Lecture 18:
Pointers & Classes

Ioan Raicu
Department of Electrical Engineering & Computer Science
Northwestern University

EECS 211
Fundamentals of Computer Programming II
April 27th, 2010

digitalblasphemy

8.5 Using const with Pointers (cont.)

- There are four ways to pass a pointer to a function
  - a nonconstant pointer to nonconstant data
  - a nonconstant pointer to constant data (Fig. 8.10)
  - a constant pointer to nonconstant data (Fig. 8.11)
  - a constant pointer to constant data (Fig. 8.12)
- Each combination provides a different level of access privilege.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

8.5 Using const with Pointers (cont.)

- A nonconstant pointer to constant data
  - A pointer that can be modified to point to any data item of the appropriate type, but the data to which it points cannot be modified through that pointer.
- Might be used to receive an array argument to a function that will process each array element, but should not be allowed to modify the data.
- Any attempt to modify the data in the function results in a compilation error.
- Sample declaration:
    - `const int *countPtr;`
  - Read from right to left as "`countPtr` is a pointer to an integer constant."
- Figure 8.10 demonstrates the compilation error messages produced when attempting to compile a function that receives a nonconstant pointer to constant data, then tries to use that pointer to modify the data.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

8.5 Using const with Pointers (cont.)

```
1  // Fig. 8.10: fig08_10.cpp
2  // Attempting to modify data through a
3  // nonconstant pointer to constant data.
4
5  void f( const int * ); // prototype
6
7  int main()
8  {
9     int y;
10
11    f( &y ); // f attempts illegal modification
12 } // end main
13
14 // xPtr cannot modify the value of constant variable to which it points
15 void f( const int *xPtr )
16 {
17    *xPtr = 100; // error: cannot modify a const object
18 } // end function f
```

Microsoft Visual C++ compiler error message:

```
c:\cpphtp7_examples\ch08\Fig08_10\fig08_10.cpp(17) :
   error C3892: 'xPtr' : you cannot assign to a variable that is const
```

Fig. 8.10 | Attempting to modify data through a nonconstant pointer to constant data. (Part 1 of 2.)

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

8.5 Using const with Pointers (cont.)

GNU C++ compiler error message:

```
fig08_10.cpp: In function `void f(const int*)':
fig08_10.cpp:17: error: assignment of read-only location
```

Fig. 8.10 | Attempting to modify data through a nonconstant pointer to constant data. (Part 2 of 2.)

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

8.5 Using const with Pointers (cont.)

- A constant pointer to nonconstant data is a pointer that always points to the same memory location; the data at that location can be modified through the pointer.
- An example of such a pointer is an array name, which is a constant pointer to the beginning of the array.
- All data in the array can be accessed and changed by using the array name and array subscripting.
- A constant pointer to nonconstant data can be used to receive an array as an argument to a function that accesses array elements using array subscript notation.
- Pointers that are declared `const` must be initialized when they're declared.
- If the pointer is a function parameter, it's initialized with a pointer that's passed to the function.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

8.5 Using const with Pointers (cont.)

Common Programming Error 8.6

Not initializing a pointer that is declared `const` *is a compilation error.*

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

8.5 Using const with Pointers (cont.)

```
1  // Fig. 8.11: fig08_11.cpp
2  // Attempting to modify a constant pointer to nonconstant data.
3
4  int main()
5  {
6     int x, y;
7
8     // ptr is a constant pointer to an integer that can
9     // be modified through ptr, but ptr always points to the
10    // same memory location.
11    int * const ptr = &x; // const pointer must be initialized
12
13    *ptr = 7; // allowed: *ptr is not const
14    ptr = &y; // error: ptr is const; cannot assign to it a new address
15 } // end main
```

Microsoft Visual C++ compiler error message:

```
c:\cpphtp7_examples\ch08\Fig08_11\fig08_11.cpp(14) : error C3892: 'ptr' :
   you cannot assign to a variable that is const
```

GNU C++ compiler error message:

Fig. 8.11 | Attempting to modify a constant pointer to nonconstant data. (Part 1 of 2.)

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

8.5 Using const with Pointers (cont.)

```
fig08_11.cpp: In function `int main()':
fig08_11.cpp:14: error: assignment of read-only variable `ptr'
```

Fig. 8.11 | Attempting to modify a constant pointer to nonconstant data. (Part 2 of 2.)

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

8.5 Using `const` with Pointers (cont.)

- The minimum access privilege is granted by a constant pointer to constant data.
  - Such a pointer always points to the same memory location, and the data at that location cannot be modified via the pointer.
  - This is how an array should be passed to a function that *only reads the array, using array subscript notation, and does not modify the array.*
- The pro-gram of Fig. 8.12 declares pointer variable `ptr` to be of type `const int * const` (line 13).
- This declaration is read from right to left as "`ptr` is a constant pointer to an integer constant."
- The figure shows the error messages generated when an attempt is made to modify the data to which `ptr` points and when an attempt is made to modify the address stored in the pointer variable.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

8.5 Using const with Pointers (cont.)

```
1  // Fig. 8.12: fig08_12.cpp
2  // Attempting to modify a constant pointer to constant data.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     int x = 5, y;
9
10    // ptr is a constant pointer to a constant integer.
11    // ptr always points to the same location; the integer
12    // at that location cannot be modified.
13    const int *const ptr = &x;
14
15    cout << *ptr << endl;
16
17    *ptr = 7; // error: *ptr is const; cannot assign new value
18    ptr = &y; // error: ptr is const; cannot assign new address
19 } // end main
```

Microsoft Visual C++ compiler error message:

Fig. 8.12 | Attempting to modify a constant pointer to constant data. (Part 1 of 2.)

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

8.5 Using const with Pointers (cont.)

```
c:\cpphtp7_examples\ch08\Fig08_12\fig08_12.cpp(17) : error C3892: 'ptr' :
   you cannot assign to a variable that is const
c:\cpphtp7_examples\ch08\Fig08_12\fig08_12.cpp(18) : error C3892: 'ptr' :
   you cannot assign to a variable that is const
```

GNU C++ compiler error message:

```
fig08_12.cpp: In function `int main()':
fig08_12.cpp:17: error: assignment of read-only location
fig08_12.cpp:18: error: assignment of read-only variable `ptr'
```

Fig. 8.12 | Attempting to modify a constant pointer to constant data. (Part 2 of 2.)

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

8.8 Pointer Expressions and Pointer Arithmetic (cont.)

- A `void *` pointer cannot be dereferenced.
  - The com-piler must know the data type to determine the number of bytes to be dereferenced for a particular pointer—for a pointer to `void`, this number of bytes cannot be determined from the type.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

8.8 Pointer Expressions and Pointer Arithmetic (cont.)

- Pointers can be compared using equality and relational operators.
  - Comparisons using relational operators are meaningless unless the pointers point to members of the same array.
  - Pointer comparisons compare the addresses stored in the pointers.
- A common use of pointer comparison is determining whether a pointer is 0 (i.e., the pointer is a null pointer—it does not point to anything).

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

8.9 Relationship Between Pointers and Arrays

- An array name can be thought of as a constant pointer.
- Pointers can be used to do any operation involving array subscripting.
- Assume the following declarations:
  - ```
    int b[ 5 ]; // create 5-element int array b
    int *bPtr; // create int pointer bPtr
    ```
- Because the array name (without a subscript) is a (constant) pointer to the first element of the array, we can set `bPtr` to the address of the first element in array `b` with the statement
  - ```
    bPtr = b; // assign address of array b to bPtr
    ```
- equivalent to
  - ```
    bPtr = &b[ 0 ]; // also assigns address of
    array b to bPtr
    ```

©1992-2010 by Pearson Education, Inc. All Rights Reserved.
    ```

8.9 Relationship Between Pointers and Arrays (cont.)

- Array element `b[ 3 ]` can alternatively be referenced with the pointer expression
  - `*( bPtr + 3 )`
- The `3` in the preceding expression is the offset to the pointer.
- This notation is re-ferred to as pointer/offset notation.
  - The parentheses are necessary, because the precedence of `*` is higher than that of `+`.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

8.9 Relationship Between Pointers and Arrays (cont.)

- Just as the array element can be refer-enced with a pointer expression, the address
  - `&b[ 3 ]`
- can be written with the pointer expression
  - `bPtr + 3`
- The array name (which is implicitly `const`) can be treated as a pointer and used in pointer arithmetic.
- For exam-ple, the expression
  - `*( b + 3 )`
- also refers to the array element `b[ 3 ]`.
- In general, all subscripted array expressions can be written with a pointer and an offset.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

8.9 Relationship Between Pointers and Arrays (cont.)

- Pointers can be subscripted exactly as arrays can.
- For example, the expression
  - `bPtr[ 1 ]`
- refers to the array element `b[ 1 ]`; this expression uses pointer/subscript notation.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

8.9 Relationship Between Pointers and Arrays (cont.)

```
1  // Fig. 8.18: fig08_18.cpp
2  // Using subscripting and pointer notations with arrays.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     int b[] = { 10, 20, 30, 40 }; // create 4-element array b
9     int *bPtr = b; // set bPtr to point to array b
10
11    // output array b using array subscript notation
12    cout << "Array b printed with:\n\nArray subscript notation\n";
13
14    for ( int i = 0; i < 4; i++ )
15       cout << "b[" << i << "] = " << b[ i ] << '\n';
16
17    // output array b using the array name and pointer/offset notation
18    cout << "\nPointer/offset notation where "
19       << "the pointer is the array name\n";
20
21    for ( int offset1 = 0; offset1 < 4; offset1++ )
22       cout << "*(b + " << offset1 << ") = " << *( b + offset1 ) << '\n';
```

Fig. 8.18 | Referencing array elements with the array name and with pointers. (Part 1 of 3.)

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

8.9 Relationship Between Pointers and Arrays (cont.)

```
23
24      // output array b using bPtr and array subscript notation
25      cout << "\nPointer subscript notation\n";
26
27      for ( int j = 0; j < 4; j++ )
28         cout << "bPtr[" << j << "] = " << bPtr[ j ] << '\n';
29
30      cout << "\nPointer/offset notation\n";
31
32      // output array b using bPtr and pointer/offset notation
33      for ( int offset2 = 0; offset2 < 4; offset2++ )
34         cout << "*(bPtr + " << offset2 << ") = "
35            << *( bPtr + offset2 ) << '\n';
36   } // end main
```

```
Array b printed with:

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

Fig. 8.18 | Referencing array elements with the array name and with pointers. (Part 2 of 3.)

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

8.9 Relationship Between Pointers and Arrays (cont.)

```
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
```

Fig. 8.18 | Referencing array elements with the array name and with pointers. (Part 3 of 3.)

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

8.10 Pointer-Based String Processing (cont.)

- Characters are the fundamental building blocks of C++ source programs.
- Character constant
  - An integer value represented as a character in single quotes.
  - The value of a character constant is the integer value of the character in the machine's character set.
- A string is a series of characters treated as a single unit.
  - May include letters, digits and various special characters such as +, -, *, /and $.
- String literals, or string constants, in C++ are written in double quotation marks
- A pointer-based string is an array of characters ending with a null character ('₩0').
- A string is accessed via a pointer to its first character.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

8.10 Pointer-Based String Processing (cont.)

Common Programming Error 8.15

Not allocating sufficient space in a character array to store the null character that terminates a string is an error.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

8.10 Pointer-Based String Processing (cont.)

Common Programming Error 8.16

Creating or using a C-style string that does not contain a terminating null character can lead to logic errors.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

8.10 Pointer-Based String Processing (cont.)

Error-Prevention Tip 8.4

When storing a string of characters in a character array, be sure that the array is large enough to hold the largest string that will be stored. C++ allows strings of any length to be stored. If a string is longer than the character array in which it's to be stored, characters beyond the end of the array will overwrite data in memory following the array, leading to logic errors.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

8.10 Pointer-Based String Processing (cont.)

- Because a string is an array of characters, we can access individual characters in a string directly with array subscript notation.
- A string can be read into a character array using stream extraction with `cin`.
- The `setw` stream manipulator can be used to ensure that the string read into `word` does not exceed the size of the array.
  - Applies only to the next value being input.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

8.10 Pointer-Based String Processing (cont.)

- In some cases, it's desirable to input an entire line of text into a character array.
- For this purpose, the `cin` object provides the member function `getline`.
- Three arguments—a character array in which the line of text will be stored, a length and a delimiter character.
- The function stops reading characters when the delimiter character `'\n'` is encountered, when the end-of-file indicator is entered or when the number of characters read so far is one less than the length specified in the second argument.
- The third argument to `cin.getline` has `'\n'` as a default value.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

8.10 Pointer-Based String Processing (cont.)

- A character array representing a null-terminated string can be output with `cout` and <<.
- The characters of the string are output until a terminating null character is encountered; the null character is not printed.
- `cin` and `cout` assume that character arrays should be processed as strings terminated by null characters; `cin` and `cout` do not provide similar input and output processing capabilities for other array types.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

8.11 Arrays of Pointers

- Arrays may contain pointers.
- A common use of such a data structure is to form an array of pointer-based strings, referred to simply as a string array.
- Each entry in the array is a string, but in C++ a string is essentially a pointer to its first character, so each entry in an array of strings is simply a pointer to the first character of a string.
- ```
  const char * const suit[ 4 ] =
     { "Hearts", "Diamonds",
       "Clubs", "Spades" };
  ```
  - An array of four elements.
  - Each element is of type "pointer to `char` constant data."

©1992-2010 by Pearson Education, Inc. All Rights Reserved.
  ```

8.11 Arrays of Pointers

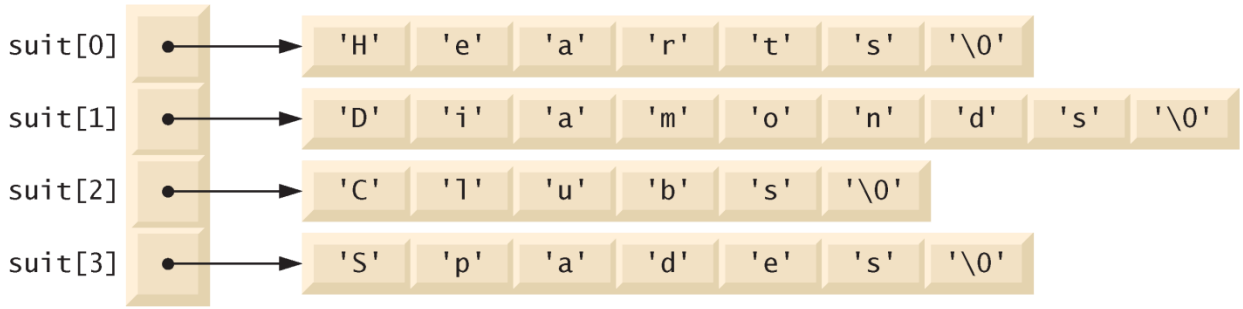

Fig. 8.19 | Graphical representation of the suit array.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

8.11 Arrays of Pointers (cont.)

- String arrays are commonly used with command-line arguments that are passed to function `main` when a program begins execution.
- Such arguments follow the program name when a program is executed from the command line.
- A typical use of command-line arguments is to pass options to a program.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

9.2 `Time` Class Case Study

- Our first example (Figs. 9.1–9.3) creates class `Time` and a driver program that tests the class.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

9.2 Time Class Case Study

```
1  // Fig. 9.1: Time.h
2  // Declaration of class Time.
3  // Member functions are defined in Time.cpp
4
5  // prevent multiple inclusions of header file
6  #ifndef TIME_H
7  #define TIME_H
8
9  // Time class definition
10 class Time
11 {
12 public:
13    Time(); // constructor
14    void setTime( int, int, int ); // set hour, minute and second
15    void printUniversal(); // print time in universal-time format
16    void printStandard(); // print time in standard-time format
17 private:
18    int hour; // 0 - 23 (24-hour clock format)
19    int minute; // 0 - 59
20    int second; // 0 - 59
21 }; // end class Time
22
23 #endif
```

Fig. 9.1 | `Time` class definition.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

9.2 Time Class Case Study (cont.)

- In Fig. 9.1, the class definition is enclosed in the following preprocessor wrapper:

```
// prevent multiple inclusions of header file
#ifndef TIME_H
#define TIME_H
   ...
#endif
```

  - Prevents the code between `#ifndef` and `#endif` from being included if the name `TIME_H` has been defined.
  - If the header has not been included previously in a file, the name `TIME_H` is defined by the `#define` directive and the header file statements are included.
  - If the header has been in-cluded previously, `TIME_H` is defined already and the header file is not included again.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

9.2 Time Class Case Study (cont.)

```
1  // Fig. 9.2: Time.cpp
2  // Member-function definitions for class Time.
3  #include <iostream>
4  #include <iomanip>
5  #include "Time.h" // include definition of class Time from Time.h
6  using namespace std;
7
8  // Time constructor initializes each data member to zero.
9  // Ensures all Time objects start in a consistent state.
10 Time::Time()
11 {
12    hour = minute = second = 0;
13 } // end Time constructor
14
15 // set new Time value using universal time; ensure that
16 // the data remains consistent by setting invalid values to zero
17 void Time::setTime( int h, int m, int s )
18 {
19    hour = ( h >= 0 && h < 24 ) ? h : 0; // validate hour
20    minute = ( m >= 0 && m < 60 ) ? m : 0; // validate minute
21    second = ( s >= 0 && s < 60 ) ? s : 0; // validate second
22 } // end function setTime
23
```

Fig. 9.2 | `Time` class member-function definitions. (Part 1 of 2.)

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

9.2 Time Class Case Study (cont.)

```
24  // print Time in universal-time format (HH:MM:SS)
25  void Time::printUniversal()
26  {
27     cout << setfill( '0' ) << setw( 2 ) << hour << ":"
28        << setw( 2 ) << minute << ":" << setw( 2 ) << second;
29  } // end function printUniversal
30
31  // print Time in standard-time format (HH:MM:SS AM or PM)
32  void Time::printStandard()
33  {
34     cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 ) << ":"
35        << setfill( '0' ) << setw( 2 ) << minute << ":" << setw( 2 )
36        << second << ( hour < 12 ? " AM" : " PM" );
37  } // end function printStandard
```

Fig. 9.2 | `Time` class member-function definitions. (Part 2 of 2.)

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

9.2 Time Class Case Study (cont.)

- Even though a member function declared in a class definition may be defined outside that class definition, that member function is still within that class's scope.
- If a member function is defined in the body of a class definition, the compiler attempts to inline calls to the member function.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

9.2 Time Class Case Study (cont.)

- Once class `Time` has been defined, it can be used as a type in object, array, pointer and reference declarations as follows:

```
Time sunset; // object of type Time
Time arrayOfTimes[ 5 ]; // array of 5 Time objects
Time &dinnerTime = sunset; // reference to a Time object
Time *timePtr = &dinnerTime; // pointer to a Time object
```

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

9.2 Time Class Case Study (cont.)

```
1  // Fig. 9.3: fig09_03.cpp
2  // Program to test class Time.
3  // NOTE: This file must be compiled with Time.cpp.
4  #include <iostream>
5  #include "Time.h" // include definition of class Time from Time.h
6  using namespace std;
7
8  int main()
9  {
10    Time t; // instantiate object t of class Time
11
12    // output Time object t's initial values
13    cout << "The initial universal time is ";
14    t.printUniversal(); // 00:00:00
15    cout << "\nThe initial standard time is ";
16    t.printStandard(); // 12:00:00 AM
17
18    t.setTime( 13, 27, 6 ); // change time
19
20    // output Time object t's new values
21    cout << "\n\nUniversal time after setTime is ";
22    t.printUniversal(); // 13:27:06
```

Fig. 9.3 | Program to test class `Time`. (Part 1 of 2.)

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

9.2 Time Class Case Study (cont.)

```
23      cout << "\nStandard time after setTime is ";
24      t.printStandard(); // 1:27:06 PM
25
26      t.setTime( 99, 99, 99 ); // attempt invalid settings
27
28      // output t's values after specifying invalid values
29      cout << "\n\nAfter attempting invalid settings:"
30         << "\nUniversal time: ";
31      t.printUniversal(); // 00:00:00
32      cout << "\nStandard time: ";
33      t.printStandard(); // 12:00:00 AM
34      cout << endl;
35   } // end main
```

```
The initial universal time is 00:00:00
The initial standard time is 12:00:00 AM

Universal time after setTime is 13:27:06
Standard time after setTime is 1:27:06 PM

After attempting invalid settings:
Universal time: 00:00:00
Standard time: 12:00:00 AM
```

Fig. 9.3 | Program to test class `Time`. (Part 2 of 2.)

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

9.2 Time Class Case Study (cont.)

- People new to object-oriented programming often suppose that objects must be quite large because they contain data members and member functions.
- Logically, this is true—you may think of objects as containing data and functions (and our discussion has certainly encouraged this view); physically, however, this is not true.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

9.2 `Time` Class Case Study (cont.)

Performance Tip 9.2

Objects contain only data, so objects are much smaller than if they also contained member functions. Applying operator `sizeof` *to a class name or to an object of that class will report only the size of the class's data members. The compiler creates one copy (only) of the member functions separate from all objects of the class. All objects of the class share this one copy. Each object, of course, needs its own copy of the class's data, because the data can vary among the objects. The function code is nonmodifiable and, hence, can be shared among all objects of one class.*

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

9.3 Class Scope and Accessing Class Members

- A class's data members and member functions belong to that class's scope.
- Nonmember functions are defined at global namespace scope.
- Within a class's scope, class members are immediately accessible by all of that class's member functions and can be referenced by name.
- Outside a class's scope, `public` class members are referenced through one of the handles on an object—an object name, a reference to an object or a pointer to an object.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

9.3 Class Scope and Accessing Class Members (cont.)

- Member functions of a class can be overloaded only by other member functions of that class.
- If a member function defines a variable with the same name as a variable with class scope, the class-scope variable is hidden by the block-scope variable in the local scope.
  - Such a hidden variable can be ac-cessed by preceding the variable name with the class name followed by the scope resolution operator (`::`).
- The dot member selection operator (`.`) is preceded by an object's name or with a reference to an object to access the object's members.
- The arrow member selection operator (`->`) is preceded by a pointer to an object to access the object's members.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

9.3 Class Scope and Accessing Class Members (cont.)

```
1  // Fig. 9.4: fig09_04.cpp
2  // Demonstrating the class member access operators . and ->
3  #include <iostream>
4  using namespace std;
5
6  // class Count definition
7  class Count
8  {
9  public: // public data is dangerous
10    // sets the value of private data member x
11    void setX( int value )
12    {
13       x = value;
14    } // end function setX
15
16    // prints the value of private data member x
17    void print()
18    {
19       cout << x << endl;
20    } // end function print
21
```

Fig. 9.4 | Accessing an object's member functions through each type of object handle—the object's name, a reference to the object and a pointer to the object. (Part 1 of 3.)

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

9.3 Class Scope and Accessing Class Members (cont.)

```
22  private:
23     int x;
24  }; // end class Count
25
26  int main()
27  {
28     Count counter; // create counter object
29     Count *counterPtr = &counter; // create pointer to counter
30     Count &counterRef = counter; // create reference to counteró
31
32     cout << "Set x to 1 and print using the object's name: ";
33     counter.setX( 1 ); // set data member x to 1
34     counter.print(); // call member function print
35
36     cout << "Set x to 2 and print using a reference to an object: ";
37     counterRef.setX( 2 ); // set data member x to 2
38     counterRef.print(); // call member function print
39
40     cout << "Set x to 3 and print using a pointer to an object: ";
41     counterPtr->setX( 3 ); // set data member x to 3
42     counterPtr->print(); // call member function print
43  } // end main
```

Fig. 9.4 | Accessing an object's member functions through each type of object handle—the object's name, a reference to the object and a pointer to the object. (Part 2 of 3.)

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

9.3 Class Scope and Accessing Class Members (cont.)

```
Set x to 1 and print using the object's name: 1
Set x to 2 and print using a reference to an object: 2
Set x to 3 and print using a pointer to an object: 3
```

Fig. 9.4 | Accessing an object's member functions through each type of object handle—the object's name, a reference to the object and a pointer to the object. (Part 3 of 3.)

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

9.4 Separating Interface from Implementation

- Separating classes into two files—a header file for the class definition (i.e., the class's interface) and a source code file for the class's member-function definitions (i.e., the class's implementation) makes it easier to modify programs.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

9.5 Access Functions and Utility Functions

- Access functions can read or display data.
- A common use for access functions is to test the truth or falsity of conditions—such functions are often called predicate functions.
- A utility function is a `private` member function that supports the operation of the class's `public` member functions.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

9.5 Access Functions and Utility Functions

```
1  // Fig. 9.5: SalesPerson.h
2  // SalesPerson class definition.
3  // Member functions defined in SalesPerson.cpp.
4  #ifndef SALESP_H
5  #define SALESP_H
6
7  class SalesPerson
8  {
9  public:
10    static const int monthsPerYear = 12; // months in one year
11    SalesPerson(); // constructor
12    void getSalesFromUser(); // input sales from keyboard
13    void setSales( int, double ); // set sales for a specific month
14    void printAnnualSales(); // summarize and print sales
15 private:
16    double totalAnnualSales(); // prototype for utility function
17    double sales[ monthsPerYear ]; // 12 monthly sales figures
18 }; // end class SalesPerson
19
20 #endif
```

Fig. 9.5 | `SalesPerson` class definition.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

9.5 Access Functions and Utility Functions

```
1  // Fig. 9.6: SalesPerson.cpp
2  // SalesPerson class member-function definitions.
3  #include <iostream>
4  #include <iomanip>
5  #include "SalesPerson.h" // include SalesPerson class definition
6  using namespace std;
7
8  // initialize elements of array sales to 0.0
9  SalesPerson::SalesPerson()
10 {
11    for ( int i = 0; i < monthsPerYear; i++ )
12       sales[ i ] = 0.0;
13 } // end SalesPerson constructor
14
15 // get 12 sales figures from the user at the keyboard
16 void SalesPerson::getSalesFromUser()
17 {
18    double salesFigure;
19
20    for ( int i = 1; i <= monthsPerYear; i++ )
21    {
22       cout << "Enter sales amount for month " << i << ": ";
23       cin >> salesFigure;
```

Fig. 9.6 | SalesPerson class member-function definitions. (Part 1 of 3.)

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

9.5 Access Functions and Utility Functions

```
24          setSales( i, salesFigure );
25       } // end for
26    } // end function getSalesFromUser
27
28    // set one of the 12 monthly sales figures; function subtracts
29    // one from month value for proper subscript in sales array
30    void SalesPerson::setSales( int month, double amount )
31    {
32       // test for valid month and amount values
33       if ( month >= 1 && month <= monthsPerYear && amount > 0 )
34          sales[ month - 1 ] = amount; // adjust for subscripts 0-11
35       else // invalid month or amount value
36          cout << "Invalid month or sales figure" << endl;
37    } // end function setSales
38
39    // print total annual sales (with the help of utility function)
40    void SalesPerson::printAnnualSales()
41    {
42       cout << setprecision( 2 ) << fixed
43          << "\nThe total annual sales are: $"
44          << totalAnnualSales() << endl; // call utility function
45    } // end function printAnnualSales
46
```

Fig. 9.6 | `SalesPerson` class member-function definitions. (Part 2 of 3.)

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

9.5 Access Functions and Utility Functions

```
47 // private utility function to total annual sales
48 double SalesPerson::totalAnnualSales()
49 {
50    double total = 0.0; // initialize total
51
52    for ( int i = 0; i < monthsPerYear; i++ ) // summarize sales results
53       total += sales[ i ]; // add month i sales to total
54
55    return total;
56 } // end function totalAnnualSales
```

Fig. 9.6 | `SalesPerson` class member-function definitions. (Part 3 of 3.)

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

9.5 Access Functions and Utility Functions

```
1  // Fig. 9.7: fig09_07.cpp
2  // Utility function demonstration.
3  // Compile this program with SalesPerson.cpp
4
5  // include SalesPerson class definition from SalesPerson.h
6  #include "SalesPerson.h"
7
8  int main()
9  {
10    SalesPerson s; // create SalesPerson object s
11
12    s.getSalesFromUser(); // note simple sequential code; there are
13    s.printAnnualSales(); // no control statements in main
14 } // end main
```

Fig. 9.7 | Utility function demonstration. (Part 1 of 2.)

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

9.5 Access Functions and Utility Functions

```
Enter sales amount for month 1: 5314.76
Enter sales amount for month 2: 4292.38
Enter sales amount for month 3: 4589.83
Enter sales amount for month 4: 5534.03
Enter sales amount for month 5: 4376.34
Enter sales amount for month 6: 5698.45
Enter sales amount for month 7: 4439.22
Enter sales amount for month 8: 5893.57
Enter sales amount for month 9: 4909.67
Enter sales amount for month 10: 5123.45
Enter sales amount for month 11: 4024.97
Enter sales amount for month 12: 5923.92

The total annual sales are: $60120.59
```

Fig. 9.7 | Utility function demonstration. (Part 2 of 2.)

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

Questions

©1992-2010 by Pearson Education, Inc. All Rights Reserved.