer to a constant integer. ptr always
11       points to the same location; the integer at that location
12       cannot be modified */
13    const int *const ptr = &x;
14
15    printf( "%d\n", *ptr );
16
17    *ptr = 7; /* error: *ptr is const; cannot assign new value */
18    ptr = &y; /* error: ptr is const; cannot assign new address */
19
20    return 0; /* indicates successful termination */
21
22 } /* end main */
```

宣告 "ptr is a constant pointer which points to a constant integer." Changing either *ptr or ptr is not allowed.

```
Compiling...
FIG07_14.c
D:\books\2003\chtp4\Examples\ch07\FIG07_14.c(17) : error C2166: l-value
   specifies const object
D:\books\2003\chtp4\Examples\ch07\FIG07_14.c(18) : error C2166: l-value
   specifies const object
Error executing cl.exe.

FIG07_12.exe - 2 error(s), 0 warning(s)
```

Program Output

Questions

```
#include <stdio.h>

int main()
{
   int x = 5; /* initialize x */

   int *const ptr = &x;

   printf( "%d       %p\n", *ptr, ptr );

   *ptr = 7;

   return 0;

}
```

```
Linking...


fig07_14.exe - 0 error(s), 0 warning(s)
```

Questions

```
#include <stdio.h>

int main()
{
   int x = 5 , y ;

   int *const ptr = &x;

   printf( "%d       %p\n", *ptr, ptr );

   *ptr = 7;
   ptr = &y;

   return 0;

}
```

```
Compiling...
fig07_14.c
m:\fig07_14.c(12) : error C2166: l-value specifies const object
Error executing cl.exe.


fig07_14.exe - 1 error(s), 0 warning(s)
```

Questions

```
#include <stdio.h>

int main()
{
   int x = 5 , y ;

   const int *const ptr = &x;

   printf( "%d       %p\n", *ptr, ptr );

   *ptr = 7;
   ptr = &y;

   return 0;

}
```

```
fig07_14.c
m:\fig07_14.c(11) : error C2166: l-value specifies const object
m:\fig07_14.c(12) : error C2166: l-value specifies const object
Error executing cl.exe.
```

Questions

```
#include <stdio.h>

int main()
{
   int x = 5 ;

   const int *const ptr = &x;

   x = 7;
   printf( "%d       %p\n", *ptr, ptr );

   return 0;

}
```

```
Linking...
fig07_14.exe - 0 error(s), 0 warning(s)
7       0012FF7C
Press any key to continue
```

Bubble Sort Using Call-by-reference

- Implement Bubble Sort Using Pointers
  - Swap two elements
  - **`swap`** function must receive address (using &) of array elements
    - Array elements have call-by-value default
  - Using pointers and the * operator, **`swap`** can switch array elements
- Pseudocode

Initialize array
print data in original order
Call function bubblesort
print sorted array
Define bubblesort

Function bubbleSort with Call-by-Value - Review

```
135void bubbleSort( int a[] )
136{
137   int pass; /* counter */
138   int j;    /* counter */
139   int hold; /* temporary location used to swap elements */
140
141   /* loop to control number of passes */
142   for ( pass = 1; pass < SIZE; pass++ ) {
143
144      /* loop to control number of comparisons per pass */
145      for ( j = 0; j < SIZE - 1; j++ ) {
146
147         /* swap elements if out of order */
148         if ( a[ j ] > a[ j + 1 ] ) {
149            hold = a[ j ];
150            a[ j ] = a[ j + 1 ];
151            a[ j + 1 ] = hold;
152         } /* end if */
153
154      } /* end inner for */
155
156   } /* end outer for */
157
158} /* end function bubbleSort */
159
154      } /* end inner for */
```

Bubble Sort with Call-by-Reference

```c
/* Fig. 7.15: fig07_15.c
   This program puts values into an array, sorts the values into
   ascending order, and prints the resulting array. */
#include <stdio.h>
#define SIZE 10

void bubbleSort( int *array, const int size ); /* prototype */

int main()
{
   /* initialize array a */
   int a[ SIZE ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };

   int i; /* counter */

   printf( "Data items in original order\n" );

   /* loop through array a */
   for ( i = 0; i < SIZE; i++ ) {
      printf( "%4d", a[ i ] );
   } /* end for */

   bubbleSort( a, SIZE ); /* sort the array */

   printf( "\nData items in ascending order\n" );
```

fig07_15.c (Part 1 of 3)

```
27    /* loop through array a */
28    for ( i = 0; i < SIZE; i++ ) {
29       printf( "%4d", a[ i ] );
30    } /* end for */
31
32    printf( "\n" );
33
34    return 0; /* indicates successful termination */
35
36 } /* end main */
```

fig07_15.c (Part 2 of 3)

```
38 /* sort an array of integers using bubble sort algorithm */
39 void bubbleSort( int *array, const int size )
40 {
41    void swap( int *element1Ptr, int *element2Ptr ); /* prototype */
42    int pass; /* pass counter */
43    int j;    /* comparison counter */
44
45    /* loop to control passes */
46    for ( pass = 0; pass < size - 1; pass++ ) {
47
48       /* loop to control comparisons during each pass */
49       for ( j = 0; j < size - 1; j++ ) {
50
51          /* swap adjacent elements if they are out of order */
52          if ( array[ j ] > array[ j + 1 ] ) {
53             swap( &array[ j ], &array[ j + 1 ] );
54          } /* end if */
55
56       } /* end inner for */
57
58    } /* end outer for */
59
60 } /* end function bubbleSort */
```

注意在這裡我們並未定義 **array** 為陣列 (事實上 **array** 為指標變數)，但為什麼能夠用 **array[j]** 與 **array[j+1]** ？這種利用指標變數找陣列元素的方法將在後面 (Sec. 7.9) 會詳加說明。簡單的說，就是陣列的名稱就是陣列的地址，指標變數所存的值也是地址，因此當 **array** 存的是陣列 **a** 的地址時，**array[3]** 和 **a[3]** 是同樣的意思。

```
61
62 /* swap values at memory locations to which element1Ptr and
63    element2Ptr point */
64 void swap( int *element1Ptr, int *element2Ptr )
65 {
66    int hold = *element1Ptr;
67    *element1Ptr = *element2Ptr;
68    *element2Ptr = hold;
69 } /* end function swap */
```

fig07_15.c (Part 3 of 3)

傳址呼叫後，將函數的引數 ***element1Ptr*** 與 ***element2Ptr*** 之值交換；same as

```
hold = a[ j ];
a[ j ] = a[ j + 1 ];
a[ j + 1 ] = hold;
```

Program Output

```
Data items in original order
   2   6   4   8  10  12  89  68  45  37
Data items in ascending order
   2   4   6   8  10  12  37  45  68  89
```

Function sizeof

- **sizeof**
  - Returns size of operand in bytes
  - 1 byte = 8 bits
  - For arrays: (size of 1 element) × (number of elements)
  - 用法：**sizeof**(變數型態) ，或 **sizeof** 變數名稱，或 **sizeof(**變數名稱)
  - if **sizeof(int)** equals **4** bytes, then

```
int myArray[ 10 ];
printf( "%d", sizeof( myArray ) );
```

    - will print 40
- **sizeof** can be used with
  - Variable names
  - Type name
  - Constant values

Typical Size and Range of Data Types

For Borland Compiler

Data Type	Size Bytes	Min Value	Max Value
char	1	-128	127
short int	2	-32768	32767
int	4	-2147483648	2147483647
long int	4	-2147483648	2147483647
float	4	1.17549e-38	3.40282e+38
double	8	2.22507e-308	1.79769e+308
long double	10	3.3621e-4932	1.18973e+4932

For Visual C++ and C Compiler

Data Type	Size Bytes	Min Value	Max Value
char	1	-128	127
short int	2	-32768	32767
int	4	-2147483648	2147483647
long int	4	-2147483648	2147483647
float	4	1.17549e-38	3.40282e+38
double	8	2.22507e-308	1.79769e+308
long double	8	2.22507e-308	1.79769e+308

```
/* SIZEOF.C--Program to tell the size of the C variable */
/*          type in bytes */

#include <stdio.h>

main()
{

    printf( "\nA char             is %d bytes", sizeof( char ));
    printf( "\nAn int             is %d bytes", sizeof( int ));
    printf( "\nA short            is %d bytes", sizeof( short ));
    printf( "\nA long             is %d bytes", sizeof( long ));
    printf( "\nAn unsigned char   is %d bytes", sizeof( unsigned char ));
    printf( "\nAn unsigned int    is %d bytes", sizeof( unsigned int ));
    printf( "\nAn unsigned short  is %d bytes", sizeof( unsigned short ));
    printf( "\nAn unsigned long   is %d bytes", sizeof( unsigned long ));
    printf( "\nA float            is %d bytes", sizeof( float ));
    printf( "\nA double           is %d bytes", sizeof( double ));
    printf( "\nA long double      is %d bytes\n", sizeof( long double ));

    return 0;
}
```

```
A char             is 1  bytes
An int             is 4  bytes
A short            is 2  bytes
A long             is 4  bytes
An unsigned char   is 1  bytes
An unsigned int    is 4  bytes
An unsigned short  is 2  bytes
An unsigned long   is 4  bytes
A float            is 4  bytes
A double           is 8  bytes
A long double      is 8  bytes  -  for Visual C++ Compiler
A long double      is 10  bytes -  for Borland Compiler
```

Operator `sizeof` When Applied to an Array name Returns the Number of Bytes in the Array

fig07_16.c

```
1  /* Fig. 7.16: fig07_16.c
2     Sizeof operator when used on an array name
3     returns the number of bytes in the array. */
4  #include <stdio.h>
5
6  size_t getSize( float *ptr ); /* prototype */
7
8  int main()
9  {
10    float array[ 20 ]; /* create array */
11
12    printf( "The number of bytes in the array is %d"
13            "\nThe number of bytes returned by getSize is %d\n",
14            sizeof( array ), getSize( array ) );
15
16    return 0; /* indicates successful termination */
17
18 } /* end main */
19
20 /* return size of ptr */
21 size_t getSize( float *ptr )
22 {
23    return sizeof( ptr );
24
25 } /* end function getSize */
```

此處使用 size_t 當成 sizeof 函數取得的值的型態；size_t 在 C 語言中內定為無號整數 (unsigned or unsigned long) 型態，在此也可把 size_t 改成 int

```
The number of bytes in the array is 80
The number of bytes returned by getSize is 4
```

```c
/* Fig. 7.17: fig07_17.c
   Demonstrating the sizeof operator */
#include <stdio.h>

int main()
{
   char c;           /* define c */
   short s;          /* define s */
   int i;            /* define i */
   long l;           /* define l */
   float f;          /* define f */
   double d;         /* define d */
   long double ld;   /* define ld */
   int array[ 20 ];  /* initialize array */
   int *ptr = array; /* create pointer to array */

   printf( "    sizeof c = %d\tsizeof(char)  = %d"
           "\n    sizeof s = %d\tsizeof(short) = %d"
           "\n    sizeof i = %d\tsizeof(int) = %d"
           "\n    sizeof l = %d\tsizeof(long) = %d"
           "\n    sizeof f = %d\tsizeof(float) = %d"
           "\n    sizeof d = %d\tsizeof(double) = %d"
           "\n   sizeof ld = %d\tsizeof(long double) = %d"
           "\nsizeof array = %d"
           "\n  sizeof ptr = %d\n",
```

fig07_17.c (Part 1 of 2)

```
26          sizeof c, sizeof( char ), sizeof s,
27          sizeof( short ), sizeof i, sizeof( int ),
28          sizeof l, sizeof( long ), sizeof f,
29          sizeof( float ), sizeof d, sizeof( double ),
30          sizeof ld, sizeof( long double ),
31          sizeof array, sizeof ptr );
32
33    return 0; /* indicates successful termination */
34
35 } /* end main */
```

fig07_17.c (Part 2 of 2)

```
    sizeof c = 1        sizeof(char)  = 1
    sizeof s = 2        sizeof(short) = 2
    sizeof i = 4        sizeof(int) = 4
    sizeof l = 4        sizeof(long) = 4
    sizeof f = 4        sizeof(float) = 4
    sizeof d = 8        sizeof(double) = 8
   sizeof ld = 8        sizeof(long double) = 8
sizeof array = 80
  sizeof ptr = 4
```

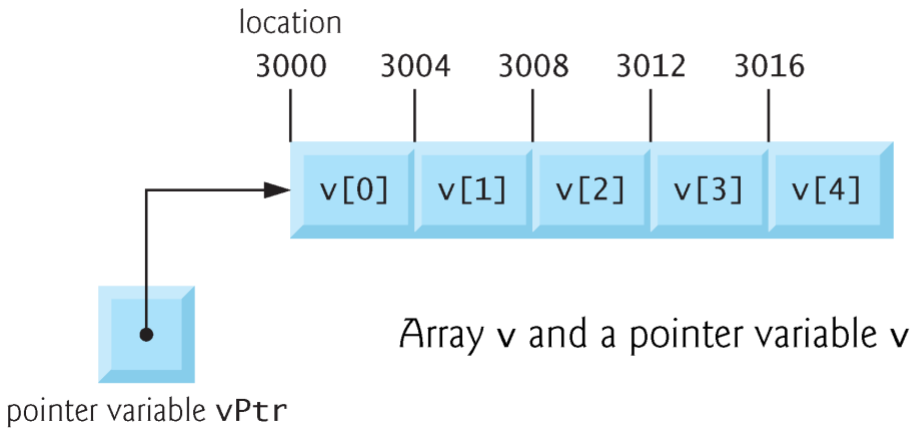
Program Output

Pointer Expressions and Pointer Arithmetic

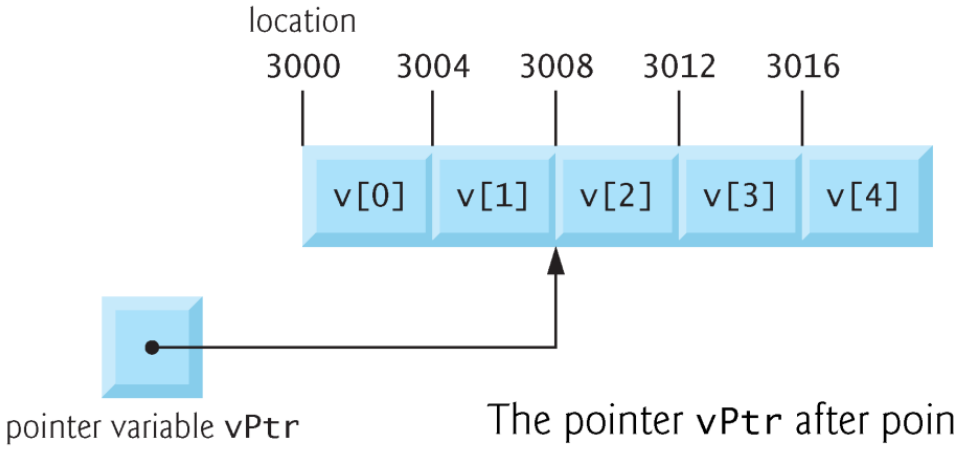
Example: 5-element **`int`** array on machine with 4 byte ints

- **`int v[ 5 ];`**
- **`int *vPtr = &v (or v or &v[0]);`**
- **`vPtr`** points to *first element* **`v[ 0 ]`**, Say at location 3000 (**`vPtr`** = 3000)
- **`vPtr += 2`**; sets **`vPtr`** to 3008(即 3000 + 2*4；並非 3002!). In other words, **`vPtr`** points to **`v[ 2 ]`** (incremented by 2), but the machine has 4 byte ints, so it points to address 3008

Pointer Expressions and Pointer Arithmetic



Array `v` and a pointer variable `vPtr` that points to `v`.



The pointer `vPtr` after pointer arithmetic.

Pointer Expressions and Pointer Arithmetic

- **Arithmetic operations** can be performed on pointers (here, vPtr)
  - `vPtr = v;` or `vPtr = &v[0];`
  - Increment/decrement pointer (++ or --)
    - `vPtr++`
  - Add an integer to a pointer(+ or += , - or -=)
    - `vPtr += 2`
  - Pointers may be subtracted from each other
    - `vPtr2 – vPtr`
    - (如果 vPtr 為3000，vPtr2 為3008，則vPtr2 – vPtr 為 2，不是8)
  - *Operations meaningless* unless performed on pointer which points to an array (如果 vPtr 不是指向陣列時，以上的運算就沒有意義)
  - Recall that 陣列的名稱即代表該陣列的開始的地址；i.e.,
    - array, &array[0], 與 &array 三者都是 array 這個陣列在記憶體中開始的位址。

Pointer Expressions and Pointer Arithmetic

- Increment/decrement pointer
  - if vPtr = 3016, which points to v[4], then **`vPtr -= 4`** would set vPtr back to 3000.
  - **`++vPtr; vPtr++; --vPtr; vPtr--;`**
- Subtracting pointers
  - Returns *number of elements from one to the other*. If

    `vPtr2 = &v[ 2 ];`

    `vPtr  = &v[ 0 ];`

    `vPtr2 - vPtr` would produce 2
- Pointer comparison (<, == , >)

  Meaningful for the pointers point to *elements of the same array.*

Pointer Expressions and Pointer Arithmetic

Pointers of the same type can be assigned to each other

- If not the same type, a cast operator, e.g., (float), (int), etc., must be used
- Exception: pointer to **`void`** (type **`void *`**)
  - **`void *bPtr;`**
  - Generic pointer, represents any type
  - No casting needed to convert a pointer to **`void`** pointer
  - **`void`** pointers cannot be dereferenced (無法依址找值，也就是說，程式中若有 **`*bPtr`** 將無法執行) since a pointer to void simply contains a memory location for an unknown data type.

Relationship Between Pointers and Arrays

- *Arrays* and *pointers* are closely related
  - Array name like a constant pointer
  - Pointers can do array subscripting operations
- Define an array **b[5]** and a pointer **bPtr**

  To set them equal to one another, we can use:

```
int b[5];
int *bPtr;
bPtr = b;
```

  - The array name, **b**, is actually the address of first element , **b[0]**, of the array **b**.

```
bPtr = &b[ 0 ] ;
```

    Explicitly assigns **bPtr** to address of first element of **b**

Relationship Between Pointers and Arrays

Element `b[ 3 ]`

1. Can be accessed by `*( bPtr + 3 )`
   - 在此 `bPtr` 即為 `b` 這個陣列第一個元素 `b[0]` 的地址；`bPtr + 3` 即為 `b[3]` 的地址。Where 3 is the offset. Called **pointer/offset notation**
   - `&b[3]` 、`b + 3` 、與 `bPtr + 3` 同義，都是 `b[3]` 的『地址』。
2. Can also be accessed by `bPtr[ 3 ]`
   - Called **pointer/subscript notation**
   - `bPtr[ 3 ]` 是 `bPtr+3` 這個地址所存的值
   - `bPtr[ 3 ]` same as `b[ 3 ]`
3. Can finally be accessed by performing pointer arithmetic on the array itself

   `*( b + 3 )`

Question: `bPtr += 3` 與 `b += 3` 是否一樣？ **Ans:** NO!

Remark: `bPtr` 是 pointer，可以改變指向的位址，但 `b` 是陣列，也是第一個元素 `b[0]` 的地址，因此 `b` 可視為 a constant pointer，不能改變 `b` 的值。

Using **Subscripting** and **Pointer** Notations with Arrays

```
1  /* Fig. 7.20: fig07_20.cpp
2     Using subscripting and pointer notations with arrays */
3
4  #include <stdio.h>
5
6  int main()
7  {
8     int b[] = { 10, 20, 30, 40 }; /* initialize array b */
9     int *bPtr = b;                /* set bPtr to point to array b */
10    int i;                        /* counter */
11    int offset;                   /* counter */
12
13    /* output array b using array subscript notation */
14    printf( "Array b printed with:\nArray subscript notation\n" );
15
16    /* loop through array b */
17    for ( i = 0; i < 4; i++ ) {
18       printf( "b[ %d ] = %d\n", i, b[ i ] );
19    } /* end for */
20
21    /* output array b using array name and pointer/offset notation */
22    printf( "\nPointer/offset notation where\n"
23            "the pointer is the array name\n" );
24
```

```
b[ 0 ] = 10
b[ 1 ] = 20
b[ 2 ] = 30
b[ 3 ] = 40
```

fig07_20.c (Part 1 of 2)

```
25    /* loop through array b */
26    for ( offset = 0; offset < 4; offset++ ) {
27       printf( "*( b + %d ) = %d\n", offset, *( b + offset ) );
28    } /* end for */
29
30    /* output array b using bPtr and array subscript notation */
31    printf( "\nPointer subscript notation\n" );
32
33    /* loop through array b */
34    for ( i = 0; i < 4; i++ ) {
35       printf( "bPtr[ %d ] = %d\n", i, bPtr[ i ] );
36    } /* end for */
37
38    /* output array b using bPtr and pointer/offset notation */
39    printf( "\nPointer/offset notation\n" );
40
41    /* loop through array b */
42    for ( offset = 0; offset < 4; offset++ ) {
43       printf( "*( bPtr + %d ) = %d\n", offset, *( bPtr + offset ) );
44    } /* end for */
45
46    return 0; /* indicates successful termi
47
48 } /* end main */
```

```
*( b + 0 ) = 10
*( b + 1 ) = 20
*( b + 2 ) = 30
*( b + 3 ) = 40
```

```
bPtr[ 0 ] = 10
bPtr[ 1 ] = 20
bPtr[ 2 ] = 30
bPtr[ 3 ] = 40
```

```
*( bPtr + 0 ) = 10
*( bPtr + 1 ) = 20
*( bPtr + 2 ) = 30
*( bPtr + 3 ) = 40
```

fig07_20.c (Part 2 of 2)

Program Output

```
Array b printed with:
Array subscript notation
b[ 0 ] = 10
b[ 1 ] = 20
b[ 2 ] = 30
b[ 3 ] = 40

Pointer/offset notation where
the pointer is the array name
*( b + 0 ) = 10
*( b + 1 ) = 20
*( b + 2 ) = 30
*( b + 3 ) = 40

Pointer subscript notation
bPtr[ 0 ] = 10
bPtr[ 1 ] = 20
bPtr[ 2 ] = 30
bPtr[ 3 ] = 40

Pointer/offset notation
*( bPtr + 0 ) = 10
*( bPtr + 1 ) = 20
*( bPtr + 2 ) = 30
*( bPtr + 3 ) = 40
```

Copying a String Using Array Notation and Pointer Notation

```
1  /* Fig. 7.21: fig07_21.c
2     Copying a string using array notation and pointer notation. */
3  #include <stdio.h>
4
5  void copy1( char *s1, const char *s2 ); /* prototype */
6  void copy2( char *s1, const char *s2 ); /* prototype */
7
8  int main()
9  {
10    char string1[ 10 ];           /* create array string1 */
11    char *string2 = "Hello";      /* create a pointer to a string */
12    char string3[ 10 ];           /* create array string3 */
13    char string4[] = "Good Bye"; /* create a pointer to a string */
14
15    copy1( string1, string2 );
16    printf( "string1 = %s\n", string1 );
17
18    copy2( string3, string4 );
19    printf( "string3 = %s\n", string3 );
20
21    return 0; /* indicates successful termination */
22
23 } /* end main */
24
```

fig07_21.c (Part 1 of 2)

在此 string2 並非陣列，而是一個指向陣列 "Hello" 的指標。

此處 'Hello" 及 string4 各含幾個元素？

在此把 string1 和 “Hello” 字串陣列的地址傳到函數 copy1 中。 此處 string2 就是“Hello” 字串陣列的地址。

```
25 /* copy s2 to s1 using array notation */
26 void copy1( char *s1, const char *s2 )
27 {
28    int i; /* counter */
29
30    /* loop through strings */
31    for ( i = 0; ( s1[ i ] = s2[ i ] ) != '\0'; i++ ) {
32       ; /* do nothing in body */
33    } /* end for */
34
35 } /* end function copy1 */
36
37 /* copy s2 to s1 using pointer notation */
38 void copy2( char *s1, const char *s2 )
39 {
40    /* loop through strings */
41    for ( ; ( *s1 = *s2 ) != '\0'; s1++, s2++ ) {
42       ; /* do nothing in body */
43    } /* end for */
44
45 } /* end function copy2 */
```

在此 s1 和 s2 都是指標變數，分別接收主函數傳來的地址。

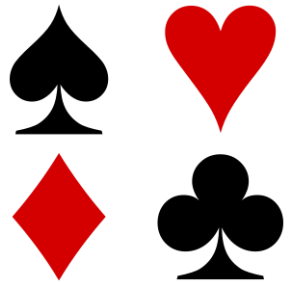
在此把儲存在 s2[i] 的字元複製到 s1[i] 中，直到字元結束(\0) 符號出現為止。

fig07_21.c (Part 2 of 2)

Program Output

```
string1 = Hello
string3 = Good Bye
```

Arrays of Pointers

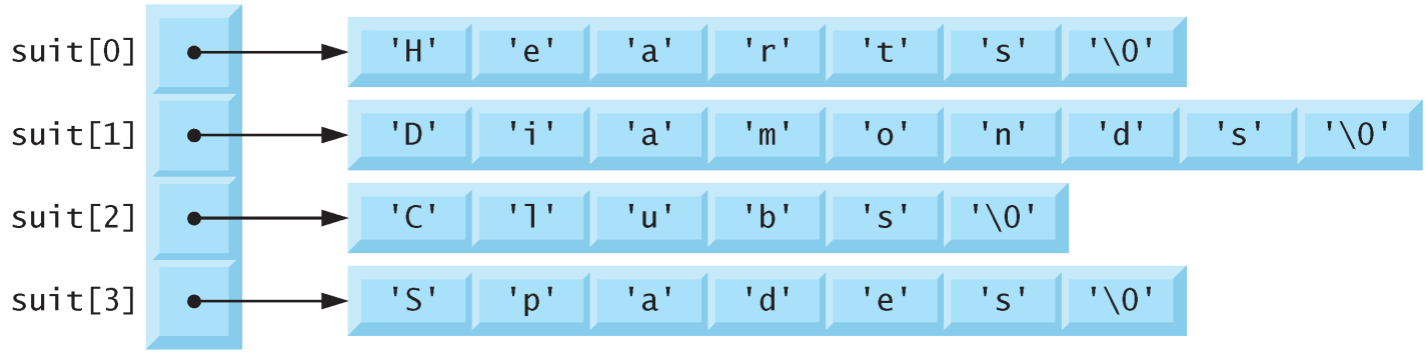


- An array can *contain pointers* (指標陣列) as its elements
- For example: an array of strings

```
const char *suit[ 4 ] = { "Hearts", "Diamonds",
   "Clubs", "Spades" };
```

  - **`suit[4]`** 為一個含四個元素的指標陣列，個別指向 const char, i.e., the strings pointed by each *element pointer* cannot be modified (const).
  - Each pointer points to the first character of a string (or an array of char).
  - The strings are not actually stored in the array **`suit`**, only *pointers to the strings* are stored in this array.

黑桃♠（Spades）、紅心♥（Hearts）、梅花♣（Clubs）、方塊♦（Diamonds）



  - **`suit`** array has a fixed size (4, in this case), but strings can be of any size

The Advantage of Pointer Arrays

The suits could have been placed into a *two-dimensional array* in which each row would represent one suit, and each column would represent one of the letters of a suit name.

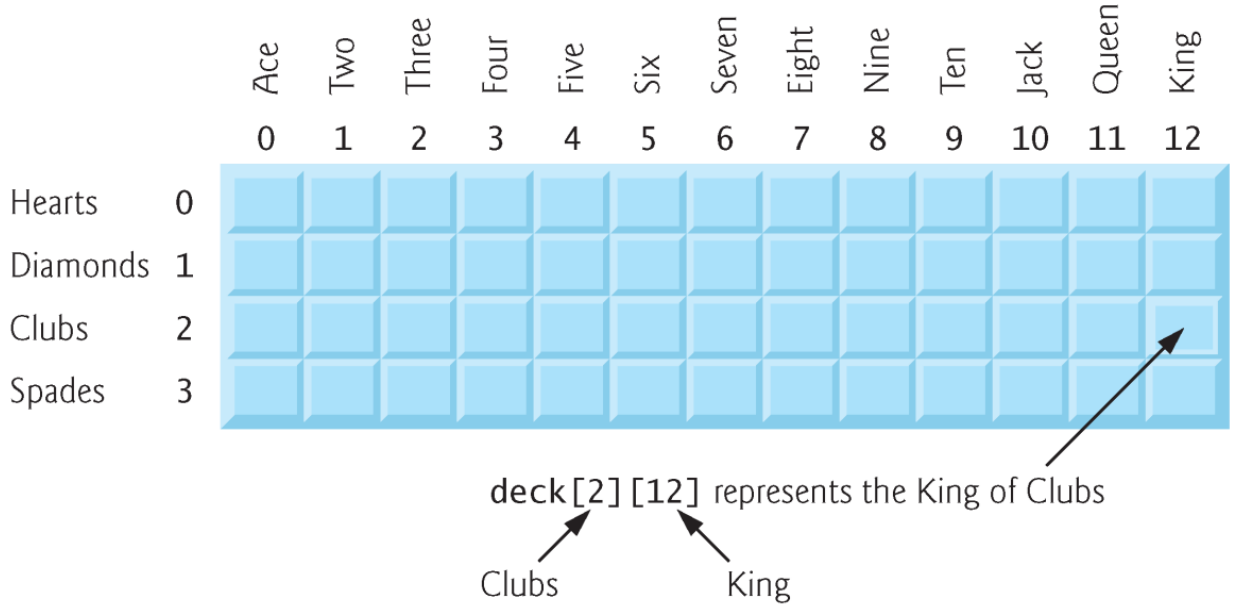
Such a data structure would have to have a *fixed number of columns per row*, and that number would have to be as large as the longest string.

Therefore, *considerable memory could be wasted* when a large number of strings being stored with most strings shorter than the longest string.

Case Study: A Card Shuffling and Dealing Simulation

Card **shuffling** program

1. Use two (2) pointer *arrays* to store strings for suits and faces
2. Use another double-scripted regular array (suit (花色), face (點數)) to represent a full deck of cards



3. The numbers 1-52 go into the array, representing the *order* in which the cards are dealt

Case Study: A Card *Shuffling* and *Dealing* Simulation

- Pseudocode
  - Top level:

    Shuffle (洗牌) and deal (發牌) 52 cards
  - First refinement:

    *Initialize the suit array (pointer array, *suit[4])*

    *Initialize the face array (pointer array, *face[13])*

    Initialize the deck array (regular array, deck[4][13])

    Shuffle the deck

    Deal 52 cards

Case Study: A Card Shuffling and Dealing Simulation

- Second refinement
  - Convert *shuffle the deck* to

    For each of the 52 cards

    Place card number (order) in randomly selected unoccupied slot of deck
  - Convert *deal 52 cards* to

    Consider each of the 52 card numbers in sequential order

    Find card number in deck array and print face and suit of card

Case Study: A Card Shuffling and Dealing Simulation

- Third refinement
  - Convert *shuffle the deck* to

    Choose slot of deck randomly

    If the chosen slot of deck has been previously chosen,

    then choose another slot of deck randomly

    If not, place card number in chosen slot of deck
  - Convert *deal 52 cards sequentially* to

    For each slot of the deck array

    If slot contains card number

    Print the face and suit of the card

```c
/* Fig. 7.24: fig07_24.c
   Card shuffling dealing program */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

/* prototypes */
void shuffle( int wDeck[][ 13 ] );
void deal( const int wDeck[][ 13 ], const char *wFace[],
           const char *wSuit[] );

int main()
{
   /* initialize suit array */
   const char *suit[ 4 ] = { "Hearts", "Diamonds", "Clubs", "Spades" };

   /* initialize face array */
   const char *face[ 13 ] =
      { "Ace", "Deuce", "Three", "Four",
        "Five", "Six", "Seven", "Eight",
        "Nine", "Ten", "Jack", "Queen", "King" };

   /* initialize deck array */
   int deck[ 4 ][ 13 ] = { 0 };

```

先將 deck 所有元素設為 0

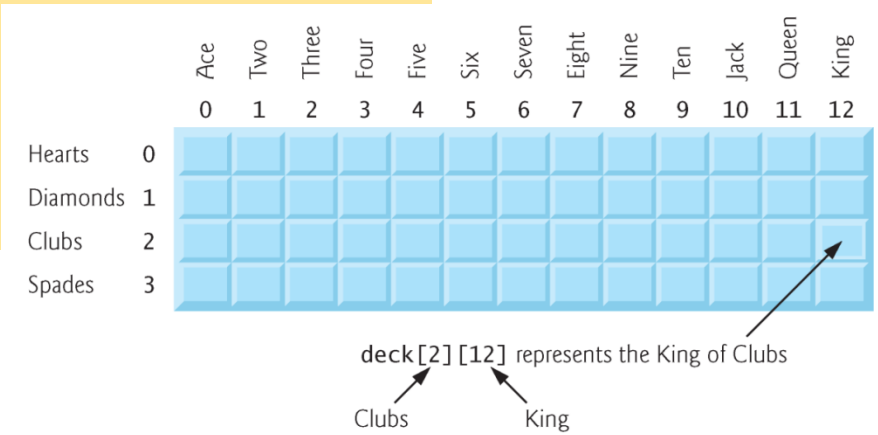


fig07_24.c (Part 1 of 4)

```
26    srand( time( 0 ) ); /* seed random-number generator */
27
28    shuffle( deck );
29    deal( deck, face, suit );
30
31    return 0; /* indicates successful termination */
32
33 } /* end main */
34
```

呼叫洗牌 shuffle 與發牌 deal 函數

fig07_24.c (Part 2 of 4)

Shuffle 洗牌程式

fig07_24.c (Part 3 of 4)

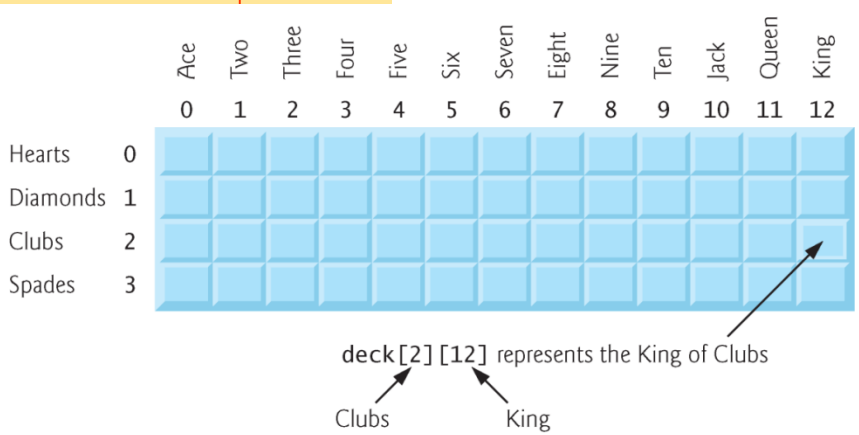
```
35 /* shuffle cards in deck */
36 void shuffle( int wDeck[][ 13 ] )
37 {
38    int row;    /* row number */
39    int column; /* column number */
40    int card;   /* counter */
41
42    /* for each of the 52 cards, choose slot of deck randomly */
43    for ( card = 1; card <= 52; card++ ) {
44
45       /* choose new random location until unoccupied slot found */
46       do {
47          row = rand() % 4;
48          column = rand() % 13;
49       } while( wDeck[ row ][ column ] != 0 ); /* end do...while */
50
51       /* place card number in chosen slot of deck */
52       wDeck[ row ][ column ] = card;
53    } /* end for */
54
55 } /* end function shuffle */
56
```

主函數中已把 wDeck 陣列的值都設成 0；

洗牌的方法就是：

取 wDeck 陣列中數值是 0 的某一個元素

再依序把 1、2、3、. . . 寫入



Deal (發牌) 程式

```
57 /* deal cards in deck */
58 void deal( const int wDeck[][ 13 ], const char *wFace[],
59            const char *wSuit[] )
60 {
61    int card;   /* card counter */
62    int row;    /* row counter */
63    int column; /* column counter */
64
65    /* deal each of the 52 cards */
66    for ( card = 1; card <= 52; card++ ) {
67
68       /* loop through rows of wDeck */
69       for ( row = 0; row <= 3; row++ ) {
70
71          /* loop through columns of wDeck for current row */
72          for ( column = 0; column <= 12; column++ ) {
73
74             /* if slot contains current card, display card */
75             if ( wDeck[ row ][ column ] == card ) {
76                printf( "%5s of %-8s%c", wFace[ column ], wSuit[ row ],
77                        card % 2 == 0 ? '\n' : '\t' );
78             } /* end if */
79
80          } /* end for */
81
82       } /* end for */
83
84    } /* end for */
85
86 } /* end function deal */
```

從第 1 張開始，發到第 52 張牌；

找第 card 出現在哪裡 (row ? And column?) ；並利用 wFace 及 wSuit 找出對應的點數及花色。

fig07_24.c (Part 4 of 4)

Program Output

```
 Nine of Hearts          Five of Clubs
Queen of Spades        Three of Spades
Queen of Hearts          Ace of Clubs
 King of Hearts          Six of Spades
 Jack of Diamonds        Five of Spades
Seven of Hearts          King of Clubs
Three of Clubs         Eight of Hearts
Three of Diamonds        Four of Diamonds
Queen of Diamonds        Five of Diamonds
  Six of Diamonds        Five of Hearts
  Ace of Spades          Six of Hearts
 Nine of Diamonds      Queen of Clubs
Eight of Spades          Nine of Clubs
Deuce of Clubs           Six of Clubs
Deuce of Spades          Jack of Clubs
 Four of Clubs         Eight of Clubs
 Four of Spades        Seven of Spades
Seven of Diamonds      Seven of Clubs
 King of Spades          Ten of Diamonds
 Jack of Hearts          Ace of Hearts
 Jack of Spades          Ten of Clubs
Eight of Diamonds      Deuce of Diamonds
  Ace of Diamonds        Nine of Spades
 Four of Hearts        Deuce of Hearts
 King of Diamonds        Ten of Spades
Three of Hearts          Ten of Hearts
```

Pointers to Functions

- Pointer to function
  - 前面已談到，**陣列的名稱**是陣列中第一個(編號為 0) 的元素在記憶體中的位置，相同的，**函數的名稱是記憶體中函數程式碼儲存的開始位置**。Similar to how array name is address of first element, function name is starting address of code that defines function
  - 指標也可以指向函數 (function) ；指向函數時，就是指向函數**程式碼**在記憶體中開始的位址。
  - Contains address of function

- *Pointer to function*就像一般的指標變數，可以：
  - Passed to functions
  - Returned from functions
  - Stored in arrays
  - Assigned to other function pointers

Pointers to Functions

Example: bubblesort

- `void bubble( int work[], const int size, int (*compare)( int a, int b ) )`
- Function **`bubble`** takes a *function pointer:* **`compare`**

  `bubble` calls this helper function which determines ascending or descending sorting
- The argument in bubblesort for the function pointer:

  `int (*compare)( int a, int b )`

  tells **`bubblesort`** to expect a pointer to a function that takes two **`int`**s and returns an **`int`** (回傳整數)

  在這裡注意 **`(*compare)`** 含括號，表示

  1. 接收一個函數的地址
  2. 放到 **'compare'** 這個函數指標

Pointers to Functions

Example: bubblesort

- `void bubble( int work[], const int size, int (*compare)( int a, int b ) )`

如果`(*compare)`沒有括號，則：

```
int *compare( int a, int b )
```

Defines a function that receives two integers and returns a **pointer** to a **`int`**，變成定義一個叫 **`compare`** 的函數，這個函數回傳的是某個整數的地址。

Multipurpose Sorting Program Using Function Pointers

```
1  /* Fig. 7.26: fig07_26.c
2     Multipurpose sorting program using function pointers */
3  #include <stdio.h>
4  #define SIZE 10
5
6  /* prototypes */
7  void bubble( int work[], const int size, int (*compare)( int a, int b ) );
8  int ascending( int a, int b );
9  int descending( int a, int b );
10
11 int main()
12 {
13    int order;   /* 1 for ascending order or 2 for descending order */
14    int counter; /* counter */
15
16    /* initialize array a */
17    int a[ SIZE ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
18
19    printf( "Enter 1 to sort in ascending order,\n"
20            "Enter 2 to sort in descending order: " );
21    scanf( "%d",  &order );
22
23    printf( "\nData items in original order\n" );
24
```

fig07_26.c (Part 1 of 4)

在這裡，程式會問排序法是要用遞增或是遞減；

order = 1 是遞增、 order = 2 是遞減。

```
25    /* output original array */
26    for ( counter = 0; counter < SIZE; counter++ ) {
27       printf( "%5d", a[ counter ] );
28    } /* end for */
29
30    /* sort array in ascending order; pass function ascending as an
31      argument to specify ascending sorting order */
32    if ( order == 1 ) {
33       bubble( a, SIZE, ascending );
34       printf( "\nData items in ascending order\n" );
35    } /* end if */
36    else { /* pass function descending */
37       bubble( a, SIZE, descending );
38       printf( "\nData items in descending order\n" );
39    } /* end else */
40
41    /* output sorted array */
42    for ( counter = 0; counter < SIZE; counter++ ) {
43       printf( "%5d", a[ counter ] );
44    } /* end for */
45
46    printf( "\n" );
47
48    return 0; /* indicates successful termination */
49
50 } /* end main */
51
```

如果是遞增的話，就呼叫 ascending 這個函數，要不然就呼叫 descending 。記得 ascending 及 descending 這兩個函數名稱也是該函數的地址。

fig07_26.c (Part 2 of 4)

fig07_26.c (Part 3 of 4)

```
52 /* multipurpose bubble sort; parameter compare is a pointer to
53    the comparison function that determines sorting order */
54 void bubble( int work[], const int size, int (*compare)( int a, int b ) )
55 {
56    int pass;  /* pass counter */
57    int count; /* comparison counter */
58
59    void swap( int *element1Ptr, int *element2ptr ); /* prototype */
60
61    /* loop to control passes */
62    for ( pass = 1; pass < size; pass++ ) {
63
64       /* loop to control number of comparisons per pass */
65       for ( count = 0; count < size - 1; count++ ) {
66
67          /* if adjacent elements are out of order, swap them */
68          if ( (*compare)( work[ count ], work[ count + 1 ] ) ) {
69             swap( &work[ count ], &work[ count + 1 ] );
70          } /* end if */
71
72       } /* end for */
73
74    } /* end for */
75
76 } /* end function bubble */
77
```

既然傳進來的是地址(函數名稱)，我們就用指標(pointer)來接收。在這裡定義 compare 是一個指標，指向某個函數 (及函數名稱) ；在不同條件下，就會呼叫不同的函數來執行。

```
78 /* swap values at memory locations to which element1Ptr and
79    element2Ptr point */
80 void swap( int *element1Ptr, int *element2Ptr )
81 {
82    int hold; /* temporary holding variable */
83
84    hold = *element1Ptr;
85    *element1Ptr = *element2Ptr;
86    *element2Ptr = hold;
87 } /* end function swap */
88
89 /* determine whether elements are out of order for an ascending
90    order sort */
91 int ascending( int a, int b )
92 {
93    return b < a; /* swap if b is less than a */
94
95 } /* end function ascending */
96
97 /* determine whether elements are out of order for a descending
98    order sort */
99 int descending( int a, int b )
100 {
101    return b > a; /* swap if b is greater than a */
102
103 } /* end function descending */
```

fig07_26.c (Part 4 of 4)

當 b < a 是對的話， return 'true'，也就是說 return 1 (ascending = 1) 回去。

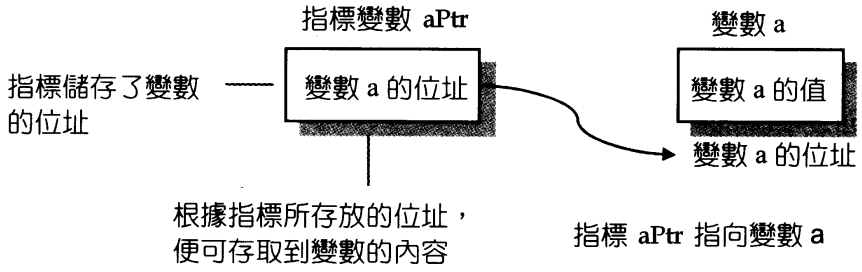
Program Output

```
Enter 1 to sort in ascending order,
Enter 2 to sort in descending order: 1

Data items in original order
    2    6    4    8   10   12   89   68   45   37
Data items in ascending order
    2    4    6    8   10   12   37   45   68   89
```

```
Enter 1 to sort in ascending order,
Enter 2 to sort in descending order: 2

Data items in original order
    2    6    4    8   10   12   89   68   45   37
Data items in descending order
   89   68   45   37   12   10    8    6    4    2
```

Review

- In this chapter, we have learned:
  - How to use pointers.
  - How to use pointers to pass arguments to functions using call by reference.
  - To understand the close relationships among pointers, arrays and strings.
  - To understand the use of pointers to functions.
  - How to define and use arrays of strings.



Exercises

Find the error in each of the following program segments. If the error can be corrected, explain how.

a)
```
int *number;
printf( "%d\n", *number );
```
ANS: number has not been assigned to point to a location in memory.

b)
```
float *realPtr;
long *integerPtr;
integerPtr = realPtr;
```
ANS: A pointer cannot be assigned to a different type, other than void *.

c)
```
int * x, y;
x = y;
```
ANS: There are two possible solutions. 1) The indirection operator (*) is not distributive and would be required for y, which would result in a valid pointer assignment. 2) y as it is defined is a valid integer variable, and would require the address operator (&) in the pointer assignment statement.

d)
```
char s[] = "this is a character array";
int count;
for ( ; *s != '\0'; s++)
printf( "%c ", *s );
```
ANS: s should be defined as char *, a constant pointer cannot be moved.

Exercises

e)
```
short *numPtr, result;
void *genericPtr = numPtr;
result = *genericPtr + 7;
```
ANS: 由於無法知道儲存在 genericPtr 地址的變數型態，所以 *genericPtr 的值無法找出。A void * pointer cannot be dereferenced.

f)
```
float x = 19.34;
float xPtr = &x;
printf( "%f\n", xPtr );
```
ANS: xPtr is not defined as a pointer so it should be dereferenced as well.

g)
```
char *s;
printf( "%s\n", s );
```
ANS: s has not been assigned a value, it does not point to anything.