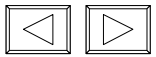
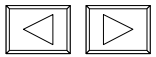
Capitolo 5 - Pointers and Strings

Outline



5.1 Introduction

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



5.2 Pointer Variable Declarations and Initialization

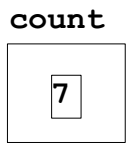
- Pointer variables
  - Contain memory addresses as their values
  - Normal variables contain a specific value (direct reference)
  - Pointers contain the address of a variable that has a specific value (indirect reference)
- Indirection
  - Referencing a pointer value
- Pointer declarations
  - **`*`** indicates variable is a pointer

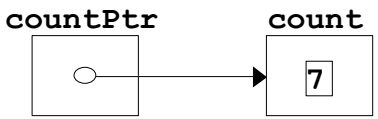
```
int *myPtr;
```

declares a pointer to an **`int`**, a pointer of type **`int *`**

  - Multiple pointers require multiple asterisks

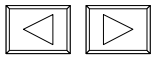
```
int *myPtr1, *myPtr2;
```





5.2 Pointer Variable Declarations and Initialization

- Can declare pointers to any data type
- Pointers initialization
  - Initialized to `0`, **`NULL`**, or an address
    - `0` or **`NULL`** points to nothing

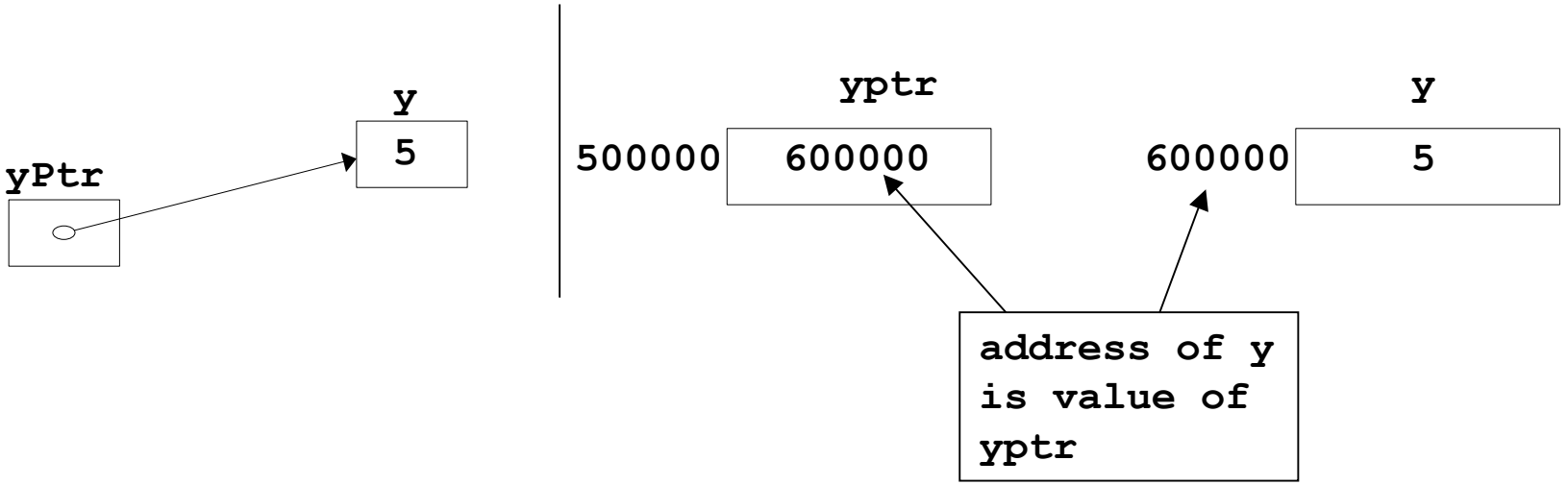


5.3 Pointer Operators

- **&** (address operator)
  - Returns the address of its operand
  - Example

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

  - **yPtr** "points to" **y**



5.3 Pointer Operators

- **`*`** (indirection/dereferencing operator)
  - Returns the value of what its operand points to
  - **`*yPtr`** returns **`y`** (because **`yPtr`** points to **`y`**).
  - **`*`** can be used to assign a value to a location in memory

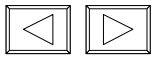
```
*yptr = 7;        // changes y to 7
```

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

```
*&myVar == myVar
```

and

```
&*yPtr == yPtr
```

```
1  // Fig. 5.4: fig05_04.cpp
2  // Using the & and * operators
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  int main()
9  {
10    int a;        // a is an integer
11    int *aPtr;    // aPtr is a pointer to an integer
12
13    a = 7;
14    aPtr = &a;    // aPtr set to address of a
15
16    cout << "The address of a is " << &a
17         << "\nThe value of aPtr is " << aPtr;
18
19    cout << "\n\nThe value of a is " << a
20         << "\nThe value of *aPtr is " << *aPtr;
21
22    cout << "\n\nShowing that * and & are inverses of "
23         << "each other.\n&*aPtr = " << &*aPtr
24         << "\n*&aPtr = " << *&aPtr << endl;
25    return 0;
26 }
```

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

Outline

1. Declare variables

2 Initialize variables

3 Print

Program Output

```
The address of a is 006AFDF4
The value of aPtr is 006AFDF4
The value of a is 7
The value of *aPtr is 7
Showing that * and & are inverses of each other.
&*aPtr = 006AFDF4
*&aPtr = 006AFDF4
```

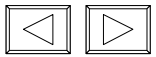
5.4 Calling Functions by Reference

- 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 doubleNum( int *number )
{
    *number = 2 * ( *number );
}
```

  - ***number** used as nickname for the variable passed in
  - When the function is called, must be passed an address

```
doubleNum( &myNum );
```

```
1  // Fig. 5.7: fig05_07.cpp
2  // Cube a variable using call-by-reference
3  // with a pointer argument
4  #include <iostream>
5
6  using std::cout;
7  using std::endl;
8
9  void cubeByReference( int * );    // p
10
11 int main()
12 {
13    int number = 5;
14
15    cout << "The original value of number is " << number;
16    cubeByReference( &number );
17    cout << "\nThe new value of number is " << number << endl;
18    return 0;
19 }
20
21 void cubeByReference( int *nPtr )
22 {
23    *nPtr = *nPtr * *nPtr * *nPtr;  // cube number in main
24 }
```

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**).

Outline

1. Function prototype - takes a pointer to an `int`.

1.1 Initialize variables

2. Call function

3. Define function

Program Output

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

5.5 Using the Const Qualifier with Pointers

- **`const`** qualifier
  - Variable cannot be changed
  - **`const`** used when function does not need to change a variable
  - Attempting to change a **`const`** variable is a compiler error
- **`const`** pointers
  - Point to same memory location
  - Must be initialized when declared

```
int *const myPtr = &x;
```

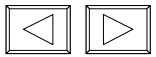
    - Constant pointer to a non-constant **`int`**

```
const int *myPtr = &x;
```

    - Non-constant pointer to a constant **`int`**

```
const int *const Ptr = &x;
```

    - Constant pointer to a constant **`int`**



```
1  // Fig. 5.13: fig05_13.cpp
2  // Attempting to modify a constant pointer to
3  // non-constant data
4  #include <iostream>
5
6  int main()
7  {
8     int x, y;
9
10    int * const ptr = &x; // ptr is a constant pointer to an
11                          // integer. An integer can be modified
12                          // through ptr, but ptr always points
13                          // to the same memory location.
14    *ptr = 7;
15    ptr = &y;
16
17    return 0;
18 }
```

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

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

```
Error E2024 Fig05_13.cpp 15: Cannot modify a const object in function
main()
```

Outline

1. Declare variables

1.1 Declare `const` pointer to an `int`.

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

2.1 Attempt to change `ptr`.

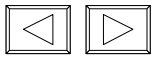
3. Output

Program Output

5.6 Bubble Sort Using Call-by-reference

- Implement bubblesort using pointers
  - **`swap`** function must receive the address (using **`&`**) of the array elements
    - array elements have call-by-value default
  - Using pointers and the **`*`** operator, **`swap`** is able to switch the values of the actual array elements
- Psuedocode

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



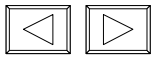
5.6 Bubble Sort Using Call-by-reference

- **`sizeof`**
  - Returns size of operand in bytes
  - For arrays, sizeof returns

    (the size of 1 element) * (number of elements)
  - if **`sizeof( int ) = 4`**, then

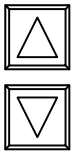
```
int myArray[10];
cout << sizeof(myArray);
```

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



```
1  // Fig. 5.15: fig05 15.cpp
2  // This program puts values into an array, sorts the values into
3  // ascending order, and prints the resulting array.
4  #include <iostream>
5
6  using std::cout;
7  using std::endl;
8
9  #include <iomanip>
10
11 using std::setw;
12
13 void bubbleSort( int *, const int );
14
15 int main()
16 {
17    const int arraySize = 10;
18    int a[ arraySize ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
19    int i;
20
21    cout << "Data items in original order\n";
22
23    for ( i = 0; i < arraySize; i++ )
24       cout << setw( 4 ) << a[ i ];
25
26    bubbleSort( a, arraySize );          // sort the array
27    cout << "\nData items in ascending order\n";
28
29    for ( i = 0; i < arraySize; i++ )
30       cout << setw( 4 ) << a[ i ];
31
32    cout << endl;
33    return 0;
34 }
```

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



Outline

1. Initialize array

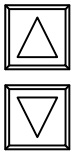
1.1 Declare variables

2. Print array

2.1 Call `bubbleSort`

2.2 Print array

```
36 void bubbleSort( int *array, const int size )
37 {
38    void swap( int * const, int * const );
39
40    for ( int pass = 0; pass < size - 1; pass++ )
41
42       for ( int j = 0; j < size - 1; j++ )
43
44          if ( array[ j ] > array[ j + 1 ] )
45             swap( &array[ j ], &array[ j + 1 ] );
46 }
47
48 void swap( int * const element1Ptr, int * const element2Ptr )
49 {
50    int hold = *element1Ptr;
51    *element1Ptr = *element2Ptr;
52    *element2Ptr = hold;
53 }
```



Outline

3. Define `bubbleSort`

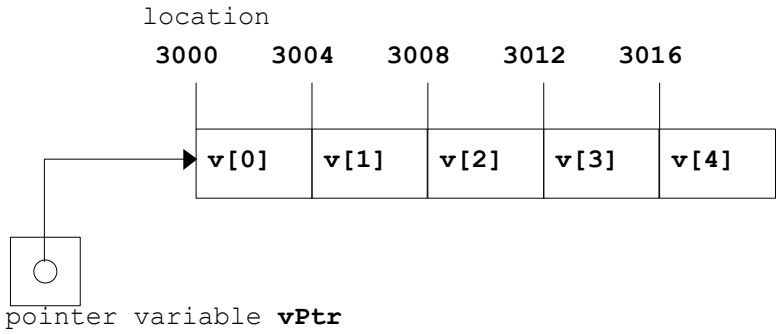
`swap` takes pointers (addresses of array elements) and dereferences them to modify the original array elements.

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
```

5.7 Pointer Expressions and Pointer Arithmetic

- Pointer arithmetic
  - Increment/decrement pointer (**++** or **--**)
  - Add/subtract an integer to/from a pointer(**+** or **+=** , **-** or **-=**)
  - Pointers may be subtracted from each other
  - Pointer arithmetic is meaningless unless performed on an array
- 5 element **int** array on a machine using 4 byte **int**s
  - **vPtr** points to first element **v[0]**, which is at location 3000
    - **vPtr = 3000**
  - **vPtr += 2**; sets **vPtr** to **3008**
    - **vPtr** points to **v[2]**



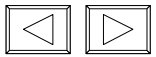
5.7 Pointer Expressions and Pointer Arithmetic

- Subtracting pointers
  - Returns the number of elements between two addresses

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

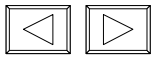
- Pointer comparison
  - Test which pointer points to the higher numbered array element
  - Test if a pointer points to **0** (**NULL**)

```
if ( vPtr == '0' )
   statement
```

5.7 Pointer Expressions and Pointer Arithmetic

- Pointers assignment
  - 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

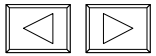


5.8 The Relationship Between Pointers and Arrays

- Arrays and pointers closely related
  - Array name like constant pointer
  - Pointers can do array subscripting operations
  - Having declared an array **`b[ 5 ]`** and a pointer **`bPtr`**
    - **`bPtr`** is equal to **`b`**

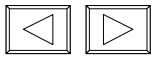
      `bptr == b`
    - **`bptr`** is equal to the address of the first element of **`b`**

      `bptr == &b[ 0 ]`



5.8 The Relationship Between Pointers and Arrays

- Accessing array elements with pointers
  - Element `b[ n ]` can be accessed by `*( bPtr + n )`
    - Called pointer/offset notation
  - Array itself can use pointer arithmetic.
    - `b[ 3 ]` same as `*(b + 3)`
  - Pointers can be subscripted (pointer/subscript notation)
    - `bPtr[ 3 ]` same as `b[ 3 ]`

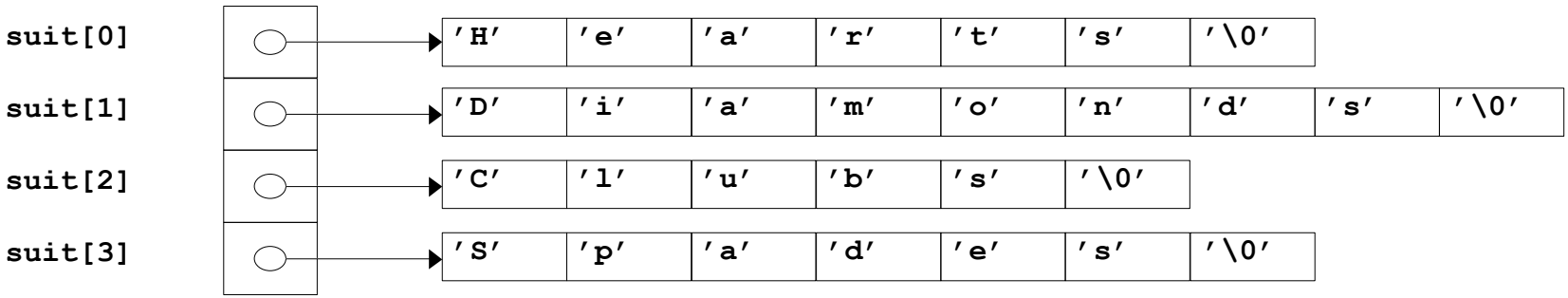


5.9 Arrays of Pointers

- Arrays can contain pointers
  - Commonly used to store an array of strings

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

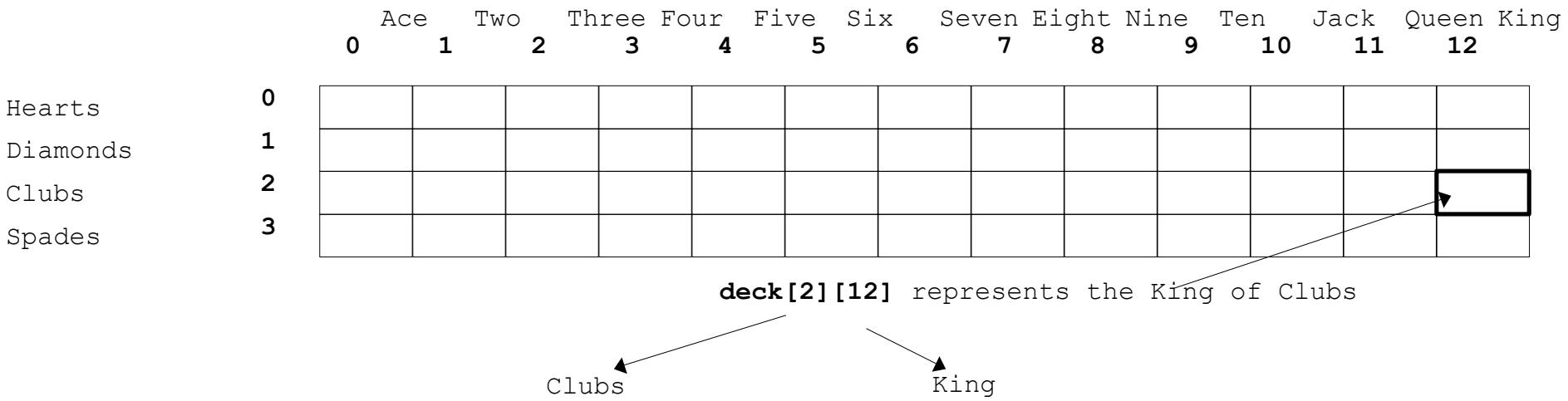
  - Each element of suit is a pointer to a **char *** (a string)
  - The strings are not in the array, only pointers to the strings are in the array



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

5.10 Case Study: A Card Shuffling and Dealing Simulation

- Card shuffling program
  - Use an array of pointers to strings, to store suit names
  - Use a double scripted array (suit by value)



  - Place 1-52 into the array to specify the order in which the cards are dealt

5.10 Case Study: A Card Shuffling and Dealing Simulation

- Pseudocode for shuffling and dealing simulation

First refinement

Initialize the suit array
Initialize the face array
Initialize the deck array

Shuffle the deck

Deal 52 cards

Second refinement

For each of the 52 cards

    Place card number in randomly selected unoccupied slot of deck

For each of the 52 cards

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

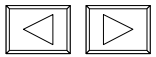
Third refinement

Choose slot of deck randomly

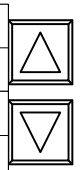
    While chosen slot of deck has been previously chosen
        Choose slot of deck randomly
    Place card number in chosen slot of deck

For each slot of the deck array

    If slot contains card number
        Print the face and suit of the card



```
1  // Fig. 5.24: fig05 24.cpp
2  // Card shuffling dealing program
3  #include <iostream>
4
5  using std::cout;
6  using std::ios;
7
8  #include <iomanip>
9
10 using std::setw;
11 using std::setiosflags;
12
13 #include <cstdlib>
14 #include <ctime>
15
16 void shuffle( int [][ 13 ] );
17 void deal( const int [][ 13 ], const char *[], const char *[] );
18
19 int main()
20 {
21    const char *suit[ 4 ] =
22       { "Hearts", "Diamonds", "Clubs", "Spades" };
23    const char *face[ 13 ] =
24       { "Ace", "Deuce", "Three", "Four",
25         "Five", "Six", "Seven", "Eight",
26         "Nine", "Ten", "Jack", "Queen", "King" };
27    int deck[ 4 ][ 13 ] = { 0 };
28
29    srand( time( 0 ) );
30
31    shuffle( deck );
32    deal( deck, face, suit );
33
```



Outline

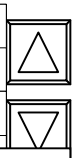
1. Initialize `suit` and `face` arrays

1.1 Initialize `deck` array

2. Call function `shuffle`

2.1 Call function `deal`

```
34    return 0;
35 }
36
37 void shuffle( int wDeck[][ 13 ] )
38 {
39    int row, column;
40
41    for ( int card = 1; card <= 52; card++ )
42       do {
43          row = rand() % 4;
44          column = rand() % 13;
45       } while( wDeck[ row ][ column ] != 0 );
46
47       wDeck[ row ][ column ] = card;
48    }
49 }
50
51 void deal( const int wDeck[][ 13 ], const char *wFace[],
52            const char *wSuit[] )
53 {
54    for ( int card = 1; card <= 52; card++ )
55
56       for ( int row = 0; row <= 3; row++ )
57
58          for ( int column = 0; column <= 12; column++ )
59
60             if ( wDeck[ row ][ column ] == card )
61                cout << setw( 5 ) << setiosflags( ios::right )
62                     << wFace[ column ] << " of "
63                     << setw( 8 ) << setiosflags( ios::left )
64                     << wSuit[ row ]
65                     << ( card % 2 == 0 ? '\n' : '\t' );
66 }
```



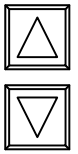
The numbers 1-52 are randomly placed into the **deck** array.

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

Outline

Define functions

```
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
```

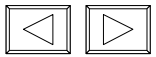


Outline

Program Output

5.11 Function Pointers

- Pointers to functions
  - Contain the address of the function
  - Similar to how an array name is the address of its 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



5.11 Function Pointers

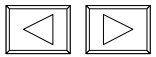
- Example: bubblesort
  - Function **bubble** takes a function pointer
    - The function determines whether the the array is sorted into ascending or descending sorting
  - The argument in **bubble** for the function pointer

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

    tells **bubble** to expect a pointer to a function that takes two **int**s and returns a **bool**
  - If the parentheses were left out

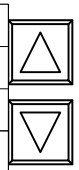
    ```
    bool *compare( int, int )
    ```

    would declare a function that receives two integers and returns a pointer to a **bool**



```
1  // Fig. 5.26: fig05 26.cpp
2  // Multipurpose sorting program using function pointers
3  #include <iostream>
4
5  using std::cout;
6  using std::cin;
7  using std::endl;
8
9  #include <iomanip>
10
11 using std::setw;
12
13 void bubble( int [], const int, bool (*)( int, int ) );
14 bool ascending( int, int );
15 bool descending( int, int );
16
17 int main()
18 {
19    const int arraySize = 10;
20    int order,
21        counter,
22        a[ arraySize ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
23
24    cout << "Enter 1 to sort in ascending order,\n"
25         << "Enter 2 to sort in descending order: ";
26    cin >> order;
27    cout << "\nData items in original order\n";
28
29    for ( counter = 0; counter < arraySize; counter++ )
30       cout << setw( 4 ) << a[ counter ];
31
32    if ( order == 1 ) {
33       bubble( a, arraySize, ascending );
34       cout << "\nData items in ascending order\n";
```

Notice the function pointer parameter.



Outline

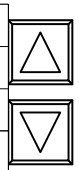
1. Initialize array

. Prompt for scending or descending sorting

2.1 Put appropriate function pointer into bubblesort

2.2 Call `bubble`

3. Print results



Outline

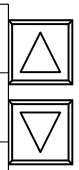
3.1 Define functions

```
35    }
36    else {
37       bubble( a, arraySize, descending );
38       cout << "\nData items in descending order\n";
39    }
40
41    for ( counter = 0; counter < arraySize; counter++ )
42       cout << setw( 4 ) << a[ counter ];
43
44    cout << endl;
45    return 0;
46 }
47
48 void bubble( int work[], const int size,
49              bool (*compare)( int, int ) )
50 {
51    void swap( int * const, int * const );   // prototype
52
53    for ( int pass = 1; pass < size; pass++ )
54
55       for ( int count = 0; count < size - 1; count++ )
56
57          if ( (*compare)( work[ count ], work[ count + 1 ] ) )
58             swap( &work[ count ], &work[ count + 1 ] );
59 }
60
61 void swap( int * const element1Ptr, int * const element2Ptr )
62 {
63    int temp;
64
65    temp = *element1Ptr;
66    *element1Ptr = *element2Ptr;
67    *element2Ptr = temp;
68 }
```

`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.

```
69
70 bool ascending( int a, int b )
71 {
72    return b < a;   // swap if b is less than a
73 }
74
75 bool descending( int a, int b )
76 {
77    return b > a;   // swap if b is greater than a
78 }
```



Outline

3.1 Define functions

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
```

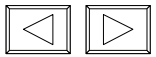
```
Enter 1 to sort in ascending order,
Enter 2 to sort in descending order: 2

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
```

5.12.1 Fundamentals of Characters and Strings

- Character constant
  - Integer value of a character
  - Single quotes
  - `'z'` is the integer value of **z**, which is **122**
- String
  - Series of characters treated as one unit
  - Can include letters, digits, special characters **+**, **-**, ***** ...
  - String literal (string constants)
    - Enclosed in double quotes, for example:

      ```
      "I like C++"
      ```
  - Array of characters, ends with null character `'\0'`
  - Strings are constant pointers (like arrays)
    - Value of string is the address of its first character



5.12.1 Fundamentals of Characters and Strings

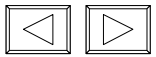
- String assignment
  - Character array:

```
char color[] = "blue";
```

    - Creates 5 element **char** array, **color**, (last element is **'\0'**)
  - variable of type **char ***

```
char *colorPtr = "blue";
```

    - Creates a pointer to string "**blue**", **colorPtr**, and stores it somewhere in memory



5.12.1 Fundamentals of Characters and Strings

- Reading strings
  - Assign input to character array **`word[ 20 ]`**

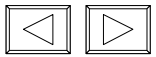
```
cin >> word
```

    - Reads characters until whitespace or EOF
    - String could exceed array size

```
cin >> setw( 20 ) >> word;
```

    - Reads 19 characters (space reserved for `'\0'`)
- **`cin.getline`**
  - Reads a line of text
  - Using **`cin.getline`**

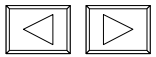
```
cin.getline( array, size, delimiter character);
```

5.12.1 Fundamentals of Characters and Strings

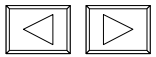
- **`cin.getline`**
  - Copies input into specified array until either
    - One less than the size is reached
    - The delimiter character is input
  - Example

```
char sentence[ 80 ];
cin.getline( sentence, 80, '\n' );
```

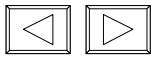
5.12.2 String Manipulation Functions of the String-handling Library

- String handling library **<cstring>** provides functions to
  - Manipulate strings
  - Compare strings
  - Search strings
  - Tokenize strings (separate them into logical pieces)
- ASCII character code
  - Strings are compared using their character codes
  - Easy to make comparisons (greater than, less than, equal to)
- Tokenizing
  - Breaking strings into tokens, separated by user-specified characters
  - Tokens are usually logical units, such as words (separated by spaces)
  - **"This is my string"** has 4 word tokens (separated by spaces)



5.12.2 String Manipulation Functions of the String-handling Library

`char *strcpy( char *s1, const char *s2 );`	Copies the string **s2** into the character array **s1**. The value of **s1** is returned.
`char *strncpy( char *s1, const char *s2, size_t n );`	Copies at most **n** characters of the string **s2** into the character array **s1**. The value of **s1** is returned.
`char *strcat( char *s1, const char *s2 );`	Appends the string **s2** to the string **s1**. The first character of **s2** overwrites the terminating null character of **s1**. The value of **s1** is returned.
`char *strncat( char *s1, const char *s2, size_t n );`	Appends at most **n** characters of string **s2** to string **s1**. The first character of **s2** overwrites the terminating null character of **s1**. The value of **s1** is returned.
`int strcmp( const char *s1, const char *s2 );`	Compares the string **s1** with the string **s2**. The function returns a value of zero, less than zero or greater than zero if **s1** is equal to, less than or greater than **s2**, respectively.



5.12.2 String Manipulation Functions of the String-handling Library (III)

`int strncmp( const char *s1, const char *s2, size_t n );`	Compares up to **n** characters of the string **s1** with the string **s2**. The function returns zero, less than zero or greater than zero if **s1** is equal to, less than or greater than **s2**, respectively.
`char *strtok( char *s1, const char *s2 );`	A sequence of calls to **strtok** breaks string **s1** into "tokens"—logical pieces such as words in a line of text—delimited by characters contained in string **s2**. The first call contains **s1** as the first argument, and subsequent calls to continue tokenizing the same string contain **NULL** as the first argument. A pointer to the current to-ken is returned by each call. If there are no more tokens when the function is called, **NULL** is returned.
`size_t strlen( const char *s );`	Determines the length of string **s**. The number of characters preceding the terminating null character is returned.