C Pointers

C How to Program, 8/e

7.2 Pointer Variable Definitions and Initialization

- Pointers are variables whose values are memory addresses.
  - Normally, a variable directly contains a specific value.

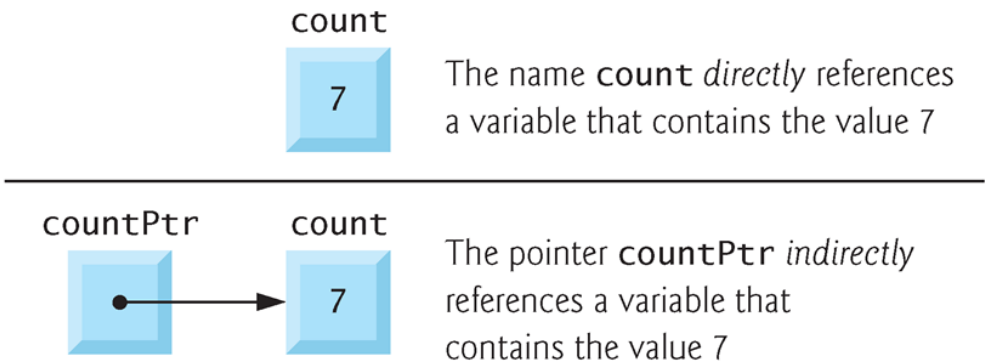


Fig. 7.1 | Directly and indirectly referencing a variable.

7.2 Pointer Variable Definitions and Initialization (cont.)

- `int *countPtr;`
  specifies that variable `countPtr` is of type `int *` (i.e., a pointer to an integer)

- A pointer may be initialized to `NULL`, `0` or an address.
  - Initializing a pointer to `0` is equivalent to initializing a pointer to `NULL`, but `NULL` is preferred.

7.3 Pointer Operators

- The &, or address operator, is a unary operator that returns the address of its operand.
- For example, assuming the definitions
  - ```
    int y = 5;
    int *yPtr;
    ```

  the statement
  - `yPtr = &y;`

  assigns the address of the variable `y` to pointer variable `yPtr`.

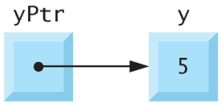


Fig. 7.2 | Graphical representation of a pointer pointing to an integer variable in memory.

Fig. 7.3 | Representation of `y` and `yPtr` in memory.
    ```

7.3 Pointer Operators (Cont.)

- The unary * operator, commonly referred to as the indirection operator, returns the value of the object to which a pointer points.

- For example,
  - `printf( "%d", *yPtr );`

  prints the value of variable `y`, namely 5.

7.3 Pointer Operators (Cont.)

- Figure 7.5 lists the precedence and associativity of the operators introduced to this point.

Operators	Associativity	Type
() [] ++ *(postfix)* -- *(postfix)*	left to right	postfix
+ - ++ -- ! * & (*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 | Precedence and associativity of the operators discussed so far.

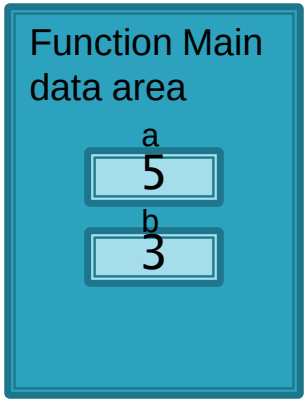
7.4 Passing Arguments to Functions by Reference

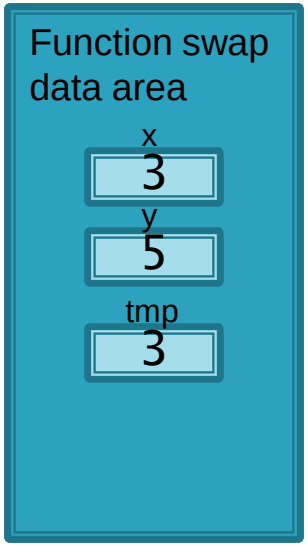
```
1.  int main( )
2.  {
3.        int  a=5, b=3;
4.        swap(a, b);
5.        printf("%d  %d", a, b);
6.  }

7.  int swap(int x, int y)
8.  {
9.        int  tmp;

10.       tmp = y;
11.       y = x;
12.       x = tmp;
13. }
```

- There are two ways to pass arguments to a function—call-by-value and call-by-reference.



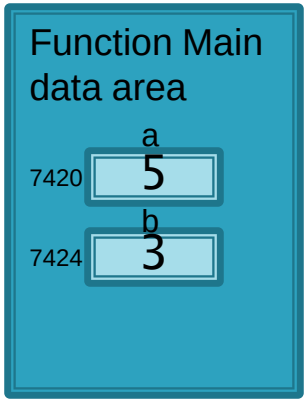


Call by Reference

```
1.  int main( )
2.  {
3.      int  a=5, b=3;
4.      swap(&a, &b);
5.      printf("%d  %d", a, b);
6.  }

7.  void swap(int *x, int *y)
8.  {
9.      int  tmp;

10.     tmp = *y;
11.     *y = *x;
12.     *x = tmp;
13. }
```

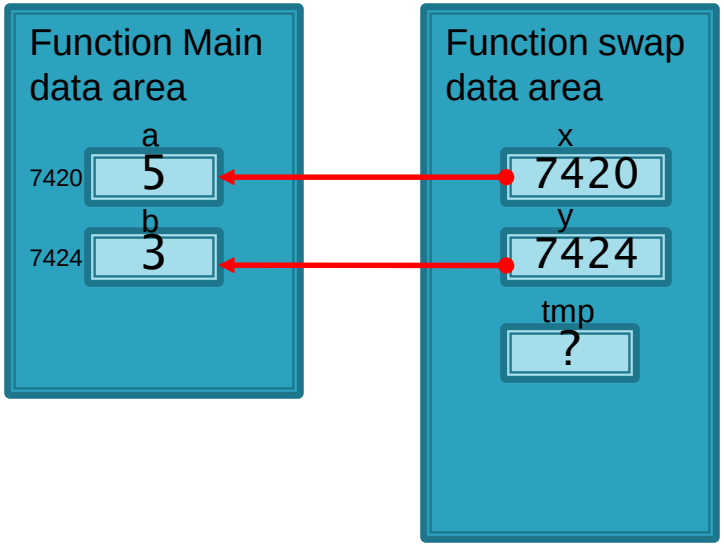


Call by Reference

```
1.   int main( )
2.   {
3.        int  a=5, b=3;
4.        swap(&a, &b);
5.        printf("%d  %d", a, b);
6.   }

7.   void swap(int *x, int *y)
8.   {
9.        int  tmp;

10.       tmp = *y;
11.       *y = *x;
12.       *x = tmp;
13.  }
```

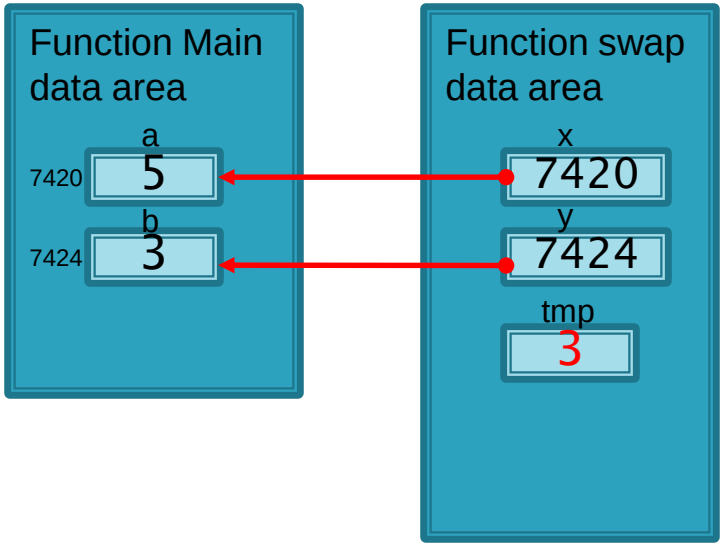


Call by Reference

```
1.  int main( )
2.  {
3.      int  a=5, b=3;
4.      swap(&a, &b);
5.      printf("%d  %d", a, b);
6.  }

7.  void swap(int *x, int *y)
8.  {
9.      int  tmp;

10.     tmp = *y;
11.     *y = *x;
12.     *x = tmp;
13. }
```

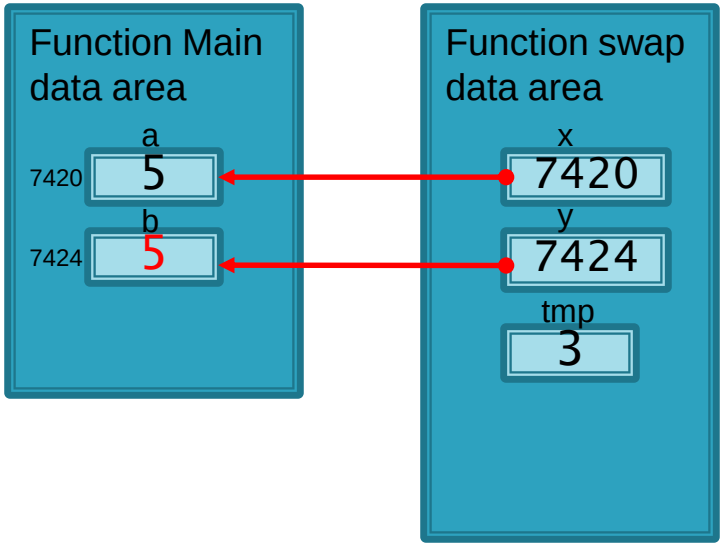


Call by Reference

```
1.   int main( )
2.   {
3.        int  a=5, b=3;
4.        swap(&a, &b);
5.        printf("%d  %d", a, b);
6.   }

7.   void swap(int *x, int *y)
8.   {
9.        int  tmp;

10.       tmp = *y;
11.  ➡    *y = *x;
12.       *x = tmp;
13.  }
```

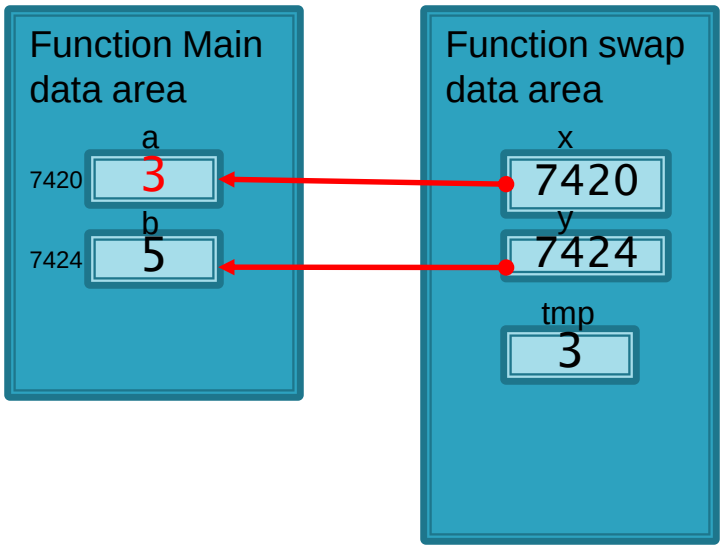


Call by Reference

```
1.   int main( )
2.   {
3.        int  a=5, b=3;
4.        swap(&a, &b);
5.        printf("%d  %d", a, b);
6.   }

7.   void swap(int *x, int *y)
8.   {
9.        int  tmp;

10.       tmp = *y;
11.       *y = *x;
12.       *x = tmp;
13.  }
```

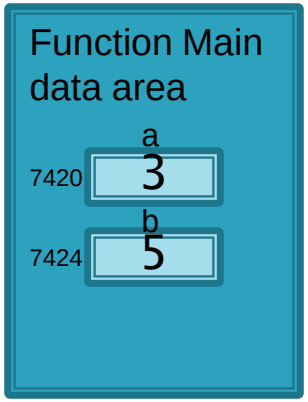


Call by Reference

```
1.  int main( )
2.  {
3.      int  a=5, b=3;
4.      swap(&a, &b);
5.      printf("%d  %d", a, b);
6.  }

7.  void swap(int *x, int *y)
8.  {
9.      int  tmp;

10.     tmp = *y;
11.     *y = *x;
12.     *x = tmp;
13. }
```



7.5 Using the const Qualifier with Pointers

- void func(const int *haha)
- {
- ….
- }

*haha 指到的地方所儲存的資料不可更改

- void func(int * const hehe)
- {
- }

hehe 本身不可更改

```c
// Fig. 7.12: fig07_12.c
// Attempting to modify data through a
// non-constant pointer to constant data.
#include <stdio.h>
void f(const int *xPtr); // prototype

int main(void)
{
   int y; // define y

   f(&y); // f attempts illegal modification
}

// xPtr cannot be used to modify the
// value of the variable to which it points
void f(const int *xPtr)
{
   *xPtr = 100; // error: cannot modify a const object
}
```

ERROR!!

```
error C2166: l-value specifies const object
```

Fig. 7.12 | Attempting to modify data through a non-constant pointer to constant data.

7.5 Using the const Qualifier with Pointers (Const.)

```
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(void)
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 }
```

ptr 本身是一個 constant pointer 不可更改

```
c:\examples\ch07\fig07_13.c(15) : error C2166: l-value specifies const object
```

Fig. 7.13 | Attempting to modify a constant pointer to non-constant data.

- A constant pointer to non-constant data always points to the same memory location, and the data at that location can be modified through the pointer.

```c
// Fig. 7.14: fig07_14.c
// Attempting to modify a constant pointer to constant data.
#include <stdio.h>

int main(void)
{
   int x = 5; // initialize x
   int y; // define y

   // ptr is a constant pointer to a constant integer. ptr always
   // points to the same location; the integer at that location
   // cannot be modified
   const int *const ptr = &x; // initialization is OK

   printf("%d\n", *ptr);
   *ptr = 7; // error: *ptr is const; cannot assign new value
   ptr = &y; // error: ptr is const; cannot assign new address
}
```

ptr所指到的data不可更改
ptr本身亦是const不可更改

```
c:\examples\ch07\fig07_14.c(16) : error C2166: l-value specifies const object
c:\examples\ch07\fig07_14.c(17) : error C2166: l-value specifies const object
```

Fig. 7.14 | Attempting to modify a constant pointer to constant data.

Bubble sort

```
1  // Fig. 7.15: fig07_15.c
2  // Putting values into an array, sorting the values into
3  // ascending order and printing the resulting array.
4  #include <stdio.h>
5  #define SIZE 10
6
7  void bubbleSort(int * const array, const size_t size); // prototype
8
9  int main(void)
10 {
11    // initialize array a
12    int a[SIZE] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
13
14    puts("Data items in original order");
15
16    // loop through array a
17    for (size_t i = 0; i < SIZE; ++i) {
18       printf("%4d", a[i]);
19    }
20
21    bubbleSort(a, SIZE); // sort the array
22
```

int const array[]

將陣列a傳入

Fig. 7.15 | Putting values into an array, sorting the values into ascending order and printing the resulting array. (Part 1 of 4.)

```
23      puts("\nData items in ascending order");
24
25      // loop through array a
26      for (size_t i = 0; i < SIZE; ++i) {
27         printf("%4d", a[i]);
28      }
29
30      puts("");
31   }
32
```

Fig. 7.15 | Putting values into an array, sorting the values into ascending order and printing the resulting array. (Part 2 of 4.)

array本身是const 不可更改
但array所指到的data可以更改

```
33  // sort an array of integers using bubble sort algorithm
34  void bubbleSort(int * const array, const size_t size)
35  {
36     void swap(int *element1Ptr, int *element2Ptr); // prototype
37
38     // loop to control passes
39     for (unsigned int pass = 0; pass < size - 1; ++pass) {
40
41        // loop to control comparisons during each pass
42        for (size_t j = 0; j < size - 1; ++j) {
43
44           // swap adjacent elements if they're out of order
45           if (array[j] > array[j + 1]) {
46              swap(&array[j], &array[j + 1]);
47           }
48        }
49     }
50  }
51
```

swap 原型在此宣告因此swap只能在bubbleSort函式中使用

Fig. 7.15 | Putting values into an array, sorting the values into ascending order and printing the resulting array. (Part 3 of 4.)

```
52  // swap values at memory locations to which element1Ptr and
53  // element2Ptr point
54  void swap(int *element1Ptr, int *element2Ptr)
55  {
56     int hold = *element1Ptr;
57     *element1Ptr = *element2Ptr;
58     *element2Ptr = hold;
59  }
```

```
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
```

Fig. 7.15 | Putting values into an array, sorting the values into ascending order and printing the resulting array. (Part 4 of 4.)

7.7 `sizeof` Operator

- C provides the special unary operator `sizeof` to determine the **size in bytes** of an array (or any other data type) during program compilation.

```c
// Fig. 7.16: fig07_16.c
// Applying sizeof to an array name returns
// the number of bytes in the array.
#include <stdio.h>
#define SIZE 20

size_t getSize(float *ptr); // prototype

int main(void)
{
   float array[SIZE]; // create array

   printf("The number of bytes in the array is %u"
          "\nThe number of bytes returned by getSize is %u\n",
          sizeof(array), getSize(array));
}

// return size of ptr
size_t getSize(float *ptr)
{
   return sizeof(ptr);
}
```

整個array大小

ptr本身的大小

Fig. 7.16 | Applying `sizeof` to an array name returns the number of bytes in the array. (Part 1 of 2.)

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

Fig. 7.16 | Applying `sizeof` to an array name returns the number of bytes in the array. (Part 2 of 2.)

```c
// Fig. 7.17: fig07_17.c
// Using operator sizeof to determine standard data type sizes.
#include <stdio.h>

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

   printf("    sizeof c = %u\tsizeof(char)  = %u"
          "\n     sizeof s = %u\tsizeof(short) = %u"
          "\n     sizeof i = %u\tsizeof(int) = %u"
          "\n     sizeof l = %u\tsizeof(long) = %u"
          "\n    sizeof ll = %u\tsizeof(long long) = %u"
          "\n     sizeof f = %u\tsizeof(float) = %u"
```

不同平台可能有不同結果

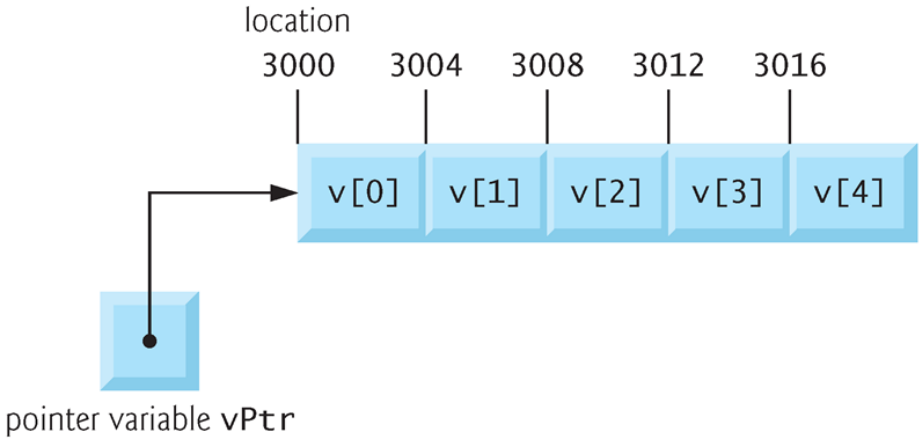
Fig. 7.17 | Using operator `sizeof` to determine standard data type sizes. (Part 1 of 2.)

```
24              "\n     sizeof d = %u\tsizeof(double) = %u"
25              "\n    sizeof ld = %u\tsizeof(long double) = %u"
26              "\n sizeof array = %u"
27              "\n   sizeof ptr = %u\n",
28              sizeof c, sizeof(char), sizeof s, sizeof(short), sizeof i,
29              sizeof(int), sizeof l, sizeof(long), sizeof ll,
30              sizeof(long long), sizeof f, sizeof(float), sizeof d,
31              sizeof(double), sizeof ld, sizeof(long double),
32              sizeof array, sizeof ptr);
33   }
```

```
     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 ll = 8       sizeof(long long) = 8
     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
```

Fig. 7.17 | Using operator `sizeof` to determine standard data type sizes. (Part 2 of 2.)

7.8 Pointer Expressions and Pointer Arithmetic



1. int v[5];
2. int *vPtr;

3. vPtr = &v[0];

Fig. 7.18 | Array v and a pointer variable vPtr that points to v.

- `vPtr += 2;`
  would produce 3008 (3000 + 2 * 4) assuming an integer is stored in 4 bytes of memory.

7.8 Pointer Expressions and Pointer Arithmetic (Cont.)

- In the array `v`, `vPtr` would now point to `v[2]`

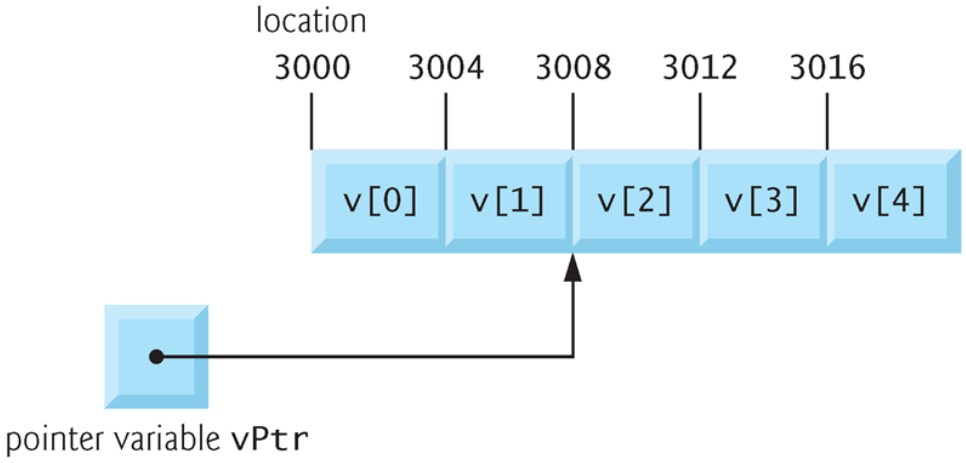


Fig. 7.19 | The pointer `vPtr` after pointer arithmetic.

7.8 Pointer Expressions and Pointer Arithmetic (Cont.)

- If `vPtr` contains the location `3000`, and `v2Ptr` contains the address `3008`, the statement
  - `x = v2Ptr - vPtr;`

  would assign to **`x` the number of array elements** from `vPtr` to `v2Ptr`, in this case `2` (not 8).
- Pointer arithmetic is **meaningless unless performed on an array**.

- Pointer comparison
  - A common use of pointer comparison is determining whether a pointer is `NULL`.

7.9 Relationship between Pointers and Arrays

- An array name can be thought of as a constant pointer.

- int b[5];
- int *bPtr;
- bPtr = b;
  is equivalent to
  bPtr = &b[0];

- b[3];
  is equivalent to
  *(bPtr + 3);

7.10 Arrays of Pointers

- Arrays may contain pointers.
- A common use of an array of pointers is to form an array of strings

- Consider the definition of string array `suit`, which might be useful in representing a deck of cards.
  - `const char *suit[ 4 ] = { "Hearts", "Diamonds", "Clubs", "Spades" };`

7.10 Arrays of Pointers (Cont.)

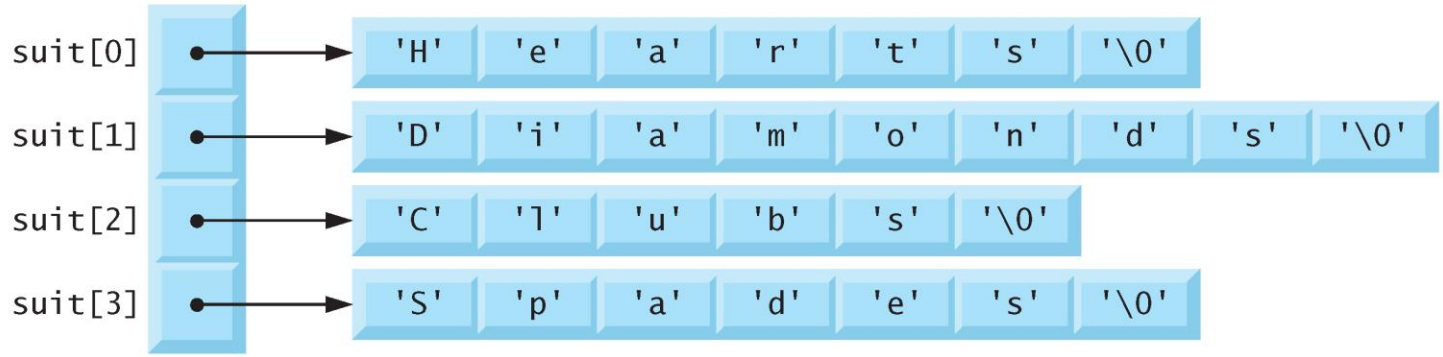


Fig. 7.22 | Graphical representation of the `suit` array.

- Each pointer points to the first character of its corresponding string.
- The array `suit` provides access to character strings of any length.

7.11 Case Study: Card Shuffling and Dealing Simulation

- We use 4-by-13 double-subscripted array `deck` to represent the deck of playing cards (Fig. 7.23).

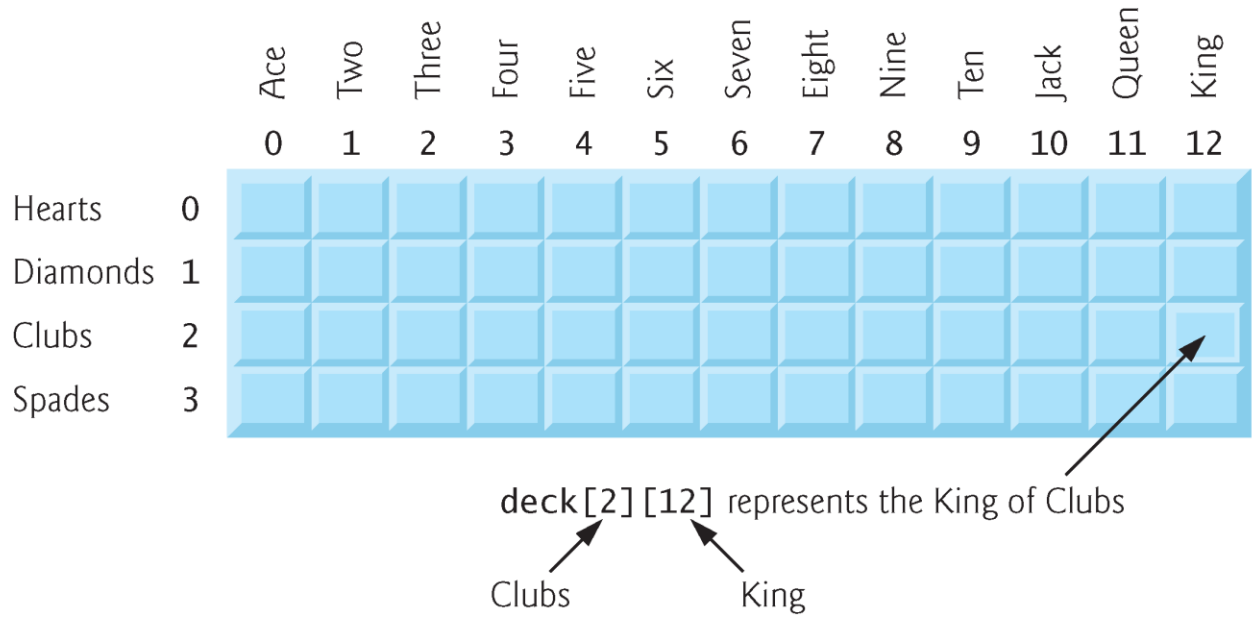


Fig. 7.23 | Two-dimensional array representation of a deck of cards.

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

#define SUITS 4
#define FACES 13
#define CARDS 52

// prototypes
void shuffle(unsigned int wDeck[][FACES]); // shuffling modifies wDeck
void deal(unsigned int wDeck[][FACES], const char *wFace[],
   const char *wSuit[]); // dealing doesn't modify the arrays

int main(void)
{
   // initialize deck array
   unsigned int deck[SUITS][FACES] = {0};

   srand(time(NULL)); // seed random-number generator
   shuffle(deck); // shuffle the deck

```

Fig. 7.24 | Card shuffling and dealing. (Part 1 of 4.)

```
24      // initialize suit array
25      const char *suit[SUITS] =
26         {"Hearts", "Diamonds", "Clubs", "Spades"};
27
28      // initialize face array
29      const char *face[FACES] =
30         {"Ace", "Deuce", "Three", "Four",
31          "Five", "Six", "Seven", "Eight",
32          "Nine", "Ten", "Jack", "Queen", "King"};
33
34      deal(deck, face, suit); // deal the deck
35   }
36
```

Fig. 7.24 | Card shuffling and dealing. (Part 2 of 4.)

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

Fig. 7.24 | Card shuffling and dealing. (Part 3 of 4.)

```
56  // deal cards in deck
57  void deal(unsigned int wDeck[][FACES], const char *wFace[],
58     const char *wSuit[])
59  {
60     // deal each of the cards
61     for (size_t card = 1; card <= CARDS; ++card) {
62        // loop through rows of wDeck
63        for (size_t row = 0; row < SUITS; ++row) {
64           // loop through columns of wDeck for current row
65           for (size_t column = 0; column < FACES; ++column) {
66              // if slot contains current card, display card
67              if (wDeck[row][column] == card) {
68                 printf("%5s of %-8s%c", wFace[column], wSuit[row],
69                    card % 2 == 0 ? '\n' : '\t'); // 2-column format
70              }
71           }
72        }
73     }
74  }
```

Fig. 7.24 | Card shuffling and dealing. (Part 4 of 4.)

```
 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
```

Fig. 7.25 | Sample run of card dealing program.

7.12 Pointers to Functions

- A pointer to a function contains the address of the function in memory.

```
1.  void  function1(int);
2.  void  function2(int);
3.  int main()
4.  {
5.      void (*func)(int);
6.      int choice;
7.
8.      printf("Enter a choice:");
9.      scanf("%d", &choice);
10.     if( choice == 1 )
11.         func = function1;
12.     else
13.         func = function2;
14.     func(choice);
15. }
16. void  function1(int c)
17. {
18.         printf("Your choice is %d\n", c);
19.         printf("function1 is called");
20. }
21. void  function2(int c)
22. {
23.         printf("Your choice is %d\n", c);
24.         printf("function2 is called");
25. }
```

宣告 function pointer變數

根據choice 呼叫不同的 function

```c
// Fig. 7.28: fig07_28.c
// Demonstrating an array of pointers to functions.
#include <stdio.h>

// prototypes
void function1(int a);
void function2(int b);
void function3(int c);

int main(void)
{
   // initialize array of 3 pointers to functions that each take an
   // int argument and return void
   void (*f[3])(int) = { function1, function2, function3 };

   printf("%s", "Enter a number between 0 and 2, 3 to end: ");
   size_t choice; // variable to hold user's choice
   scanf("%u", &choice);

```

Fig. 7.28 | Demonstrating an array of pointers to functions. (Part 1 of 3.)

```
20      // process user's choice
21      while (choice >= 0 && choice < 3) {
22
23         // invoke function at location choice in array f and pass
24         // choice as an argument
25         (*f[choice])(choice);
26
27         printf("%s", "Enter a number between 0 and 2, 3 to end: ");
28         scanf("%u", &choice);
29      }
30
31      puts("Program execution completed.");
32   }
33
34   void function1(int a)
35   {
36      printf("You entered %d so function1 was called\n\n", a);
37   }
38
39   void function2(int b)
40   {
41      printf("You entered %d so function2 was called\n\n", b);
42   }
43
```

Fig. 7.28 | Demonstrating an array of pointers to functions. (Part 2 of 3.)

```c
44  void function3(int c)
45  {
46     printf("You entered %d so function3 was called\n\n", c);
47  }
```

```
Enter a number between 0 and 2, 3 to end: 0
You entered 0 so function1 was called

Enter a number between 0 and 2, 3 to end: 1
You entered 1 so function2 was called

Enter a number between 0 and 2, 3 to end: 2
You entered 2 so function3 was called

Enter a number between 0 and 2, 3 to end: 3
Program execution completed.
```

Fig. 7.28 | Demonstrating an array of pointers to functions. (Part 3 of 3.)