Instructor Note: *C How to Program*, Chapter 17 is a copy of *C++ How to Program, 9/e* Chapter 9. We have not renumbered the PowerPoint Slides.

Chapter 9
Classes: A Deeper Look; Throwing Exceptions

C++ How to Program, 9/e, GE

OBJECTIVES

In this chapter you'll:

- Use an include guard.
- Access class members via an object's name, a reference or a pointer.
- Use destructors to perform "termination housekeeping."
- Learn the order of constructor and destructor calls.
- Learn about the dangers of returning a reference to `private` data.
- Assign the data members of one object to those of another object.
- Create objects composed of other objects.
- Use `friend` functions and `friend` classes.
- Use the `this` pointer in a member function to access a non-`static` class member.
- Use `static` data members and member functions.

9.1 Introduction

- This chapter takes a deeper look at classes.
- Coverage includes:
  - The example also demonstrates using an *include guard* in headers to prevent header code from being included in the same source code file more than once.
  - We demonstrate how client code can access a class's `public` members via the name of an object, a reference to an object or a pointer to an object.
  - We discuss access functions that can read or write an object's data members.
  - We also demonstrate utility functions—`private` member functions that support the operation of the class's `public` member functions.

9.1 Introduction (cont.)

- Coverage includes (cont.):
  - How default arguments can be used in constructors.
  - Destructors that perform "termination housekeeping" on objects before they're destroyed.
  - The *order* in which constructors and destructors are called.
  - We show that returning a reference or pointer to `private` data *breaks the encapsulation* of a class, allowing client code to directly access an object's data.
  - We use default memberwise assignment to assign an object of a class to another object of the same class.

9.1 Introduction (cont.)

- Coverage includes (cont.):
  - `const` objects and `const` member functions to prevent modifications of objects and enforce the principle of least privilege.
  - *Composition*—a form of reuse in which a class can have objects of other classes as members.
  - *Friendship* to specify that a nonmember function can also access a class's non-public members—a technique that's often used in operator overloading for performance reasons.
  - `this` pointer, which is an implicit argument in all calls to a class's non-static member functions, allowing them to access the correct object's data members and non-static member functions.

9.2 `Time` Class Case Study

- Our first example (Fig. 9.1) creates class `Time` and tests the class.

Good Programming Practice 9.1

For clarity and readability, use each access specifier only once in a class definition. Place `public` members first, where they're easy to locate.

Software Engineering Observation 9.1

Each member of a class should have `private` visibility unless it can be proven that the element needs `public` visibility. This is another example of the principle of least privilege.

```
1  // Fig. 9.1: Time.h
2  // Time class definition.
3  // Member functions are defined in Time.cpp
4
5  // prevent multiple inclusions of header
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() const; // print time in universal-time format
16    void printStandard() const; // print time in standard-time format
17 private:
18    unsigned int hour; // 0 - 23 (24-hour clock format)
19    unsigned int minute; // 0 - 59
20    unsigned int second; // 0 - 59
21 }; // end class Time
22
23 #endif
```

Fig. 9.1 | Time class definition.

9.2 `Time` Class Case Study (cont.)

- In Fig. 9.1, the class definition is enclosed in the following include guard:

```
// 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 included previously, `TIME_H` is defined already and the header file is not included again.

Error-Prevention Tip 9.1

Use `#ifndef`, `#define` and `#endif` preprocessing directives to form an include guard that prevents headers from being included more than once in a source-code file.

Good Programming Practice 9.2

By convention, use the name of the header in uppercase with the period replaced by an underscore in the `#ifndef` and `#define` preprocessing directives of a header.

9.2 `Time` Class Case Study (cont.)

Time Class Member Functions

- In Fig. 9.2, the `Time` constructor (lines 11–14) initializes the data members to 0—the universal-time equivalent of 12 AM.
- Invalid values cannot be stored in the data members of a `Time` object, because the constructor is called when the `Time` object is created, and all subsequent attempts by a client to modify the data members are scrutinized by function `setTime` (discussed shortly).
- You can define *overloaded constructors* for a class.

```
1  // Fig. 9.2: Time.cpp
2  // Time class member-function definitions.
3  #include <iostream>
4  #include <iomanip>
5  #include <stdexcept> // for invalid_argument exception class
6  #include "Time.h" // include definition of class Time from Time.h
7
8  using namespace std;
9
10 // Time constructor initializes each data member to zero.
11 Time::Time()
12    : hour( 0 ), minute( 0 ), second( 0 )
13 {
14 } // end Time constructor
15
```

Fig. 9.2 | `Time` class member-function definitions. (Part 1 of 3.)

```
16  // set new Time value using universal time
17  void Time::setTime( int h, int m, int s )
18  {
19     // validate hour, minute and second
20     if ( ( h >= 0 && h < 24 ) && ( m >= 0 && m < 60 ) &&
21        ( s >= 0 && s < 60 ) )
22     {
23        hour = h;
24        minute = m;
25        second = s;
26     } // end if
27     else
28        throw invalid_argument(
29           "hour, minute and/or second was out of range" );
30  } // end function setTime
31
32  // print Time in universal-time format (HH:MM:SS)
33  void Time::printUniversal() const
34  {
35     cout << setfill( '0' ) << setw( 2 ) << hour << ":"
36        << setw( 2 ) << minute << ":" << setw( 2 ) << second;
37  } // end function printUniversal
38
```

Fig. 9.2 | Time class member-function definitions. (Part 2 of 3.)

```
39  // print Time in standard-time format (HH:MM:SS AM or PM)
40  void Time::printStandard() const
41  {
42     cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 ) << ":"
43        << setfill( '0' ) << setw( 2 ) << minute << ":" << setw( 2 )
44        << second << ( hour < 12 ? " AM" : " PM" );
45  } // end function printStandard
```

Fig. 9.2 | `Time` class member-function definitions. (Part 3 of 3.)

9.2 `Time` Class Case Study (cont.)

- Before C++11, only `static const int` data members (which you saw in Chapter 7) could be initialized where they were declared in the class body.
- For this reason, data members typically should be initialized by the class's constructor as *there is no default initialization for fundamental-type data members.*
- As of C++11, you can now use an *in-class initializer* to initialize any data member where it's declared in the class definition.

9.2 `Time` Class Case Study (cont.)

- Parameterized stream manipulator `setfill` specifies the fill character that is displayed when an integer is output in a field wider than the number of digits in the value.
- The fill characters appear to the *left* of the digits in the number, because the number is *right aligned* by default—for *left aligned* values, the fill characters would appear to the right.
- If the number being output fills the specified field, the fill character will not be displayed.
- Once the fill character is specified with `setfill`, it applies for *all* subsequent values that are displayed in fields wider than the value being displayed.

Error-Prevention Tip 9.2

Each sticky setting (such as a fill character or floating-point precision) should be restored to its previous setting when it's no longer needed. Failure to do so may result in incorrectly formatted output later in a program. Chapter 13, Stream Input/Output: A Deeper Look, discusses how to reset the fill character and precision.

9.2 `Time` Class Case Study (cont.)

Defining Member Functions Outside the Class Definition; Class Scope

- 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 class's body, the compiler attempts to inline calls to the member function.

Performance Tip 9.1

Defining a member function inside the class definition inlines the member function (if the compiler chooses to do so). This can improve performance.

Software Engineering Observation 9.2

Only the simplest and most stable member functions (i.e., whose implementations are unlikely to change) should be defined in the class header.

Software Engineering Observation 9.3

Using an object-oriented programming approach often simplifies function calls by reducing the number of parameters. This benefit derives from the fact that encapsulating data members and member functions within a class gives the member functions the right to access the data members.

Software Engineering Observation 9.4

Member functions are usually shorter than functions in non-object-oriented programs, because the data stored in data members have ideally been validated by a constructor or by member functions that store new data. Because the data is already in the object, the member-function calls often have no arguments or fewer arguments than function calls in non-object-oriented languages. Thus, the calls, the function definitions and the function prototypes are shorter. This improves many aspects of program development.

Error-Prevention Tip 9.3

The fact that member function calls generally take either no arguments or substantially fewer arguments than conventional function calls in non-object-oriented languages reduces the likelihood of passing the wrong arguments, the wrong types of arguments or the wrong number of arguments.

9.2 `Time` Class Case Study (cont.)

Using Class `Time`

- 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
array< Time, 5 > arrayOfTimes; // array of 5 Time objects
Time &dinnerTime = sunset; // reference to a Time object
Time *timePtr = &dinnerTime; // pointer to a Time object
```

- Figure 9.3 uses class `Time`.

```
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 <stdexcept> // for invalid_argument exception class
6  #include "Time.h" // include definition of class Time from Time.h
7  using namespace std;
8
9  int main()
10 {
11    Time t; // instantiate object t of class Time
12
13    // output Time object t's initial values
14    cout << "The initial universal time is ";
15    t.printUniversal(); // 00:00:00
16    cout << "\nThe initial standard time is ";
17    t.printStandard(); // 12:00:00 AM
18
19    t.setTime( 13, 27, 6 ); // change time
20
```

Fig. 9.3 | Program to test class `Time`. (Part 1 of 3.)

```
21      // output Time object t's new values
22      cout << "\n\nUniversal time after setTime is ";
23      t.printUniversal(); // 13:27:06
24      cout << "\nStandard time after setTime is ";
25      t.printStandard(); // 1:27:06 PM
26
27      // attempt to set the time with invalid values
28      try
29      {
30         t.setTime( 99, 99, 99 ); // all values out of range
31      } // end try
32      catch ( invalid_argument &e )
33      {
34         cout << "Exception: " << e.what() << endl;
35      } // end catch
36
37      // output t's values after specifying invalid values
38      cout << "\n\nAfter attempting invalid settings:"
39         << "\nUniversal time: ";
40      t.printUniversal(); // 13:27:06
41      cout << "\nStandard time: ";
42      t.printStandard(); // 1:27:06 PM
43      cout << endl;
44   } // end main
```

Fig. 9.3 | Program to test class `Time`. (Part 2 of 3.)

```
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

Exception: hour, minute and/or second was out of range

After attempting invalid settings:
Universal time: 13:27:06
Standard time: 1:27:06 PM
```

Fig. 9.3 | Program to test class `Time`. (Part 3 of 3.)

9.2 `Time` Class Case Study (cont.)

Object Size

- 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.

Performance Tip 9.2

Objects contain only data, so objects are much smaller than if they also contained member functions. 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.

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*, by default.
- 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.

9.3 Class Scope and Accessing Class Members (cont.)

Class Scope and Block Scope

- If a member function defines a variable with the same name as a variable with class scope, the class-scope variable is *hidden* in the function by the block-scope variable.
  - Such a hidden variable can be accessed by preceding the variable name with the class name followed by the scope resolution operator (`::`).

9.3 Class Scope and Accessing Class Members (cont.)

Dot (.) and Arrow (->) Member Selection Operators

- 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.

9.3 Class Scope and Accessing Class Members (cont.)

Accessing public Class Members Through Objects, References and Pointers

- Consider an Account class that has a public `setBalance` member function. Given the following declarations:

```
Account account; // an Account object
// accountRef refers to an Account object
Account &accountRef = account;
// accountPtr points to an Account object
Account *accountPtr = &account;
```

9.3 Class Scope and Accessing Class Members (cont.)

You can invoke member function setBalance using the dot (.) and arrow (->) member selection operators as follows:

```
// call setBalance via the Account object
account.setBalance( 123.45 );
// call setBalance via a reference to the Account object
accountRef.setBalance( 123.45 );
// call setBalance via a pointer to the Account object
accountPtr->setBalance( 123.45 );
```

9.4 Access Functions and Utility Functions

Access 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.

Utility Functions

- A utility function (also called a helper function) is a `private` member function that supports the operation of the class's other member functions.

9.5 `Time` Class Case Study: Constructors with Default Arguments

- The program of Figs. 9.4–9.6 enhances class `Time` to demonstrate how arguments are implicitly passed to a constructor.
- The constructor defined in Fig. 9.2 initialized `hour`, `minute` and `second` to 0 (i.e., midnight in universal time).
- Like other functions, constructors can specify *default arguments.*

```
1  // Fig. 9.4: Time.h
2  // Time class containing a constructor with default arguments.
3  // Member functions defined in Time.cpp.
4
5  // prevent multiple inclusions of header
6  #ifndef TIME_H
7  #define TIME_H
8
9  // Time class definition
10 class Time
11 {
12 public:
13    explicit Time( int = 0, int = 0, int = 0 ); // default constructor
14
15    // set functions
16    void setTime( int, int, int ); // set hour, minute, second
17    void setHour( int ); // set hour (after validation)
18    void setMinute( int ); // set minute (after validation)
19    void setSecond( int ); // set second (after validation)
20
```

Fig. 9.4 | `Time` class containing a constructor with default arguments. (Part 1 of 2.)

```
21      // get functions
22      unsigned int getHour() const; // return hour
23      unsigned int getMinute() const; // return minute
24      unsigned int getSecond() const; // return second
25
26      void printUniversal() const; // output time in universal-time format
27      void printStandard() const; // output time in standard-time format
28   private:
29      unsigned int hour; // 0 - 23 (24-hour clock format)
30      unsigned int minute; // 0 - 59
31      unsigned int second; // 0 - 59
32   }; // end class Time
33
34   #endif
```

Fig. 9.4 | `Time` class containing a constructor with default arguments. (Part 2 of 2.)

Software Engineering Observation 9.5

Any change to the default argument values of a function requires the client code to be recompiled (to ensure that the program still functions correctly).

```
1  // Fig. 9.5: Time.cpp
2  // Member-function definitions for class Time.
3  #include <iostream>
4  #include <iomanip>
5  #include <stdexcept>
6  #include "Time.h" // include definition of class Time from Time.h
7  using namespace std;
8
9  // Time constructor initializes each data member
10 Time::Time( int hour, int minute, int second )
11 {
12    setTime( hour, minute, second ); // validate and set time
13 } // end Time constructor
14
15 // set new Time value using universal time
16 void Time::setTime( int h, int m, int s )
17 {
18    setHour( h ); // set private field hour
19    setMinute( m ); // set private field minute
20    setSecond( s ); // set private field second
21 } // end function setTime
22
```

Fig. 9.5 | Member-function definitions for class `Time`. (Part 1 of 4.)

```
23  // set hour value
24  void Time::setHour( int h )
25  {
26     if ( h >= 0 && h < 24 )
27        hour = h;
28     else
29        throw invalid_argument( "hour must be 0-23" );
30  } // end function setHour
31
32  // set minute value
33  void Time::setMinute( int m )
34  {
35     if ( m >= 0 && m < 60 )
36        minute = m;
37     else
38        throw invalid_argument( "minute must be 0-59" );
39  } // end function setMinute
40
```

Fig. 9.5 | Member-function definitions for class `Time`. (Part 2 of 4.)

```
41  // set second value
42  void Time::setSecond( int s )
43  {
44     if ( s >= 0 && s < 60 )
45        second = s;
46     else
47        throw invalid_argument( "second must be 0-59" );
48  } // end function setSecond
49
50  // return hour value
51  unsigned int Time::getHour() const
52  {
53     return hour;
54  } // end function getHour
55
56  // return minute value
57  unsigned Time::getMinute() const
58  {
59     return minute;
60  } // end function getMinute
61
```

Fig. 9.5 | Member-function definitions for class `Time`. (Part 3 of 4.)

```
62  // return second value
63  unsigned Time::getSecond() const
64  {
65     return second;
66  } // end function getSecond
67
68  // print Time in universal-time format (HH:MM:SS)
69  void Time::printUniversal() const
70  {
71     cout << setfill( '0' ) << setw( 2 ) << getHour() << ":"
72        << setw( 2 ) << getMinute() << ":" << setw( 2 ) << getSecond();
73  } // end function printUniversal
74
75  // print Time in standard-time format (HH:MM:SS AM or PM)
76  void Time::printStandard() const
77  {
78     cout << ( ( getHour() == 0 || getHour() == 12 ) ? 12 : getHour() % 12 )
79        << ":" << setfill( '0' ) << setw( 2 ) << getMinute()
80        << ":" << setw( 2 ) << getSecond() << ( hour < 12 ? " AM" : " PM" );
81  } // end function printStandard
```

Fig. 9.5 | Member-function definitions for class `Time`. (Part 4 of 4.)

```
1  // Fig. 9.6: fig09_06.cpp
2  // Constructor with default arguments.
3  #include <iostream>
4  #include <stdexcept>
5  #include "Time.h" // include definition of class Time from Time.h
6  using namespace std;
7
8  int main()
9  {
10    Time t1; // all arguments defaulted
11    Time t2( 2 ); // hour specified; minute and second defaulted
12    Time t3( 21, 34 ); // hour and minute specified; second defaulted
13    Time t4( 12, 25, 42 ); // hour, minute and second specified
14
15    cout << "Constructed with:\n\nt1: all arguments defaulted\n  ";
16    t1.printUniversal(); // 00:00:00
17    cout << "\n  ";
18    t1.printStandard(); // 12:00:00 AM
19
20    cout << "\n\nt2: hour specified; minute and second defaulted\n  ";
21    t2.printUniversal(); // 02:00:00
22    cout << "\n  ";
23    t2.printStandard(); // 2:00:00 AM
24
```

Fig. 9.6 | Constructor with default arguments. (Part 1 of 3.)

```
25      cout << "\n\nt3: hour and minute specified; second defaulted\n  ";
26      t3.printUniversal(); // 21:34:00
27      cout << "\n  ";
28      t3.printStandard(); // 9:34:00 PM
29
30      cout << "\n\nt4: hour, minute and second specified\n  ";
31      t4.printUniversal(); // 12:25:42
32      cout << "\n  ";
33      t4.printStandard(); // 12:25:42 PM
34
35      // attempt to initialize t6 with invalid values
36      try
37      {
38         Time t5( 27, 74, 99 ); // all bad values specified
39      } // end try
40      catch ( invalid_argument &e )
41      {
42         cerr << "\n\nException while initializing t5: " << e.what() << endl;
43      } // end catch
44   } // end main
```

Fig. 9.6 | Constructor with default arguments. (Part 2 of 3.)

```
Constructed with:

t1: all arguments defaulted
  00:00:00
  12:00:00 AM

t2: hour specified; minute and second defaulted
  02:00:00
  2:00:00 AM

t3: hour and minute specified; second defaulted
  21:34:00
  9:34:00 PM

t4: hour, minute and second specified
  12:25:42
  12:25:42 PM

Exception while initializing t5: hour must be 0-23
```

Fig. 9.6 | Constructor with default arguments. (Part 3 of 3.)

9.5 `Time` Class Case Study: Constructors with Default Arguments (cont.)

Notes Regarding Class Time's Set and Get Functions and Constructor

- `Time`'s *set* and *get* functions are called throughout the class's body.
- In each case, these functions could have accessed the class's `private` data directly.
- Consider changing the representation of the time from three `int` values (requiring 12 bytes of memory on systems with four-byte `int`s) to a single `int` value representing the total number of seconds that have elapsed since midnight (requiring only four bytes of memory).
- If we made such a change, only the bodies of the functions that access the `private` data directly would need to change.
  - No need to modify the bodies of the other functions.

9.5 `Time` Class Case Study: Constructors with Default Arguments (cont.)

- Designing the class in this manner reduces the likelihood of programming errors when altering the class's implementation.
- Duplicating statements in multiple functions or constructors makes changing the class's internal data representation more difficult.

Software Engineering Observation 9.6

If a member function of a class already provides all or part of the functionality required by a constructor (or other member function) of the class, call that member function from the constructor (or other member function). This simplifies the maintenance of the code and reduces the likelihood of an error if the implementation of the code is modified. As a general rule: Avoid repeating code.

Common Programming Error 9.1

A constructor can call other member functions of the class, such as set or get functions, but because the constructor is initializing the object, the data members may not yet be initialized. Using data members before they have been properly initialized can cause logic errors.

9.5 `Time` Class Case Study: Constructors with Default Arguments (cont.)

C++11: Using List Initializers to Call Constructors

- C++11 now provides a uniform initialization syntax called list initializers that can be used to initialize any variable. Lines 11–13 of Fig. 9.6 can be written using list initializers as follows:

```
Time t2{ 2 }; // hour specified; minute and second defaulted
Time t3{ 21, 34 }; // hour and minute specified; second defaulted
Time t4{ 12, 25, 42 }; // hour, minute and second specified
```

or

```
Time t2 = { 2 }; // hour specified; minute and second defaulted
Time t3 = { 21, 34 }; // hour and minute specified; second defaulted
Time t4 = { 12, 25, 42 }; // hour, minute and second specified
```

- The form without the = is preferred.

9.5 `Time` Class Case Study: Constructors with Default Arguments (cont.)

C++11: Overloaded Constructors and Delegating Constructors

- A class's constructors and member functions can also be overloaded.
- Overloaded constructors typically allow objects to be initialized with different types and/or numbers of arguments.
- To overload a constructor, provide in the class definition a prototype for each version of the constructor, and provide a separate constructor definition for each overloaded version.
  - This also applies to the class's member functions.

9.6 `Time` Class Case Study: Constructors with Default Arguments (cont.)

- In Figs. 9.4–9.6, the Time constructor with three parameters had a default argument for each parameter. We could have defined that constructor instead as four overloaded constructors with the following prototypes:

```
Time(); // default hour, minute and second to 0
Time( int ); // initialize hour; default minute and second to 0
Time( int, int); // initialize hour and minute; default second to 0
Time( int, int, int ); // initialize hour, minute and second
```

- C++11 now allows constructors to call other constructors in the same class.
- The calling constructor is known as a delegating constructor—it *delegates* its work to another constructor.

9.5 `Time` Class Case Study: Constructors with Default Arguments (cont.)

- The first three of the four `Time` constructors declared on the previous slide can delegate work to one with three `int` arguments, passing 0 as the default value for the extra parameters.
- Use a member initializer with the name of the class as follows:

```
Time::Time()
   Time( 0, 0, 0 ) //delegate to Time( int, int, int )
{
} // end constructor with no arguments

Time::Time( int hour )
   Time( hour, 0, 0 ) //delegate to Time( int, int, int )
{
} // end constructor with one argument
Time::Time( int hour, int minute )
   Time( hour, minute, 0 ) //delegate to Time( int, int, int )
{
} // end constructor with two arguments
```

9.6 Destructors

- The name of the destructor for a class is the tilde character (~) followed by the class name.
- Called *implicitly* when an object is destroyed.
- *The destructor itself does not actually release the object's memory*—it performs termination housekeeping before the object's memory is reclaimed, so the memory may be reused to hold new objects.
- Receives no parameters and returns no value.
- May not specify a return type—not even `void`.
- A class has *one destructor*.
- A destructor must be `public`.
- If you do not *explicitly* define a destructor, the compiler defines an "empty" destructor.

9.7 When Constructors and Destructors Are Called

- Constructors and destructors are called implicitly.
- The order in which these function calls occur depends on the order in which execution enters and leaves the scopes where the objects are instantiated.
- Generally, destructor calls are made in the reverse order of the corresponding constructor calls
  - The storage classes of objects can alter the order in which destructors are called.

9.7 When Constructors and Destructors Are Called

Constructors and Destructors for Objects in Global Scope

- Constructors are called for objects defined in global scope (also called global namespace scope) *before* any other function (including `main`) in that program begins execution (although the order of execution of global object constructors between files is *not* guaranteed).
  - The corresponding destructors are called when `main` terminates.
- Function `exit` forces a program to terminate immediately and does *not* execute the destructors of local objects.
- Function `abort` performs similarly to function `exit` but forces the program to terminate *immediately*, without allowing the destructors of any objects to be called.

9.7 When Constructors and Destructors Are Called (cont.)

Constructors and Destructors for Local Objects

- Constructors and destructors for local objects are called each time execution enters and leaves the scope of the object.
- Destructors are not called for local objects if the program terminates with a call to function `exit` or function `abort`.

9.7 When Constructors and Destructors Are Called (cont.)

Constructors and Destructors for `static` Local Objects

- The constructor for a `static` local object is called only *once*, when execution first reaches the point where the object is defined—the corresponding destructor is called when `main` terminates or the program calls function `exit`.
- Global and `static` objects are destroyed in the reverse order of their creation.
- Destructors are not called for `static` objects if the program terminates with a call to function `abort`.

9.7 When Constructors and Destructors Are Called (cont.)

Demonstrating When Constructors and Destructors Are Called

- The program of Figs. 9.7–9.9 demonstrates the order in which constructors and destructors are called for objects of class `CreateAndDestroy` (Fig. 9.7 and Fig. 9.8) of various storage classes in several scopes.

```
1  // Fig. 9.7: CreateAndDestroy.h
2  // CreateAndDestroy class definition.
3  // Member functions defined in CreateAndDestroy.cpp.
4  #include <string>
5  using namespace std;
6
7  #ifndef CREATE_H
8  #define CREATE_H
9
10 class CreateAndDestroy
11 {
12 public:
13    CreateAndDestroy( int, string ); // constructor
14    ~CreateAndDestroy(); // destructor
15 private:
16    int objectID; // ID number for object
17    string message; // message describing object
18 }; // end class CreateAndDestroy
19
20 #endif
```

Fig. 9.7 | CreateAndDestroy class definition.

```
1  // Fig. 9.8: CreateAndDestroy.cpp
2  // CreateAndDestroy class member-function definitions.
3  #include <iostream>
4  #include "CreateAndDestroy.h"// include CreateAndDestroy class definition
5  using namespace std;
6
7  // constructor sets object's ID number and descriptive message
8  CreateAndDestroy::CreateAndDestroy( int ID, string messageString )
9     : objectID( ID ), message( messageString )
10 {
11    cout << "Object " << objectID << "   constructor runs   "
12       << message << endl;
13 } // end CreateAndDestroy constructor
14
15 // destructor
16 CreateAndDestroy::~CreateAndDestroy()
17 {
18    // output newline for certain objects; helps readability
19    cout << ( objectID == 1 || objectID == 6 ? "\n" : "" );
20
21    cout << "Object " << objectID << "   destructor runs    "
22       << message << endl;
23 } // end ~CreateAndDestroy destructor
```

Fig. 9.8 | CreateAndDestroy class member-function definitions.

```
1  // Fig. 9.9: fig09_09.cpp
2  // Order in which constructors and
3  // destructors are called.
4  #include <iostream>
5  #include "CreateAndDestroy.h" // include CreateAndDestroy class definition
6  using namespace std;
7
8  void create( void ); // prototype
9
10 CreateAndDestroy first( 1, "(global before main)" ); // global object
11
12 int main()
13 {
14    cout << "\nMAIN FUNCTION: EXECUTION BEGINS" << endl;
15    CreateAndDestroy second( 2, "(local automatic in main)" );
16    static CreateAndDestroy third( 3, "(local static in main)" );
17
18    create(); // call function to create objects
19
20    cout << "\nMAIN FUNCTION: EXECUTION RESUMES" << endl;
21    CreateAndDestroy fourth( 4, "(local automatic in main)" );
22    cout << "\nMAIN FUNCTION: EXECUTION ENDS" << endl;
23 } // end main
24
```

Fig. 9.9 | Order in which constructors and destructors are called. (Part 1 of 3.)

```
25  // function to create objects
26  void create( void )
27  {
28     cout << "\nCREATE FUNCTION: EXECUTION BEGINS" << endl;
29     CreateAndDestroy fifth( 5, "(local automatic in create)" );
30     static CreateAndDestroy sixth( 6, "(local static in create)" );
31     CreateAndDestroy seventh( 7, "(local automatic in create)" );
32     cout << "\nCREATE FUNCTION: EXECUTION ENDS" << endl;
33  } // end function create
```

Fig. 9.9 | Order in which constructors and destructors are called. (Part 2 of 3.)

```
Object 1   constructor runs   (global before main)

MAIN FUNCTION: EXECUTION BEGINS
Object 2   constructor runs   (local automatic in main)
Object 3   constructor runs   (local static in main)

CREATE FUNCTION: EXECUTION BEGINS
Object 5   constructor runs   (local automatic in create)
Object 6   constructor runs   (local static in create)
Object 7   constructor runs   (local automatic in create)

CREATE FUNCTION: EXECUTION ENDS
Object 7   destructor runs    (local automatic in create)
Object 5   destructor runs    (local automatic in create)

MAIN FUNCTION: EXECUTION RESUMES
Object 4   constructor runs   (local automatic in main)

MAIN FUNCTION: EXECUTION ENDS
Object 4   destructor runs    (local automatic in main)
Object 2   destructor runs    (local automatic in main)

Object 6   destructor runs    (local static in create)
Object 3   destructor runs    (local static in main)

Object 1   destructor runs    (global before main)
```

Fig. 9.9 | Order in which constructors and destructors are called. (Part 3 of 3.)

9.8 `Time` Class Case Study: A Subtle Trap—Returning a Reference or a Pointer to a `private` Data Member

- A reference to an object is an alias for the name of the object and, hence, may be used on the left side of an assignment statement.
- In this context, the reference makes a perfectly acceptable *lvalue* that can receive a value.
- Unfortunately a `public` member function of a class can return a reference to a `private` data member of that class.
- Such a reference return actually makes a call to that member function an alias for the `private` data member!
  - The function call can be used in any way that the `private` data member can be used, including as an *lvalue* in an assignment statement
  - The same problem would occur if a pointer to the `private` data were to be returned by the function.
- If a function returns a reference that's declared `const`, the reference is a non-modifiable *lvalue* and cannot be used to modify the data.

9.8 `Time` Class Case Study: A Subtle Trap—Returning a Reference or a Pointer to a `private` Data Member

- The program of Figs. 9.10–9.12 uses a simplified `Time` class (Fig. 9.10 and Fig. 9.11) to demonstrate returning a reference to a private data member with member function `badSetHour`.

```
1  // Fig. 9.10: Time.h
2  // Time class declaration.
3  // Member functions defined in Time.cpp
4
5  // prevent multiple inclusions of header
6  #ifndef TIME_H
7  #define TIME_H
8
9  class Time
10 {
11 public:
12    explicit Time( int = 0, int = 0, int = 0 );
13    void setTime( int, int, int );
14    unsigned int getHour() const;
15    unsigned int &badSetHour( int ); // dangerous reference return
16 private:
17    unsigned int hour;
18    unsigned int minute;
19    unsigned int second;
20 }; // end class Time
21
22 #endif
```

Fig. 9.10 | Time class declaration.

```
1  // Fig. 9.11: Time.cpp
2  // Time class member-function definitions.
3  #include <stdexcept>
4  #include "Time.h" // include definition of class Time
5  using namespace std;
6
7  // constructor function to initialize private data; calls member function
8  // setTime to set variables; default values are 0 (see class definition)
9  Time::Time( int hr, int min, int sec )
10 {
11    setTime( hr, min, sec );
12 } // end Time constructor
13
14 // set values of hour, minute and second
15 void Time::setTime( int h, int m, int s )
16 {
17    // validate hour, minute and second
18    if ( ( h >= 0 && h < 24 ) && ( m >= 0 && m < 60 ) &&
19       ( s >= 0 && s < 60 ) )
20    {
21       hour = h;
22       minute = m;
23       second = s;
24    } // end if
```

Fig. 9.11 | Time class member-function definitions. (Part 1 of 2.)

```
25      else
26         throw invalid_argument(
27            "hour, minute and/or second was out of range" );
28   } // end function setTime
29
30   // return hour value
31   unsigned int Time::getHour()
32   {
33      return hour;
34   } // end function getHour
35
36   // poor practice: returning a reference to a private data member.
37   unsigned int &Time::badSetHour( int hh )
38   {
39      if ( hh >= 0 && hh < 24 )
40         hour = hh;
41      else
42         throw invalid_argument( "hour must be 0-23" );
43
44      return hour; // dangerous reference return
45   } // end function badSetHour
```

Fig. 9.11 | `Time` class member-function definitions. (Part 2 of 2.)

Software Engineering Observation 9.7

Returning a reference or a pointer to a `private` data member breaks the encapsulation of the class and makes the client code dependent on the representation of the class's data. There are cases where doing this is appropriate—we'll show an example of this when we build our custom `Array` class in Section 10.10.

```
1  // Fig. 9.12: fig09_12.cpp
2  // Demonstrating a public member function that
3  // returns a reference to a private data member.
4  #include <iostream>
5  #include "Time.h" // include definition of class Time
6  using namespace std;
7
8  int main()
9  {
10    Time t; // create Time object
11
12    // initialize hourRef with the reference returned by badSetHour
13    int &hourRef = t.badSetHour( 20 ); // 20 is a valid hour
14
15    cout << "Valid hour before modification: " << hourRef;
16    hourRef = 30; // use hourRef to set invalid value in Time object t
17    cout << "\nInvalid hour after modification: " << t.getHour();
18
19    // Dangerous: Function call that returns
20    // a reference can be used as an lvalue!
21    t.badSetHour( 12 ) = 74; // assign another invalid value to hour
22
```

Fig. 9.12 | `public` member function that returns a reference to a `private` data member. (Part 1 of 2.)

```
23      cout << "\n\n************************************************\n"
24         << "POOR PROGRAMMING PRACTICE!!!!!!!!\n"
25         << "t.badSetHour( 12 ) as an lvalue, invalid hour: "
26         << t.getHour()
27         << "\n************************************************" << endl;
28   } // end main
```

```
Valid hour before modification: 20
Invalid hour after modification: 30

************************************************
POOR PROGRAMMING PRACTICE!!!!!!!!
t.badSetHour( 12 ) as an lvalue, invalid hour: 74
************************************************
```

Fig. 9.12 | `public` member function that returns a reference to a `private` data member. (Part 2 of 2.)

9.9 Default Memberwise Assignment

- The assignment operator (=) can be used to assign an object to another object of the same type.
- By default, such assignment is performed by memberwise assignment (also called copy assignment).
  - Each data member of the object on the right of the assignment operator is assigned individually to the *same* data member in the object on the *left* of the assignment operator.
- [*Caution:* Memberwise assignment can cause serious problems when used with a class whose data members contain pointers to dynamically allocated memory; we discuss these problems in Chapter 10 and show how to deal with them.]

```
1  // Fig. 9.13: Date.h
2  // Date class declaration.  Member functions are defined in Date.cpp.
3
4  // prevent multiple inclusions of header
5  #ifndef DATE_H
6  #define DATE_H
7
8  // class Date definition
9  class Date
10 {
11 public:
12    explicit Date( int = 1, int = 1, int = 2000 ); // default constructor
13    void print();
14 private:
15    unsigned int month;
16    unsigned int day;
17    unsigned int year;
18 }; // end class Date
19
20 #endif
```

Fig. 9.13 | `Date` class declaration.

```
1  // Fig. 9.14: Date.cpp
2  // Date class member-function definitions.
3  #include <iostream>
4  #include "Date.h" // include definition of class Date from Date.h
5  using namespace std;
6
7  // Date constructor (should do range checking)
8  Date::Date( int m, int d, int y )
9     : month( m ), day( d ), year( y )
10 {
11 } // end constructor Date
12
13 // print Date in the format mm/dd/yyyy
14 void Date::print()
15 {
16    cout << month << '/' << day << '/' << year;
17 } // end function print
```

Fig. 9.14 | `Date` class member-function definitions.

```
1  // Fig. 9.15: fig09_15.cpp
2  // Demonstrating that class objects can be assigned
3  // to each other using default memberwise assignment.
4  #include <iostream>
5  #include "Date.h" // include definition of class Date from Date.h
6  using namespace std;
7
8  int main()
9  {
10    Date date1( 7, 4, 2004 );
11    Date date2; // date2 defaults to 1/1/2000
12
13    cout << "date1 = ";
14    date1.print();
15    cout << "\ndate2 = ";
16    date2.print();
17
18    date2 = date1; // default memberwise assignment
19
20    cout << "\n\nAfter default memberwise assignment, date2 = ";
21    date2.print();
22    cout << endl;
23 } // end main
```

Fig. 9.15 | Class objects can be assigned to each other using default memberwise assignment. (Part 1 of 2.)

```
date1 = 7/4/2004
date2 = 1/1/2000

After default memberwise assignment, date2 = 7/4/2004
```

Fig. 9.15 | Class objects can be assigned to each other using default memberwise assignment. (Part 2 of 2.)

9.9 Default Memberwise Assignment (cont.)

- Objects may be passed as function arguments and may be returned from functions.
- Such passing and returning is performed using pass-by-value by default—a *copy* of the object is passed or returned.
  - C++ creates a new object and uses a copy constructor to copy the original object's values into the new object.
- For each class, the compiler provides a default copy constructor that copies each member of the original object into the corresponding member of the new object.
  - Copy constructors can cause serious problems when used with a class whose data members contain pointers to dynamically allocated memory.
- Chapter 10 discusses customized copy constructors.

9.10 `const` Objects and `const` Member Functions

- Some objects need to be modifiable and some do not.
- You may use keyword `const` to specify that an object *is not* modifiable and that any attempt to modify the object should result in a compilation error.
- The statement

```
const Time noon( 12, 0, 0 );
```

declares a `const` object `noon` of class `Time` and initializes it to 12 noon. It's possible to instantiate `const` and non-`const` objects of the same class.

Software Engineering Observation 9.8

Attempts to modify a `const` object are caught at compile time rather than causing execution-time errors.

Performance Tip 9.3

Declaring variables and objects `const` when appropriate can improve performance—compilers can perform optimizations on constants that cannot be performed on non-`const` variables.

9.10 `const` Objects and `const` Member Functions (cont.)

9.10 `const` Objects and `const` Member Functions (cont.)

- C++ *disallows member function calls for* `const` *objects unless the member functions themselves are also declared* `const`.
- This is true even for *get* member functions that do *not* modify the object.
- *This is also a key reason that we've declared as* `const` *all member-functions that do not modify the objects on which they're called.*
- A member function is specified as `const` both in its prototype by inserting the keyword `const` *after* the function's parameter list and, in the case of the function definition, before the left brace that begins the function *body*.

Common Programming Error 9.2

Defining as `const` a member function that modifies a data member of the object is a compilation error.

Common Programming Error 9.3

Defining as `const` a member function that calls a non-`const` member function of the class on the same object is a compilation error.

Common Programming Error 9.4

Invoking a non-`const` member function on a `const` object is a compilation error.

9.10 `const` Objects and `const` Member Functions (cont.)

- A constructor *must* be allowed to modify an object so that the object can be initialized properly.
- A destructor must be able to perform its termination housekeeping chores before an object's memory is reclaimed by the system.
- Attempting to declare a constructor or destructor `const` is a compilation error.
- The "`const`ness" of a `const` object is enforced from the time the constructor *completes* initialization of the object until that object's destructor is called.

9.10 `const` Objects and `const` Member Functions (cont.)

Using `const` and Non-`const` Member Functions

- The program of Fig. 9.16 uses class `Time` from Figs. 9.4–9.5, but removes `const` from function `printStandard`'s prototype and definition so that we can show a compilation error.

```
1  // Fig. 9.16: fig09_16.cpp
2  // const objects and const member functions.
3  #include "Time.h" // include Time class definition
4
5  int main()
6  {
7     Time wakeUp( 6, 45, 0 ); // non-constant object
8     const Time noon( 12, 0, 0 ); // constant object
9
10                              // OBJECT      MEMBER FUNCTION
11    wakeUp.setHour( 18 );     // non-const   non-const
12
13    noon.setHour( 12 );       // const       non-const
14
15    wakeUp.getHour();         // non-const   const
16
17    noon.getMinute();         // const       const
18    noon.printUniversal();    // const       const
19
20    noon.printStandard();     // const       non-const
21 } // end main
```

Fig. 9.16 | `const` objects and `const` member functions. (Part 1 of 2.)

Microsoft Visual C++ compiler error messages:

```
C:\examples\ch09\Fig09_16_18\fig09_18.cpp(13) : error C2662:
   'Time::setHour' : cannot convert 'this' pointer from 'const Time' to
   'Time &'
         Conversion loses qualifiers
C:\examples\ch09\Fig09_16_18\fig09_18.cpp(20) : error C2662:
   'Time::printStandard' : cannot convert 'this' pointer from 'const Time' to
   'Time &'
       Conversion loses qualifiers
```

Fig. 9.16 | `const` objects and `const` member functions. (Part 2 of 2.)

9.11 Composition: Objects as Members of Classes

- An `AlarmClock` object needs to know when it's supposed to sound its alarm, so why not include a `Time` object as a member of the `AlarmClock` class?
- Such a capability is called composition and is sometimes referred to as a *has-a* relationship—*a class can have objects of other classes as members*.
- The next program uses classes `Date` (Figs. 9.17–9.18) and `Employee` (Figs. 9.19–9.20) to demonstrate composition.

Software Engineering Observation 9.9

A common form of software reusability is composition, in which a class has objects of other types as members.

Software Engineering Observation 9.10

Data members are constructed in the order in which they're declared in the class definition (not in the order they're listed in the constructor's member initializer list) and before their enclosing class objects (sometimes called host objects) are constructed.

```
1  // Fig. 9.17: Date.h
2  // Date class definition; Member functions defined in Date.cpp
3  #ifndef DATE_H
4  #define DATE_H
5
6  class Date
7  {
8  public:
9     static const unsigned int monthsPerYear = 12; // months in a year
10    explicit Date( int = 1, int = 1, int = 1900 ); // default constructor
11    void print() const; // print date in month/day/year format
12    ~Date(); // provided to confirm destruction order
13 private:
14    unsigned int month; // 1-12 (January-December)
15    unsigned int day; // 1-31 based on month
16    unsigned int year; // any year
17
18    // utility function to check if day is proper for month and year
19    unsigned int checkDay( int ) const;
20 }; // end class Date
21
22 #endif
```

Fig. 9.17 | Date class definition.

```
1  // Fig. 9.18: Date.cpp
2  // Date class member-function definitions.
3  #include <array>
4  #include <iostream>
5  #include <stdexcept>
6  #include "Date.h" // include Date class definition
7  using namespace std;
8
9  // constructor confirms proper value for month; calls
10 // utility function checkDay to confirm proper value for day
11 Date::Date( int mn, int dy, int yr )
12 {
13    if ( mn > 0 && mn <= monthsPerYear ) // validate the month
14       month = mn;
15    else
16       throw invalid_argument( "month must be 1-12" );
17
18    year = yr; // could validate yr
19    day = checkDay( dy ); // validate the day
20
21    // output Date object to show when its constructor is called
22    cout << "Date object constructor for date ";
23    print();
24    cout << endl;
25 } // end Date constructor
```

Fig. 9.18 | `Date` class member-function definitions. (Part 1 of 3.)

```
26
27  // print Date object in form month/day/year
28  void Date::print() const
29  {
30     cout << month << '/' << day << '/' << year;
31  } // end function print
32
33  // output Date object to show when its destructor is called
34  Date::~Date()
35  {
36     cout << "Date object destructor for date ";
37     print();
38     cout << endl;
39  } // end ~Date destructor
40
```

Fig. 9.18 | `Date` class member-function definitions. (Part 2 of 3.)

```
41  // utility function to confirm proper day value based on
42  // month and year; handles leap years, too
43  unsigned int Date::checkDay( int testDay ) const
44  {
45     static const array< int, monthsPerYear + 1 > daysPerMonth =
46        { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
47
48     // determine whether testDay is valid for specified month
49     if ( testDay > 0 && testDay <= daysPerMonth[ month ] )
50        return testDay;
51
52     // February 29 check for leap year
53     if ( month == 2 && testDay == 29 && ( year % 400 == 0 ||
54        ( year % 4 == 0 && year % 100 != 0 ) ) )
55        return testDay;
56
57     throw invalid_argument( "Invalid day for current month and year" );
58  } // end function checkDay
```

Fig. 9.18 | `Date` class member-function definitions. (Part 3 of 3.)

```
1  // Fig. 9.19: Employee.h
2  // Employee class definition showing composition.
3  // Member functions defined in Employee.cpp.
4  #ifndef EMPLOYEE_H
5  #define EMPLOYEE_H
6
7  #include <string>
8  #include "Date.h" // include Date class definition
9
10 class Employee
11 {
12 public:
13    Employee( const std::string &, const std::string &,
14       const Date &, const Date & );
15    void print() const;
16    ~Employee(); // provided to confirm destruction order
17 private:
18    std::string firstName; // composition: member object
19    std::string lastName; // composition: member object
20    const Date birthDate; // composition: member object
21    const Date hireDate; // composition: member object
22 }; // end class Employee
23
24 #endif
```

Fig. 9.19 | Employee class definition showing composition.

```
1  // Fig. 9.20: Employee.cpp
2  // Employee class member-function definitions.
3  #include <iostream>
4  #include "Employee.h" // Employee class definition
5  #include "Date.h" // Date class definition
6  using namespace std;
7
8  // constructor uses member initializer list to pass initializer
9  // values to constructors of member objects
10 Employee::Employee( const string &first, const string &last,
11    const Date &dateOfBirth, const Date &dateOfHire )
12    : firstName( first ), // initialize firstName
13      lastName( last ), // initialize lastName
14      birthDate( dateOfBirth ), // initialize birthDate
15      hireDate( dateOfHire ) // initialize hireDate
16 {
17    // output Employee object to show when constructor is called
18    cout << "Employee object constructor: "
19       << firstName << ' ' << lastName << endl;
20 } // end Employee constructor
21
```

Fig. 9.20 | Employee class member-function definitions. (Part 1 of 2.)

```
22  // print Employee object
23  void Employee::print() const
24  {
25     cout << lastName << ", " << firstName << "  Hired: ";
26     hireDate.print();
27     cout << "  Birthday: ";
28     birthDate.print();
29     cout << endl;
30  } // end function print
31
32  // output Employee object to show when its destructor is called
33  Employee::~Employee()
34  {
35     cout << "Employee object destructor: "
36        << lastName << ", " << firstName << endl;
37  } // end ~Employee destructor
```

Fig. 9.20 | `Employee` class member-function definitions. (Part 2 of 2.)

9.11 Composition: Objects as Members of Classes (cont.)

`Employee` Constructor's Member Initializer List

- The *colon (`:`)* following the constructor's header (Fig. 9.20, line 12) begins the *member initializer list*.
- The member initializers specify the `Employee` constructor parameters being passed to the constructors of the `string` and `Date` data members.
- Again, member initializers are separated by commas.
- The order of the member initializers does not matter.
- They're executed in the order that the member objects are declared in class Employee.

Good Programming Practice 9.3

For clarity, list member initializers in the order that the class's data members are declared.

9.11 Composition: Objects as Members of Classes (cont.)

`Date` Class's Default Copy Constructor

- As we mentioned in Section 9.9, the compiler provides each class with a *default copy constructor* that copies each data member of the constructor's argument object into the corresponding member of the object being initialized.
- Chapter 10 discusses how you can define customized copy constructors.

9.11 Composition: Objects as Members of Classes (cont.)

Testing Classes `Date` and `Employee`

- Figure 9.21 creates two `Date` objects (lines 10–11) and passes them as arguments to the constructor of the `Employee` object created in line 12.
- Line 15 outputs the `Employee` object's data.
- When each `Date` object is created in lines 10–11, the `Date` constructor defined in lines 11–25 of Fig. 9.18 displays a line of output to show that the constructor was called (see the first two lines of the sample output).

9.11 Composition: Objects as Members of Classes (cont.)

- [Note: Line 12 of Fig. 9.21 causes two additional Date constructor calls that do not appear in the program's output. When each of the Employee's Date member objects is initialized in the Employee constructor's member-initializer list (Fig. 9.20, lines 14–15), the default copy constructor for class Date is called. Since this constructor is defined implicitly by the compiler, it does not contain any output statements to demonstrate when it's called.]

```
1  // Fig. 9.21: fig09_21.cpp
2  // Demonstrating composition--an object with member objects.
3  #include <iostream>
4  #include "Date.h" // Date class definition
5  #include "Employee.h" // Employee class definition
6  using namespace std;
7
8  int main()
9  {
10    Date birth( 7, 24, 1949 );
11    Date hire( 3, 12, 1988 );
12    Employee manager( "Bob", "Blue", birth, hire );
13
14    cout << endl;
15    manager.print();
16 } // end main
```

Fig. 9.21 | Demonstrating composition—an object with member objects. (Part 1 of 2.)

```
Date object constructor for date 7/24/1949
Date object constructor for date 3/12/1988
Employee object constructor: Bob Blue

Blue, Bob  Hired: 3/12/1988  Birthday: 7/24/1949
Employee object destructor: Blue, Bob
Date object destructor for date 3/12/1988
Date object destructor for date 7/24/1949
Date object destructor for date 3/12/1988
Date object destructor for date 7/24/1949
```

There are actually five constructor calls when an `Employee` is constructed—two calls to the `string` class's constructor (lines 12–13 of Fig. 9.20), two calls to the `Date` class's default copy constructor (lines 14–15 of Fig. 9.20) and

Fig. 9.21 | Demonstrating composition—an object with member objects. (Part 2 of 2.)

9.11 Composition: Objects as Members of Classes (cont.)

What Happens When You Do Not Use the Member Initializer List?

- If a member object is not initialized through a member initializer, the member object's *default constructor* will be called *implicitly*.
- Values, if any, established by the default constructor can be overridden by set functions.
- However, for complex initialization, this approach may require significant additional work and time.

Common Programming Error 9.5

A compilation error occurs if a member object is not initialized with a member initializer and the member object's class does not provide a default constructor (i.e., the member object's class defines one or more constructors, but none is a default constructor).

Performance Tip 9.4

Initialize member objects explicitly through member initializers. This eliminates the overhead of "doubly initializing" member objects—once when the member object's default constructor is called and again when set functions are called in the constructor body (or later) to initialize the member object.

Software Engineering Observation 9.11

If a data member is an object of another class, making that member object `public` does not violate the encapsulation and hiding of that member object's `private` members. But, it does violate the encapsulation and hiding of the containing class's implementation, so member objects of class types should still be `private`.

9.12 `friend` Functions and `friend` Classes

- A friend function of a class is a non-member function that has the right to access the `public` *and* `non-public` class members.
- Standalone functions, entire classes or member functions of other classes may be declared to be *friends* of another class.

9.12 `friend` Functions and `friend` Classes (cont.)

Declaring a friend

- To declare a function as a `friend` of a class, precede the function prototype in the class definition with keyword `friend`.
- To declare all member functions of class `ClassTwo` as friends of class `ClassOne`, place a declaration of the form

```
friend class ClassTwo;
```

- in the definition of class `ClassOne`.
- Friendship is *granted*, *not taken*—for class B to be a `friend` of class A, class A *must* explicitly declare that class B is its `friend`.
- Friendship is not *symmetric*—if class A is a friend of class B, you cannot infer that class B is a friend of class A.
- Friendship is not *transitive*—if class A is a friend of class B and class B is a friend of class C, you cannot infer that class A is a friend of class C.

9.12 `friend` Functions and `friend` Classes (cont.)

Modifying a Class's `private` Data with a Friend Function

- Figure 9.22 is a mechanical example in which we define `friend` function `setX` to set the private data member `x` of class `Count`.
- We place the `friend` declaration *first* in the class definition, even before `public` member functions are declared.
- Function `setX` is a stand-alone (global) function—it isn't a member function of class `Count`.
- For this reason, when `setX` is invoked for object counter, line 41 passes counter as an argument to `setX` rather than using a handle (such as the name of the object) to call the function, as in

```
counter.setX( 8 ); // error: setX not a member function
```

- If you remove the friend declaration in line 9, you'll receive error messages indicating that function `setX` cannot modify class `Count`'s `private` data member `x`.

```
1  //Fig. 9.22: fig09_22.cpp
2  // Friends can access private members of a class.
3  #include <iostream>
4  using namespace std;
5
6  // Count class definition
7  class Count
8  {
9     friend void setX( Count &, int ); // friend declaration
10 public:
11    // constructor
12    Count()
13       : x( 0 ) // initialize x to 0
14    {
15       // empty body
16    } // end constructor Count
17
```

Fig. 9.22 | Friends can access `private` members of a class. (Part 1 of 3.)

```
18      // output x
19      void print() const
20      {
21         cout << x << endl;
22      } // end function print
23   private:
24      int x; // data member
25   }; // end class Count
26
27   // function setX can modify private data of Count
28   // because setX is declared as a friend of Count (line 9)
29   void setX( Count &c, int val )
30   {
31      c.x = val; // allowed because setX is a friend of Count
32   } // end function setX
33
```

Fig. 9.22 | Friends can access `private` members of a class. (Part 2 of 3.)

```
34  int main()
35  {
36     Count counter; // create Count object
37
38     cout << "counter.x after instantiation: ";
39     counter.print();
40
41     setX( counter, 8 ); // set x using a friend function
42     cout << "counter.x after call to setX friend function: ";
43     counter.print();
44  } // end main
```

```
counter.x after instantiation: 0
counter.x after call to setX friend function: 8
```

Fig. 9.22 | Friends can access `private` members of a class. (Part 3 of 3.)

9.12 `friend` Functions and `friend` Classes (cont.)

- It would normally be appropriate to define function `setX` as a member function of class `Count`.
- It would also normally be appropriate to separate the program of Fig. 9.22 into three files:
  1. A header (e.g., `Count.h`) containing the `Count` class definition, which in turn contains the prototype of `friend` function `setX`
  2. An implementation file (e.g., `Count.cpp`) containing the definitions of class `Count`'s member functions and the definition of `friend` function `setX`
  3. A test program (e.g., `fig09_22.cpp`) with `main`.

9.12 `friend` Functions and `friend` Classes (cont.)

Overloaded `friend` Functions

- It's possible to specify overloaded functions as `friend`s of a class.
- Each function intended to be a `friend` must be explicitly declared in the class definition as a `friend` of the class.

Software Engineering Observation 9.12

Even though the prototypes for `friend` functions appear in the class definition, friends are not member functions.

Software Engineering Observation 9.13

Member access notions of `private`, `protected` and `public` are not relevant to `friend` declarations, so `friend` declarations can be placed anywhere in a class definition.

Good Programming Practice 9.4

Place all friendship declarations first inside the class definition's body and do not precede them with any access specifier.

9.13 Using the `this` Pointer

- Every object has access to its own address through a pointer called `this` (a C++ keyword).
- The `this` pointer is *not* part of the object itself—i.e., the memory occupied by the this pointer is not reflected in the result of a `sizeof` operation on the object.
- Rather, the this pointer is passed (by the compiler) as an *implicit* argument to each of the object's non-`static` member functions.

9.13 Using the `this` Pointer (cont.)

Using the `this` Pointer to Avoid Naming Collisions

- Member functions use the this pointer *implicitly* (as we've done so far) or *explicitly* to reference an object's data members and other member functions.
- A common *explicit* use of the `this` pointer is to avoid *naming conflicts* between a class's data members and member-function parameters (or other local variables).

9.13 Using the `this` Pointer (cont.)

- Consider the `Time` class's hour data member and `setHour` member function in Figs. 9.4–9.5.
- We could have defined `setHour` as:

```
// set hour value
void Time::setHour( int hour  )
{
   if ( hour >= 0 && hour < 24 )
      this->hour + hour; //use this pointer to access data member
   else
      throw invalid_argument( "hour must be 0-23" );
} // end function setHour
```

Error-Prevention Tip 9.4

To make your code clearer and more maintainable, and to avoid errors, never hide data members with local variable names.

9.13 Using the `this` Pointer (cont.)

Type of the `this` Pointer

- The type of the `this` pointer depends on the type of the object and whether the member function in which `this` is used is declared `const`.
- For example, in a non-`const` member function of class `Employee`, the `this` pointer has the type `Employee *`. In a `const` member function, the this pointer has the type `const Employee *`.

9.13 Using the `this` Pointer (cont.)

Implicitly and Explicitly Using the `this` Pointer to Access an Object's Data Members

- Figure 9.23 demonstrates the implicit and explicit use of the this pointer to enable a member function of class `Test` to print the private data `x` of a `Test` object.
- In the next example and in Chapter 10, we show some substantial and subtle examples of using `this`.

```
1  // Fig. 9.23: fig09_23.cpp
2  // Using the this pointer to refer to object members.
3  #include <iostream>
4  using namespace std;
5
6  class Test
7  {
8  public:
9     explicit Test( int = 0 ); // default constructor
10    void print() const;
11 private:
12    int x;
13 }; // end class Test
14
15 // constructor
16 Test::Test( int value )
17    : x( value ) // initialize x to value
18 {
19    // empty body
20 } // end constructor Test
21
```

Fig. 9.23 | using the `this` pointer to refer to object members. (Part 1 of 3.)

```
22  // print x using implicit and explicit this pointers;
23  // the parentheses around *this are required
24  void Test::print() const
25  {
26     // implicitly use the this pointer to access the member x
27     cout << "        x = " << x;
28
29     // explicitly use the this pointer and the arrow operator
30     // to access the member x
31     cout << "\n  this->x = " << this->x;
32
33     // explicitly use the dereferenced this pointer and
34     // the dot operator to access the member x
35     cout << "\n(*this).x = " << ( *this ).x << endl;
36  } // end function print
37
38  int main()
39  {
40     Test testObject( 12 ); // instantiate and initialize testObject
41
42     testObject.print();
43  } // end main
```

Fig. 9.23 | using the `this` pointer to refer to object members. (Part 2 of 3.)

```
        x = 12
  this->x = 12
(*this).x = 12
```

Fig. 9.23 | using the `this` pointer to refer to object members. (Part 3 of 3.)

9.13 Using the `this` Pointer (cont.)

Using the `this` Pointer to Enable Cascaded Function Calls

- Another use of the `this` pointer is to enable cascaded member-function calls—that is, invoking multiple functions in the same statement (as in line 12 of Fig. 9.26).
- The program of Figs. 9.24–9.26 modifies class `Time`'s `set` functions `setTime`, `setHour`, `setMinute` and `setSecond` such that each returns a reference to a `Time` object to enable cascaded member-function calls.
- Notice in Fig. 9.25 that the last statement in the body of each of these member functions returns `*this` (lines 23, 34, 45 and 56) into a return type of `Time &`.
- The program of Fig. 9.26 creates `Time` object `t` (line 9), then uses it in *cascaded member-function calls* (lines 12 and 24).

```
1  // Fig. 9.24: Time.h
2  // Cascading member function calls.
3
4  // Time class definition.
5  // Member functions defined in Time.cpp.
6  #ifndef TIME_H
7  #define TIME_H
8
9  class Time
10 {
11 public:
12    explicit Time( int = 0, int = 0, int = 0 ); // default constructor
13
14    // set functions (the Time & return types enable cascading)
15    Time &setTime( int, int, int ); // set hour, minute, second
16    Time &setHour( int ); // set hour
17    Time &setMinute( int ); // set minute
18    Time &setSecond( int ); // set second
19
20    // get functions (normally declared const)
21    unsigned int getHour() const; // return hour
22    unsigned int getMinute() const; // return minute
23    unsigned int getSecond() const; // return second
```

Fig. 9.24 | `Time` class modified to enable cascaded member-function calls. (Part 1 of 2.)

```
24
25    // print functions (normally declared const)
26    void printUniversal() const; // print universal time
27    void printStandard() const; // print standard time
28 private:
29    unsigned int hour; // 0 - 23 (24-hour clock format)
30    unsigned int minute; // 0 - 59
31    unsigned int second; // 0 - 59
32 }; // end class Time
33
34 #endif
```

Fig. 9.24 | `Time` class modified to enable cascaded member-function calls. (Part 2 of 2.)

```
1  // Fig. 9.25: Time.cpp
2  // Time class member-function definitions.
3  #include <iostream>
4  #include <iomanip>
5  #include <stdexcept>
6  #include "Time.h" // Time class definition
7  using namespace std;
8
9  // constructor function to initialize private data;
10 // calls member function setTime to set variables;
11 // default values are 0 (see class definition)
12 Time::Time( int hr, int min, int sec )
13 {
14    setTime( hr, min, sec );
15 } // end Time constructor
16
```

Fig. 9.25 | `Time` class member-function definitions modified to enable cascaded member-function calls. (Part 1 of 5.)

```
17  // set values of hour, minute, and second
18  Time &Time::setTime( int h, int m, int s ) // note Time & return
19  {
20     setHour( h );
21     setMinute( m );
22     setSecond( s );
23     return *this; // enables cascading
24  } // end function setTime
25
26  // set hour value
27  Time &Time::setHour( int h ) // note Time & return
28  {
29     if ( h >= 0 && h < 24 )
30        hour = h;
31     else
32        throw invalid_argument( "hour must be 0-23" );
33
34     return *this; // enables cascading
35  } // end function setHour
36
```

Fig. 9.25 | `Time` class member-function definitions modified to enable cascaded member-function calls. (Part 2 of 5.)

```
37  // set minute value
38  Time &Time::setMinute( int m ) // note Time & return
39  {
40     if ( m >= 0 && m < 60 )
41        minute = m;
42     else
43        throw invalid_argument( "minute must be 0-59" );
44
45     return *this; // enables cascading
46  } // end function setMinute
47
48  // set second value
49  Time &Time::setSecond( int s ) // note Time & return
50  {
51     if ( s >= 0 && s < 60 )
52        second = s;
53     else
54        throw invalid_argument( "second must be 0-59" );
55
56     return *this; // enables cascading
57  } // end function setSecond
```

Fig. 9.25 | `Time` class member-function definitions modified to enable cascaded member-function calls. (Part 3 of 5.)

```
58
59  // get hour value
60  unsigned int Time::getHour() const
61  {
62     return hour;
63  } // end function getHour
64
65  // get minute value
66  unsigned int Time::getMinute() const
67  {
68     return minute;
69  } // end function getMinute
70
71  // get second value
72  unsigned int Time::getSecond() const
73  {
74     return second;
75  } // end function getSecond
76
```

Fig. 9.25 | `Time` class member-function definitions modified to enable cascaded member-function calls. (Part 4 of 5.)

```
77  // print Time in universal-time format (HH:MM:SS)
78  void Time::printUniversal() const
79  {
80     cout << setfill( '0' ) << setw( 2 ) << hour << ":"
81        << setw( 2 ) << minute << ":" << setw( 2 ) << second;
82  } // end function printUniversal
83
84  // print Time in standard-time format (HH:MM:SS AM or PM)
85  void Time::printStandard() const
86  {
87     cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
88        << ":" << setfill( '0' ) << setw( 2 ) << minute
89        << ":" << setw( 2 ) << second << ( hour < 12 ? " AM" : " PM" );
90  } // end function printStandard
```

Fig. 9.25 | `Time` class member-function definitions modified to enable cascaded member-function calls. (Part 5 of 5.)

```
1  // Fig. 9.26: fig09_26.cpp
2  // Cascading member-function calls with the this pointer.
3  #include <iostream>
4  #include "Time.h" // Time class definition
5  using namespace std;
6
7  int main()
8  {
9     Time t; // create Time object
10
11    // cascaded function calls
12    t.setHour( 18 ).setMinute( 30 ).setSecond( 22 );
13
14    // output time in universal and standard formats
15    cout << "Universal time: ";
16    t.printUniversal();
17
18    cout << "\nStandard time: ";
19    t.printStandard();
20
21    cout << "\n\nNew standard time: ";
22
```

Fig. 9.26 | Cascading member-function calls with the `this` pointer. (Part 1 of 2.)

```
23      // cascaded function calls
24      t.setTime( 20, 20, 20 ).printStandard();
25      cout << endl;
26   } // end main
```

```
Universal time: 18:30:22
Standard time: 6:30:22 PM

New standard time: 8:20:20 PM
```

Fig. 9.26 | Cascading member-function calls with the `this` pointer. (Part 2 of 2.)

9.14 `static` Class Members

- In certain cases, only one copy of a variable should be *shared* by *all* objects of a class.
- A `static` data member is used for these and other reasons.
- Such a variable represents "class-wide" information, i.e., data that is shared by all instances and is not specific to any one object of the class.

Performance Tip 9.5

Use `static` data members to save storage when a single copy of the data for all objects of a class will suffice.

9.14 `static` Class Members (cont.)

Scope and Initialization of `static` Data Members

- static data members have *class scope.*
- A static data member must be initialized *exactly* once.
- Fundamental-type `static` data members are initialized by default to `0`.
- Prior to C++11, a `static const` data member of `int` or `enum` type could be initialized in its declaration in the class definition and all other `static` data members had to be defined and initialized *at global namespace scope* (i.e., outside the body of the class definition).
- Again, C++11's in-class initializers also allow you to initialize these variables where they're declared in the class definition.

9.14 `static` Class Members (cont.)

Accessing `static` Data Members

- A class's `private` and `protected` `static` members are normally accessed through the class's `public` member functions or friends.
- *A class's `static` members exist even when no objects of that class exist.*
- To access a `public static` class member when no objects of the class exist, simply prefix the class name and the scope resolution operator (`::`) to the name of the data member.
- To access a private or protected static class member when no objects of the class exist, provide a public `static` member function and call the function by prefixing its name with the class name and scope resolution operator.
- A `static` member function is a service of the *class*, *not* of a specific *object* of the class.

Software Engineering Observation 9.14

A class's `static` data members and `static` member functions exist and can be used even if no objects of that class have been instantiated.

9.14 `static` Class Members (cont.)

Demonstrating `static` Data Members

- The program of Figs. 9.27–9.29 demonstrates a `private` `static` data member called `count` (Fig. 9.27, line 24) and a `public` `static` member function called `getCount` (Fig. 9.27, line 18).

```
1  // Fig. 9.27: Employee.h
2  // Employee class definition with a static data member to
3  // track the number of Employee objects in memory
4  #ifndef EMPLOYEE_H
5  #define EMPLOYEE_H
6
7  #include <string>
8
9  class Employee
10 {
11 public:
12    Employee( const std::string &, const std::string & ); // constructor
13    ~Employee(); // destructor
14    std::string getFirstName() const; // return first name
15    std::string getLastName() const; // return last name
16
17    // static member function
18    static unsigned int getCount(); // return # of objects instantiated
19 private:
20    std::string firstName;
21    std::string lastName;
22
```

Fig. 9.27 | `Employee` class definition with a static data member to track the number of `Employee` objects in memory. (Part 1 of 2.)

```
23      // static data
24      static unsigned int count; // number of objects instantiated
25   }; // end class Employee
26
27   #endif
```

Fig. 9.27 | `Employee` class definition with a static data member to track the number of `Employee` objects in memory. (Part 2 of 2.)

```
1  // Fig. 9.28: Employee.cpp
2  // Employee class member-function definitions.
3  #include <iostream>
4  #include "Employee.h" // Employee class definition
5  using namespace std;
6
7  // define and initialize static data member at global namespace scope
8  unsigned int Employee::count = 0; // cannot include keyword static
9
10 // define static member function that returns number of
11 // Employee objects instantiated (declared static in Employee.h)
12 unsigned int Employee::getCount()
13 {
14    return count;
15 } // end static function getCount
16
17 // constructor initializes non-static data members and
18 // increments static data member count
19 Employee::Employee( const string &first, const string &last )
20    : firstName( first ), lastName( last )
21 {
```

Fig. 9.28 | Employee class member-function definitions. (Part 1 of 2.)

```
22      ++count; // increment static count of employees
23      cout << "Employee constructor for " << firstName
24         << ' ' << lastName << " called." << endl;
25   } // end Employee constructor
26
27   // destructor deallocates dynamically allocated memory
28   Employee::~Employee()
29   {
30      cout << "~Employee() called for " << firstName
31         << ' ' << lastName << endl;
32      --count; // decrement static count of employees
33   } // end ~Employee destructor
34
35   // return first name of employee
36   string Employee::getFirstName() const
37   {
38      return firstName; // return copy of first name
39   } // end function getFirstName
40
41   // return last name of employee
42   string Employee::getLastName() const
43   {
44      return lastName; // return copy of last name
45   } // end function getLastName
```

Fig. 9.28 | Employee class member-function definitions. (Part 2 of 2.)

```
1  // Fig. 9.29: fig09_29.cpp
2  // static data member tracking the number of objects of a class.
3  #include <iostream>
4  #include "Employee.h" // Employee class definition
5  using namespace std;
6
7  int main()
8  {
9     // no objects exist; use class name and binary scope resolution
10    // operator to access static member function getCount
11    cout << "Number of employees before instantiation of any objects is "
12       << Employee::getCount() << endl; // use class name
13
14    // the following scope creates and destroys
15    // Employee objects before main terminates
16    {
17       Employee e1( "Susan", "Baker" );
18       Employee e2( "Robert", "Jones" );
19
```

Fig. 9.29 | `static` data member tracking the number of objects of a class. (Part 1 of 3.)

```
20         // two objects exist; call static member function getCount again
21         // using the class name and the scope resolution operator
22         cout << "Number of employees after objects are instantiated is "
23            << Employee::getCount();
24
25         cout << "\n\nEmployee 1: "
26            << e1.getFirstName() << " " << e1.getLastName()
27            << "\nEmployee 2: "
28            << e2.getFirstName() << " " << e2.getLastName() << "\n\n";
29      } // end nested scope in main
30
31      // no objects exist, so call static member function getCount again
32      // using the class name and the scope resolution operator
33      cout << "\nNumber of employees after objects are deleted is "
34         << Employee::getCount() << endl;
35   } // end main
```

Fig. 9.29 | `static` data member tracking the number of objects of a class. (Part 2 of 3.)

```
Number of employees before instantiation of any objects is 0
Employee constructor for Susan Baker called.
Employee constructor for Robert Jones called.
Number of employees after objects are instantiated is 2

Employee 1: Susan Baker
Employee 2: Robert Jones

~Employee() called for Robert Jones
~Employee() called for Susan Baker

Number of employees after objects are deleted is 0
```

Fig. 9.29 | `static` data member tracking the number of objects of a class. (Part 3 of 3.)

Common Programming Error 9.6

Using the `this` pointer in a `static` member function is a compilation error.

Common Programming Error 9.7

Declaring a `static` member function `const` is a compilation error. The `const` qualifier indicates that a function cannot modify the contents of the object on which it operates, but `static` member functions exist and operate independently of any objects of the class.