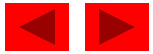
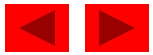
Chapter 7 - Pointers

Outline



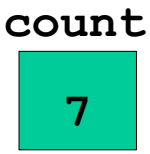
7.1 Introduction

- Pointers
  - Powerful, but difficult to master
  - Simulate call-by-reference
  - Close relationship with arrays and strings

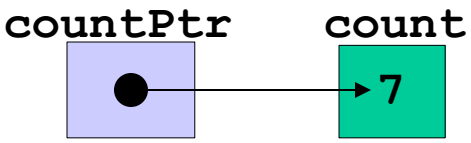


7.2 Pointer Variable Declarations and Initialization

- Pointer variables
  - Contain memory addresses as their values
  - Normal variables contain a specific value (direct reference)



  - Pointers contain address of a variable that has a specific value (indirect reference)
  - Indirection – referencing a pointer value



7.2 Pointer Variable Declarations and Initialization

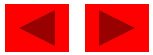
- Pointer declarations
  - `*` used with pointer variables

```
int *myPtr;
```

  - Declares a pointer to an `int` (pointer of type `int *`)
  - Multiple pointers require using a `*` before each variable declaration

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

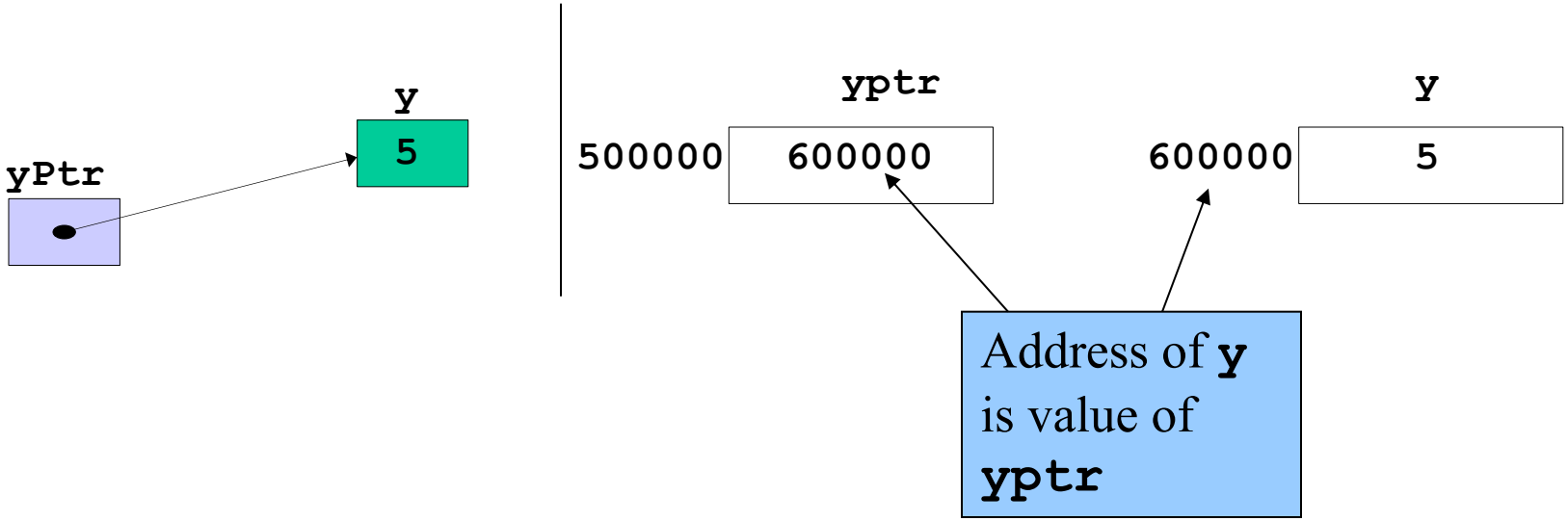
  - Can declare pointers to any data type
  - Initialize pointers to `0`, `NULL`, or an address
    - `0` or `NULL` – points to nothing (`NULL` preferred)



7.3 Pointer Operators

- **&** (address operator)
  - Returns address of operand

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

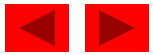


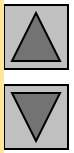
7.3 Pointer Operators

- **`*`** (indirection/dereferencing operator)
  - Returns a synonym/alias of what its operand points to
  - **`*yptr`** returns **`y`** (because **`yptr`** points to **`y`**)
  - **`*`** can be used for assignment
    - Returns alias to an object

```
*yptr = 7;  // changes y to 7
```

  - Dereferenced pointer (operand of **`*`**) must be an lvalue (no constants)
- **`*`** and **`&`** are inverses
  - They cancel each other out





Outline

1. Declare variables

itialize variables

```
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 inverses of "
20            "each other.\n&*aPtr = %p"
21            "\n*&aPtr = %p\n", &*aPtr, *&aPtr );
22
23    return 0;
24 }
```

The address of **a** is the value of **aPtr**.

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

Notice how ***** and **&** are inverses

Program Output

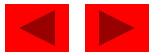
```
The address of a is 0012FF88
The value of aPtr is 0012FF88

The value of a is 7
The value of *aPtr is 7
Proving that * and & are complements of each other.
&*aPtr = 0012FF88
*&aPtr = 0012FF88
```

7.4 Calling Functions by Reference

- Call by reference with pointer arguments
  - Pass address of argument using **&** operator
  - Allows you to change actual location in memory
  - Arrays are not passed with **&** because the array name is already a pointer
- ***** operator
  - Used as alias/nickname for variable inside of function

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

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



```
1  /* Fig. 7.7: fig07_07.c
2     Cube a variable using call
3     with a pointer argument */
4
5  #include <stdio.h>
6
7  void cubeByReference( int * );    /* p
8
9  int main()
10 {
11    int number = 5;
12
13    printf( "The original value of number is %d", number );
14    cubeByReference( &number );
15    printf( "\nThe new value of number is %d\n", number );
16
17    return 0;
18 }
19
20 void cubeByReference( int *nPtr )
21 {
22    *nPtr = *nPtr * *nPtr * *nPtr;  /* cube number in main */
23 }
```

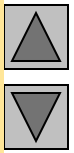
Notice that the function prototype takes a pointer to an integer (**int ***).

Notice how the address of **number** is given - **cubeByReference** expects a pointer (an address of a variable).

Inside **cubeByReference**, ***nPtr** is used (***nPtr** is **number**).

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

Program Output



Outline

1. Function prototype

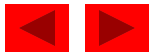
1.1 Initialize variables

2. Call function

3. Define function

7.5 Using the `const` Qualifier with Pointers

- **`const`** qualifier
  - Variable cannot be changed
  - Use **`const`** if function does not need to change a variable
  - Attempting to change a **`const`** variable produces an error
- **`const`** pointers
  - Point to a constant memory location
  - Must be initialized when declared
  - **`int *const myPtr = &x;`**
    - Type **`int *const`** – constant pointer to an **`int`**
  - **`const int *myPtr = &x;`**
    - Regular pointer to a **`const int`**
  - **`const int *const Ptr = &x;`**
    - **`const`** pointer to a **`const int`**
    - **`x`** can be changed, but not **`*Ptr`**



```c
/* Fig. 7.13: fig07_13.c
   Attempting to modify a constant pointer to
   non-constant data */

#include <stdio.h>

int main()
{
   int x, y;

   int * const ptr = &x; /* ptr is a constant pointer to an
                            integer. An integer can be modified
                            through ptr, but ptr always points
                            to the same memory location. */
   *ptr = 7;
   ptr = &y;

   return 0;
}
```

Changing ***ptr** is allowed – **x** is not a constant.

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

Program Output

```
FIG07_13.c:
Error E2024 FIG07_13.c 16: Cannot modify a const object in
function main
*** 1 errors in Compile ***
```

Outline

1. Declare variables

1.1 Declare `const` pointer to an `int`

2. Change `*ptr` (which is `x`)

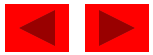
2.1 Attempt to change `ptr`

3. Output

7.6 Bubble Sort Using Call-by-reference

- Implement bubblesort 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
- Psuedocode

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

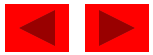


7.6 Bubble Sort Using Call-by-reference

- **`sizeof`**
  - Returns size of operand in bytes
  - For arrays: size of 1 element * number of elements
  - if **`sizeof( int )`** equals **`4`** bytes, then

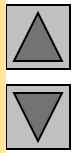
```
int myArray[ 10 ];
printf( "%d", sizeof( myArray ) );
```

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



```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 *, const int );

int main()
{

   int a[ SIZE ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
   int i;

   printf( "Data items in original
   for ( i = 0; i < SIZE; i++ )
      printf( "%4d", a[ i ] );

   bubbleSort( a, SIZE );          /* sort the array */
   printf( "\nData items in ascending order\n" );

   for ( i = 0; i < SIZE; i++ )
      printf( "%4d", a[ i ] );

   printf( "\n" );

   return 0;
}

void bubbleSort( int *array, const int size )
{
   void swap( int *, int * );
```

Bubblesort gets passed the address of array elements (pointers). The name of an array is a pointer.



Outline

1. Initialize array

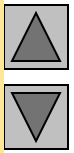
1.1 Declare variables

2. Print array

2.1 Call `bubbleSort`

2.2 Print array

```
33    int pass, j;
34    for ( pass = 0; pass < size - 1; pass++ )
35
36       for ( j = 0; j < size - 1; j++ )
37
38          if ( array[ j ] > array[ j + 1 ] )
39             swap( &array[ j ], &array[ j + 1 ] );
40 }
41
42 void swap( int *element1Ptr, int *element2Ptr )
43 {
44    int hold = *element1Ptr;
45    *element1Ptr = *element2Ptr;
46    *element2Ptr = hold;
47 }
```



Outline

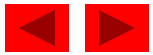
3. Function definitions

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
```

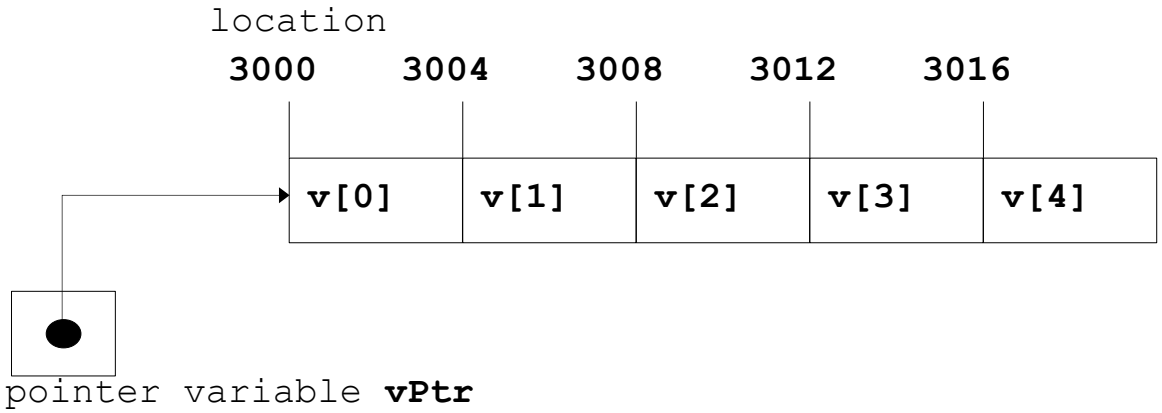
7.7 Pointer Expressions and Pointer Arithmetic

- Arithmetic operations can be performed on pointers
  - Increment/decrement pointer (**++** or **--**)
  - Add an integer to a pointer(**+** or **+=** , **-** or **-=**)
  - Pointers may be subtracted from each other
  - Operations meaningless unless performed on an array



7.7 Pointer Expressions and Pointer Arithmetic

- 5 element **int** array on machine with 4 byte **int**s
  - **vPtr** points to first element **v[0]**
    - at location **3000** (**vPtr = 3000**)
  - **vPtr += 2;** sets **vPtr** to **3008**
    - **vPtr** points to **v[2]** (incremented by 2), but the machine has 4 byte **int**s, so it points to address **3008**

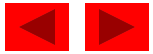


7.7 Pointer Expressions and Pointer Arithmetic

- Subtracting pointers
  - Returns number of elements from one to the other. If

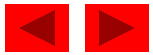
```
vPtr2 = &v[ 2 ];
vPtr = &v[ 0 ];
```

  - `vPtr2 - vPtr` would produce 2
- Pointer comparison (`<`, `==` , `>`)
  - See which pointer points to the higher numbered array element
  - Also, see if a pointer points to `0`



7.7 Pointer Expressions and Pointer Arithmetic

- Pointers of the same type can be assigned to each other
  - If not the same type, a cast operator must be used
  - Exception: pointer to **`void`** (type **`void *`**)
    - Generic pointer, represents any type
    - No casting needed to convert a pointer to **`void`** pointer
    - **`void`** pointers cannot be dereferenced



7.8 The Relationship Between Pointers and Arrays

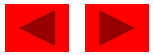
- Arrays and pointers closely related
  - Array name like a constant pointer
  - Pointers can do array subscripting operations
- Declare an array `b[ 5 ]` and a pointer `bPtr`
  - To set them equal to one another use:

```
bPtr = b;
```

    - The array name (`b`) is actually the address of first element of the array `b[ 5 ]`

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

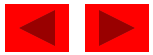
    - Explicitly assigns `bPtr` to address of first element of `b`



7.8 The Relationship Between Pointers and Arrays

- Element **b[3]**
  - Can be accessed by ***(bPtr + 3)**
    - Where **n** is the offset. Called pointer/offset notation
  - Can be accessed by **bptr[3]**
    - Called pointer/subscript notation
    - **bPtr[3]** same as **b[3]**
  - Can be accessed by performing pointer arithmetic on the array itself

    ***(b + 3)**

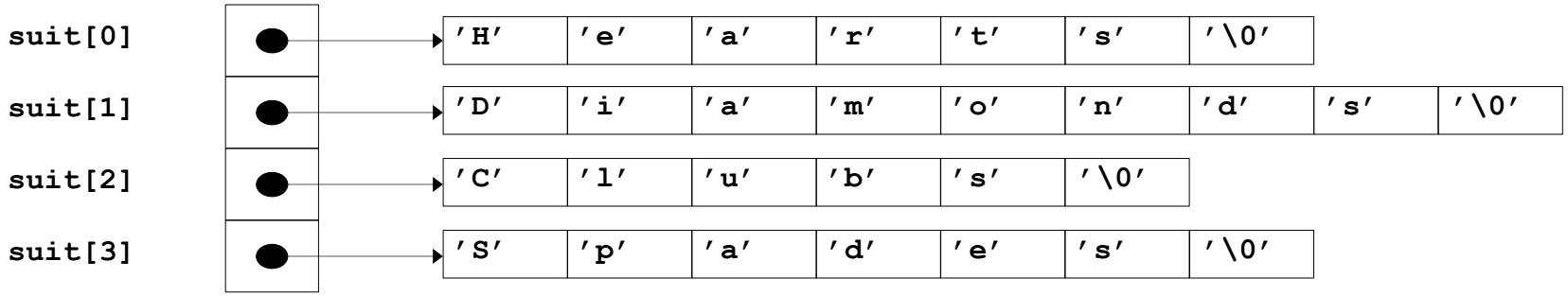


7.9 Arrays of Pointers

- Arrays can contain pointers
- For example: an array of strings

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

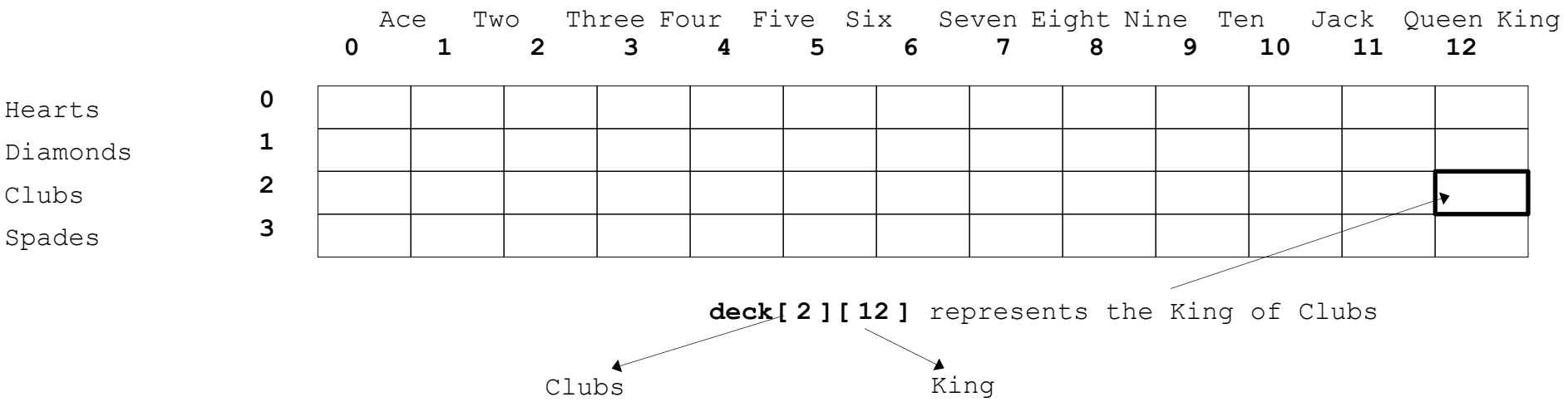
  - Strings are pointers to the first character
  - **`char *`** – each element of **`suit`** is a pointer to a **`char`**
  - The strings are not actually stored in the array **`suit`**, only pointers to the strings are stored



  - **`suit`** array has a fixed size, but strings can be of any size

7.10 Case Study: A Card Shuffling and Dealing Simulation

- Card shuffling program
  - Use array of pointers to strings
  - Use double scripted array (suit, face)



  - The numbers 1-52 go into the array
    - Representing the order in which the cards are dealt

7.10 Case Study: A Card Shuffling and Dealing Simulation

- Pseudocode
  - Top level:

    Shuffle and deal 52 cards
  - First refinement:

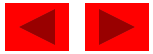
    Initialize the suit array

    Initialize the face array

    Initialize the deck array

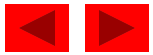
    Shuffle the deck

    Deal 52 cards



7.10 Case Study: A Card Shuffling and Dealing Simulation

- Second refinement
  - Convert *shuffle the deck* to
    For each of the 52 cards
    Place card number in randomly selected unoccupied slot of deck
  - Convert *deal 52 cards* to
    For each of the 52 cards
    Find card number in deck array and print face and suit of card



7.10 Case Study: A Card Shuffling and Dealing Simulation

- Third refinement
  - Convert *shuffle the deck* to

    Choose slot of deck randomly

    While chosen slot of deck has been previously chosen

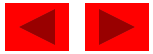
    Choose slot of deck randomly

    Place card number in chosen slot of deck
  - Convert *deal 52 cards* to

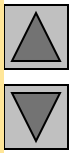
    For each slot of the deck array

    If slot contains card number

    Print the face and suit of the card



```
1  /* Fig. 7.24: fig07 24.c
2     Card shuffling dealing program */
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <time.h>
6
7  void shuffle( int [][ 13 ] );
8  void deal( const int [][ 13 ], const char *[], const char *[] );
9
10 int main()
11 {
12    const char *suit[ 4 ] =
13       { "Hearts", "Diamonds", "Clubs", "Spades" };
14    const char *face[ 13 ] =
15       { "Ace", "Deuce", "Three", "Four",
16         "Five", "Six", "Seven", "Eight",
17         "Nine", "Ten", "Jack", "Queen", "King" };
18    int deck[ 4 ][ 13 ] = { 0 };
19
20    srand( time( 0 ) );
21
22    shuffle( deck );
23    deal( deck, face, suit );
24
25    return 0;
26 }
27
28 void shuffle( int wDeck[][ 13 ] )
29 {
30    int row, column, card;
31
32    for ( card = 1; card <= 52; card++ ) {
```



Outline

1. Initialize `suit` and `face` arrays

1.1 Initialize `deck` array

2. Call function `shuffle`

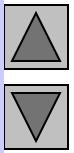
2.1 Call function `deal`

3. Define functions

```
33       do {
34          row = rand() % 4;
35          column = rand() % 13;
36       } while( wDeck[ row ][ column ] != 0 );
37
38       wDeck[ row ][ column ] = card;
39    }
40 }
41
42 void deal( const int wDeck[][ 13 ], const char *wFace[],
43            const char *wSuit[] )
44 {
45    int card, row, column;
46
47    for ( card = 1; card <= 52; card++ )
48
49       for ( row = 0; row <= 3; row++ )
50
51          for ( column = 0; column <= 12; column++ )
52
53             if ( wDeck[ row ][ column ] == card )
54                printf( "%5s of %-8s%c",
55                wFace[ column ], wSuit[ row ],
56                card % 2 == 0 ? '\n' : '\t' );
57 }
```

The numbers 1-52 are randomly placed into the **deck** array.

Searches **deck** for the **card** number, then prints the **face** and **suit**.



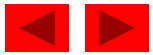
Outline

Program Output

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

7.11 Pointers to Functions

- Pointer to function
  - Contains address of function
  - Similar to how array name is address of first element
  - Function name is starting address of code that defines function
- Function pointers can be
  - Passed to functions
  - Stored in arrays
  - Assigned to other function pointers



7.11 Pointers to Functions

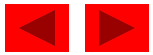
- Example: bubblesort
  - Function **bubble** takes a function pointer
    - **bubble** calls this helper function
    - this determines ascending or descending sorting
  - The argument in **bubblesort** for the function pointer:

    ```
    int ( *compare )( int, int )
    ```

    tells bubblesort to expect a pointer to a function that takes two **ints** and returns a **int**
  - If the parentheses were left out:

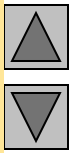
    ```
    int *compare( int, int )
    ```

    - Declares a function that receives two integers and returns a pointer to a **int**



```
1  /* Fig. 7.26: fig07_26.c
2     Multipurpose sorting program using function pointers */
3  #include <stdio.h>
4  #define SIZE 10
5  void bubble( int [], const int, int (*)( int, int ) );
6  int ascending( int, int );
7  int descending( int, int );
8
9  int main()
10 {
11
12    int order,
13        counter,
14        a[ SIZE ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
15
16    printf( "Enter 1 to sort in ascending order,\n"
17            "Enter 2 to sort in descending order: " );
18    scanf( "%d",  &order );
19    printf( "\nData items in original order\n" );
20
21    for ( counter = 0; counter < SIZE; counter++ )
22       printf( "%5d", a[ counter ] );
23
24    if ( order == 1 ) {
25       bubble( a, SIZE, ascending );
26       printf( "\nData items in ascending order\n" );
27    }
28    else {
29       bubble( a, SIZE, descending );
30       printf( "\nData items in descending order\n" );
31    }
32
```

Notice the function pointer parameter.



Outline

1. Initialize array

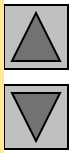
t for ascending
nding sorting

2.1 Put appropriate function pointer into bubblesort

2.2 Call `bubble`

3. Print results

```
33    for ( counter = 0; counter < SIZE; counter++ )
34       printf( "%5d", a[ counter ] );
35
36    printf( "\n" );
37
38    return 0;
39 }
40
41 void bubble( int work[], const int size,
42              int (*compare)( int, int ) )
43 {
44    int pass, count;
45
46    void swap( int *, int * );
47
48    for ( pass = 1; pass < size; pass++ )
49
50       for ( count = 0; count < size - 1; count++ )
51
52          if ( (*compare)( work[ count ], work[ count + 1 ] ) )
53             swap( &work[ count ], &work[ count + 1 ] );
54 }
55
56 void swap( int *element1Ptr, int *element2Ptr )
57 {
58    int temp;
59
60    temp = *element1Ptr;
61    *element1Ptr = *element2Ptr;
62    *element2Ptr = temp;
63 }
64
```



Outline

3.1 Define functions

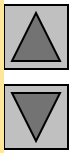
ascending and **descending** return **true** or **false**. **bubble** calls **swap** if the function call returns **true**.

Notice how function pointers are called using the dereferencing operator. The * is not required, but emphasizes that **compare** is a function pointer and not a function.

```
65 int ascending( int a, int b )
66 {
67    return b < a;   /* swap if b is less than a */
68 }
69
70 int descending( int a, int b )
71 {
72    return b > a;   /* swap if b is greater than a */
73 }
```

```
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
```



Outline

3.1 Define functions

Program Output