Classes: A Deeper Look

Based on Chapter 9 of C++ How to Program, 10/e

OBJECTIVES

In this chapter you'll:

- Engineer a class to separate its interface from its implementation and encourage reuse.
- Access class members via an object's name or a reference using the dot (`.`) operator.
- Access class members via a pointer to an object using the arrow (`->`) operator.
- Use destructors to perform "termination housekeeping."
- Learn the order of constructor and destructor calls.
- Learn about the dangers of returning a reference or a pointer 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 learn how to declare `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:
  - Using an *include guard* in a header to prevent header code from being included in the same source code file more than once.
  - Using an `ostringstream` to create string representations of objects.
  - An overview of the compile/link process.
  - Accessing object members via the object's name, a reference to an object and a pointer to an object.
  - Access functions that can read or write an object's data members.
  - 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.
  - How returning a reference or pointer to `private` data *breaks the encapsulation* of a class, allowing client code to directly access an object's data.
  - 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

- Prior class-definition examples placed a class in a header for reuse, then included the header into a source-code file containing `main`
  - Reveals the entire implementation of the class to the class's clients
- Client code (e.g., main) needs to know only
  - what member functions to call
  - what arguments to provide to each member function, and
  - what return type to expect from each member function.
- The client code does not need to know how those functions are implemented.

9.2 `Time` Class Case Study

- If client code *does* know how a class is implemented, the programmer might write client code based on the class's implementation details.
- If an implementation changes, the class's clients should not have to change.
- Hiding the class's implementation details makes it easier to change the class's implementation while minimizing, and hopefully eliminating, changes to client code.

9.2.1 Interface of a Class

- Interfaces define and standardize the ways in which things such as people and systems interact with one another
  - A radio's controls serve as an interface between the radio's users and its internal components.
  - The controls allow users to perform a limited set of operations (such as changing the station, adjusting the volume, and choosing between AM and FM stations)
  - Various radios may implement these operations differently—some provide push buttons, some provide dials and some support voice commands
  - The interface specifies what operations a radio permits users to perform but does not specify how the operations are implemented inside the radio.

9.2.1 Interface of a Class (cont.)

- Interface of a class describes what services a class's clients can use and how to request those services, but not how the class carries out the services
- A class's public interface consists of the class's public member functions (also known as the class's public services)
- You can specify a class's interface by writing a class definition that lists only the class's member-function prototypes and the class's data members

9.2.2 Separating the Interface from the Implementation

- To separate the class's interface from its implementation, we break up class `Time` into
  - header `Time.h` (Fig. 9.1) in which class `Time` is defined, and
  - source-code file `Time.cpp` (Fig. 9.2) in which `Time`'s member functions are defined
- Benefits
  - The class is reusable
  - The clients of the class know what member functions the class provides, how to call them and what return types to expect, and
  - the clients do not know how the class's member functions are implemented.

9.2.2 Separating the Interface from the Implementation

- By convention, member-function definitions are placed in a source-code file of the same base name (e.g., `Time`) as the class's header but with a `.cpp` filename extension
- The source-code file in Fig. 9.3 defines function `main` (the client code).
- Section 9.3 shows a diagram and explains how this three-file program is compiled from the perspectives of the `Time` class programmer and the client-code programmer—and what the `Time` application user sees.

9.2.3 `Time` Class Definition

- The header `Time.h` (Fig. 9.1) contains `Time`'s class definition
- Function prototypes (lines 13–15) describe the class's `public` interface without revealing the member-function implementations
  - The function prototype in line 13 indicates that `setTime` requires three `int` parameters and returns `void`
  - The prototypes for member functions `toUniversalString` and `toStandardString` (lines 14–15) each specify that the function takes no arguments and returns a `string`.

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

Fig. 9.1 | Time class definition.

9.2.3 `Time` Class Definition

- The header still specifies the class's `private` data members (lines 17–19)
  - each uses a C++11 in-class list initializer to set the data member to 0
- Compiler must know the data members of the class to determine how much memory to reserve for each object of the class
- Including the header `Time.h` in the client code (line 6 of Fig. 9.3) provides the compiler with the information it needs to ensure that the client code calls the member functions of class `Time` correctly.

9.2.3 Time Class Definition

- 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` (which means "if not defined") and `#endif` from being `#included` if the name `TIME_H` has been defined
  - When `Time.h` is `#included` the first time, the identifier `TIME_H` is not yet defined
    - `#define` directive defines `TIME_H` and the preprocessor includes the `Time.h` header's contents in the `.cpp` file
  - If the header is `#included` again, `TIME_H` is defined already and the code between `#ifndef` and `#endif` is ignored by the preprocessor

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 9.1

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.*

9.2.4 `Time` Class Member Functions

- The source-code file `Time.cpp` (Fig. 9.2) defines class `Time`'s member functions, which were declared in lines 13–15 of Fig. 9.1.
- For `const` member functions, the `const` keyword must appear in both the function prototypes (Fig. 9.1, lines 14–15) and the function definitions (Fig. 9.2, lines 26 and 34)

```
1  // Fig. 9.2: Time.cpp
2  // Time class member-function definitions.
3  #include <iomanip> // for setw and setfill stream manipulators
4  #include <stdexcept> // for invalid_argument exception class
5  #include <sstream> // for ostringstream class
6  #include <string>
7  #include "Time.h" // include definition of class Time from Time.h
8
9  using namespace std;
10
11 // set new Time value using universal time
12 void Time::setTime(int h, int m, int s) {
13    // validate hour, minute and second
14    if ((h >= 0 && h < 24) && (m >= 0 && m < 60) && (s >= 0 && s < 60)) {
15       hour = h;
16       minute = m;
17       second = s;
18    }
19    else {
20       throw invalid_argument(
21          "hour, minute and/or second was out of range");
22    }
23 }
```

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

```
24
25  // return Time as a string in universal-time format (HH:MM:SS)
26  string Time::toUniversalString() const {
27     ostringstream output;
28     output << setfill('0') << setw(2) << hour << ":"
29        << setw(2) << minute << ":" << setw(2) << second;
30     return output.str(); // returns the formatted string
31  }
32
33  // return Time as string in standard-time format (HH:MM:SS AM or PM)
34  string Time::toStandardString() const {
35     ostringstream output;
36     output << ((hour == 0 || hour == 12) ? 12 : hour % 12) << ":"
37        << setfill('0') << setw(2) << minute << ":" << setw(2)
38        << second << (hour < 12 ? " AM" : " PM");
39     return output.str(); // returns the formatted string
40  }
```

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

9.2.5 Scope Resolution Operator

- Each member function's name (lines 12, 26 and 34) is preceded by the class name and the scope resolution operator (`::`)
  - "ties" them to the (now separate) `Time` class definition (Fig. 9.1).
  - `Time::` tells the compiler that each member function is within that class's scope and its name is known to other class members
  - Without `Time::` preceding each function name, these functions would not be recognized by the compiler as `Time` member functions
  - Instead, the compiler would consider them "free" or "loose" functions, like `main`—these are also called global functions
  - Such functions cannot access `Time`'s private data or call the class's member functions, without specifying an object

Common Programming Error 9.1

When defining a class's member functions outside *that class, omitting the class name and scope resolution operator (::) that should precede the function names causes compilation errors.*

9.2.6 Scope Resolution Operator

- To indicate that the member functions in `Time.cpp` are part of class `Time`, include the `Time.h` header (Fig. 9.2, line 7)
  - allows us to use the class name `Time` in the `Time.cpp` file (lines 12, 26 and 34)
- When compiling `Time.cpp`, the compiler uses `Time.h` to ensure
  - the first line of each member function (lines 12, 26 and 34) matches its prototype
  - each member function knows about the class's data members and other member functions

9.2.7 `Time` Class Member Function `setTime` and Throwing Exceptions

- Function `setTime` tests each argument to determine whether the value is in range
- If any of the values is outside its range, `setTime` throws an `invalid_argument` exception (from header `<stdexcept>`)
  - notifies the client code that an invalid argument was received
  - Can use `try`…`catch` to catch exceptions and attempt to recover from them (Fig. 9.3)
- The `throw` statement creates a new object of type `invalid_argument`, immediately terminates `setTime` and returns the exception to the code that attempted to set the time

9.2.8 `Time` Class Member Function `toUniversalString` and String Stream Processing

- `toUniversalString` returns a `string` containing the time formatted as universal time with three colon-separated pairs of digits
  - 1:30:07 PM is returned as "13:30:07".
- `ostringstream` (`<sstream>`) objects provide the same functionality as `cout`, but write their output to `string`
  - `ostringstream`'s `str` member function gets the formatted `string`

9.2.8 `Time` Class Member Function `toUniversalString` and String Stream Processing

- Parameterized stream manipulator `setfill` specifies the fill character that's displayed when an integer is output in a field wider than the number of digits in the value
  - fill characters appear to the left of the digits in the number, because the number is right aligned by default
  - for left-aligned values (specified with the `left` stream manipulator) the fill characters would appear to the right
- If the minute value is 2, it will be displayed as 02, because the fill character is set to zero (`'0'`)
- If the number being output fills the specified field, the fill character will not be displayed

9.2.8 `Time` Class Member Function `toUniversalString` and String Stream Processing

- 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
  - `setfill` is a sticky setting
- This is in contrast to `setw`, which applies only to the next value displayed
  - `setw` is a nonsticky setting
- Line 30 calls `ostringstream`'s `str` member function to get the formatted `string`, which is returned to the client.

Error-Prevention Tip 9.2

Each sticky setting (such as a fill character or 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. Section 13.7.8 discusses how to reset the formatting for an output stream.*

9.2.9 Time Class Member Function toStandardString

- `toStandardString` returns a `string` containing the time formatted as standard time with the hour, minute and second values separated by colons and followed by an AM or PM indicator
  - 10:54:27 AM and 1:27:06 PM
- Line 36 uses the conditional operator (`?:`) to determine the value of hour to be displayed—if the hour is 0 or 12 (AM or PM, respectively), it appears as 12; otherwise, we use the remainder operator (%) to have the hour appear as a value from 1 to 11
- The conditional operator in line 38 determines whether AM or PM will be displayed.
- Line 39 calls `ostringstream`'s `str` member function to return the formatted string.

9.2.10 Implicitly Inlining Member Functions

- 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 9.1

Only the simplest and most stable member functions (i.e., whose implementations are unlikely to change) should be defined in the class header, because every change to the header requires you to recompile every source-code file that's dependent on that header (a time-consuming task in large systems).

9.2.11 Implicitly Inlining Member Functions

- `toUniversalString` and `toStandardString` take no arguments, because these member functions implicitly know that they're to create `string` representations of the data for the particular `Time` object on which they're invoked
  - Can make member-function calls more concise than conventional function calls in procedural programming

Software Engineering Observation 9.2

Using an object-oriented programming approach often requires fewer arguments when calling functions. 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 9.3

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 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.

9.2.12 Using Class Time

- `Time` 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`
- Separating `Time`'s interface from the implementation of its member functions does not affect the way that this client code uses the class

```
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> // invalid_argument exception class
6  #include "Time.h" // definition of class Time from Time.h
7  using namespace std;
8
9  // displays a Time in 24-hour and 12-hour formats
10 void displayTime(const string& message, const Time& time) {
11    cout << message << "\nUniversal time: " << time.toUniversalString()
12       << "\nStandard time: " << time.toStandardString() << "\n\n";
13 }
14
15 int main() {
16    Time t; // instantiate object t of class Time
17
18    displayTime("Initial time:", t); // display t's initial value
```

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

```
19      t.setTime(13, 27, 6); // change time
20      displayTime("After setTime:", t); // display t's new value
21
22      // attempt to set the time with invalid values
23      try {
24         t.setTime(99, 99, 99); // all values out of range
25      }
26      catch (invalid_argument& e) {
27         cout << "Exception: " << e.what() << "\n\n";
28      }
29
30      // display t's value after attempting to set an invalid time
31      displayTime("After attempting to set an invalid time:", t);
32   }
```

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

```
Initial time:
Universal time: 00:00:00
Standard time: 12:00:00 AM

After setTime:
Universal time: 13:27:06
Standard time: 1:27:06 PM

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

After attempting to set an invalid time:
Universal time: 13:27:06
Standard time: 1:27:06 PM
```

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

9.2.12 Using Class Time

- Line 16 creates the `Time` object `t`
- Recall that class `Time` does not define a constructor, so line 16 invokes the compiler-generated default constructor, and `t`'s hour, minute and second are set to `0` via their initializers in class `Time`'s definition
- To illustrate that the `setTime` member function validates its arguments, line 24 calls `setTime` with invalid arguments
  - This statement is placed in a `try` block
  - When an exception occurs, it' s caught at lines 26–28, and line 27 displays the exception's error message by calling its `what` member function
  - Line 31 displays the time again to confirm that `setTime` did not change the time when invalid arguments were supplied

9.2.13 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 the same for all objects of the class and, hence, can be shared among them.*

9.3 Compilation and Linking Process

- The diagram in Fig. 9.4 shows the compilation and linking process that results in an executable `Time` application
- Often a class's interface and implementation will be created and compiled by one programmer and used by a separate programmer who implements the client code that uses the class
- Diagram shows what's required by both the class-implementation programmer and the client-code programmer
- Dashed lines show the pieces required by the class-implementation programmer, the client-code programmer and the `Time` application user, respectively

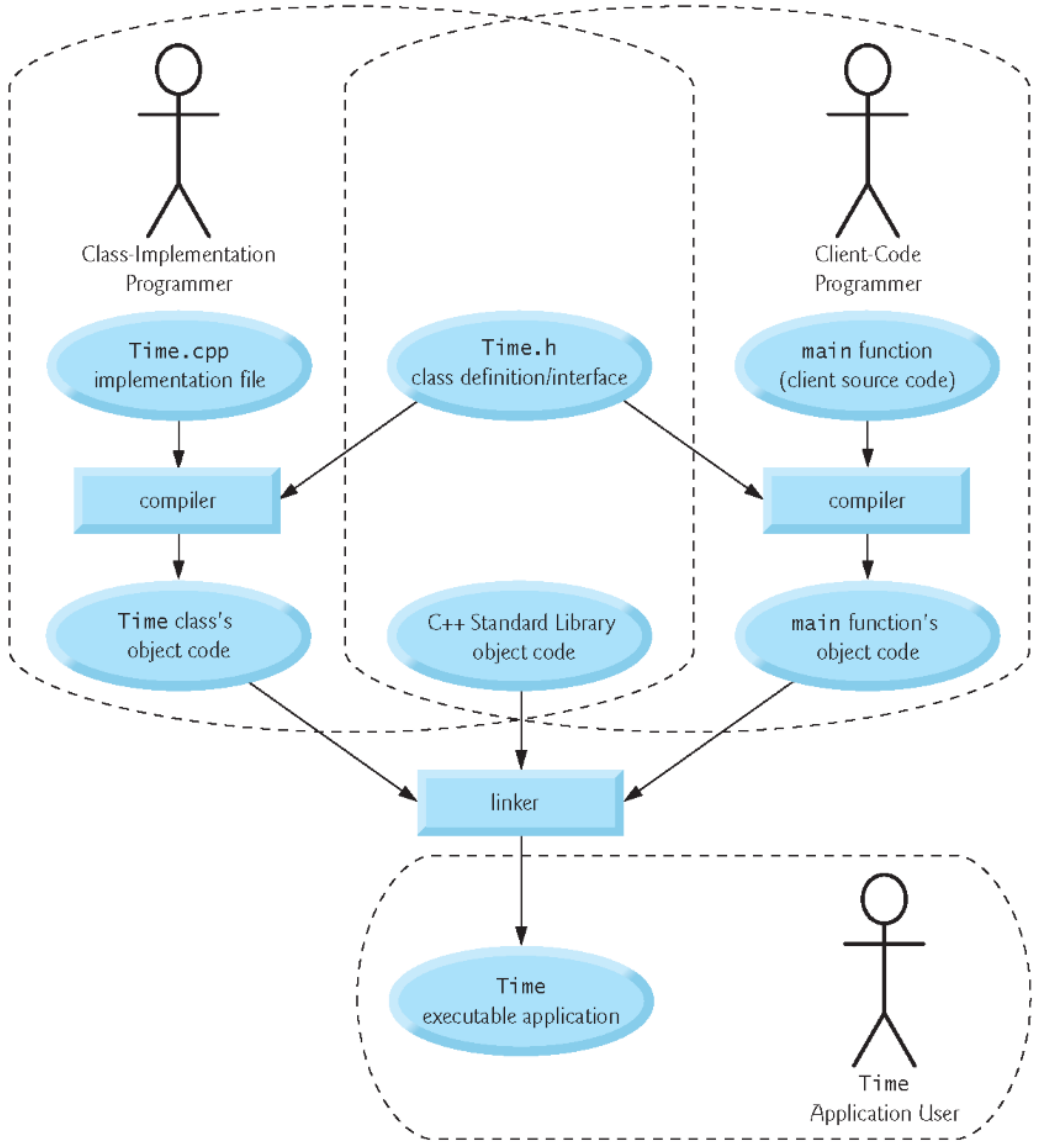

Fig. 9.4 | Compilation and linking process that produces an executable application.

9.3 Compilation and Linking Process

- A class-implementation programmer responsible for creating a reusable `Time` class creates the header `Time.h` and the source-code file `Time.cpp` that `#includes` the header, then compiles the source-code file to create `Time`'s object code
- To hide the class's member-function implementation details, the class-implementation programmer would provide the client-code programmer with the header `Time.h` (which specifies the class's interface and data members) and the `Time` object code (i.e., the machine-code instructions that represent `Time`'s member functions).

9.3 Compilation and Linking Process

- The client-code programmer is not given `Time.cpp`, so the client remains unaware of how `Time`'s member functions are implemented
- The client-code programmer needs to know only `Time`'s interface to use the class and must be able to link its object code
- Since the interface of the class is part of the class definition in the `Time.h` header, the client-code programmer must have access to this file and must `#include` it in the client's source-code file
- When the client code is compiled, the compiler uses the class definition in `Time.h` to ensure that the main function creates and manipulates objects of class `Time` correctly

9.3 Compilation and Linking Process

- To create the executable `Time` application, the last step is to link
  - the object code for the `main` function (i.e., the client code),
  - the object code for class `Time`'s member-function implementations, and
  - the C++ Standard Library object code for the C++ classes (e.g., `string`) used by the class-implementation programmer and the client-code programmer.
- The linker's output is the executable `Time` application that users can execute to create and manipulate a `Time` object
- Compilers and IDEs typically invoke the linker for you after compiling your code

9.3 Compilation and Linking Process

- Compiling Programs Containing Two or More Source-Code Files
  - In Microsoft Visual Studio, add to your project all the headers and source-code files that make up a program, then build and run the project
  - For GNU C++, open a shell and change to the directory containing all the files for a given program, then execute the following command:
    - `g++ -std=c++14 *.cpp -o ExecutableName`
    - *.cpp specifies to compile and link all of the source-code files in the current directory—the preprocessor automatically locates the headers in that directory.
  - For Apple Xcode, add to your project all the headers and source-code files that make up a program, then build and run the project

9.4 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.

9.4 Class Scope and Accessing Class Members (cont.)

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

- As you know, you can use an object's name—or a reference to an object—followed by the dot member-selection operator (`.`) to access an object's members
- To reference an object's members via a pointer to an object, follow the pointer name by the arrow member-selection operator (`->`) and the member name, as in `pointerName->memberName`.

9.4 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;
```

9.4 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 );
```

9.5 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.

9.6 `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.

9.6.1 Constructors with Default Arguments

- Like other functions, constructors can specify *default arguments*.
- Line 13 of Fig. 9.5 declares a `Time` constructor with default arguments, specifying a default value of zero for each argument passed to the constructor
- The constructor is declared `explicit` because it can be called with one argument
- We discuss `explicit` constructors in detail in Section 10.13

```
1  // Fig. 9.5: Time.h
2  // Time class containing a constructor with default arguments.
3  // Member functions defined in Time.cpp.
4  #include <string>
5
6  // prevent multiple inclusions of header
7  #ifndef TIME_H
8  #define TIME_H
9
10 // Time class definition
11 class Time {
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. 9.5 | `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      std::string toUniversalString() const; // 24-hour time format string
27      std::string toStandardString() const; // 12-hour time format string
28  private:
29      unsigned int hour{0}; // 0 - 23 (24-hour clock format)
30      unsigned int minute{0}; // 0 - 59
31      unsigned int second{0}; // 0 - 59
32  };
33
34  #endif
```

Fig. 9.5 | `Time` class containing a constructor with default arguments. (Part 2 of 2.)

Software Engineering Observation 9.4

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

Fig. 9.6 | Member-function definitions for class `Time`. (Part 1 of 4.)

```
22  // set hour value
23  void Time::setHour(int h) {
24     if (h >= 0 && h < 24) {
25        hour = h;
26     }
27     else {
28        throw invalid_argument("hour must be 0-23");
29     }
30  }
31
32  // set minute value
33  void Time::setMinute(int m) {
34     if (m >= 0 && m < 60) {
35        minute = m;
36     }
37     else {
38        throw invalid_argument("minute must be 0-59");
39     }
40  }
41
```

Fig. 9.6 | Member-function definitions for class `Time`. (Part 2 of 4.)

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

Fig. 9.6 | Member-function definitions for class `Time`. (Part 3 of 4.)

```
61  // return Time as a string in universal-time format (HH:MM:SS)
62  string Time::toUniversalString() const {
63     ostringstream output;
64     output << setfill('0') << setw(2) << getHour() << ":"
65        << setw(2) << getMinute() << ":" << setw(2) << getSecond();
66     return output.str();
67  }
68
69  // return Time as string in standard-time format (HH:MM:SS AM or PM)
70  string Time::toStandardString() const {
71     ostringstream output;
72     output << ((getHour() == 0 || getHour() == 12) ? 12 : getHour() % 12)
73        << ":" << setfill('0') << setw(2) << getMinute() << ":" << setw(2)
74        << getSecond() << (hour < 12 ? " AM" : " PM");
75     return output.str();
76  }
```

Fig. 9.6 | Member-function definitions for class `Time`. (Part 4 of 4.)

```
1  // Fig. 9.7: fig09_07.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  // displays a Time in 24-hour and 12-hour formats
9  void displayTime(const string& message, const Time& time) {
10    cout << message << "\nUniversal time: " << time.toUniversalString()
11       << "\nStandard time: " << time.toStandardString() << "\n\n";
12 }
13
```

Fig. 9.7 | Constructor with default arguments. (Part 1 of 3.)

```
14  int main() {
15     Time t1; // all arguments defaulted
16     Time t2{2}; // hour specified; minute and second defaulted
17     Time t3{21, 34}; // hour and minute specified; second defaulted
18     Time t4{12, 25, 42}; // hour, minute and second specified
19
20     cout << "Constructed with:\n\n";
21     displayTime("t1: all arguments defaulted", t1);
22     displayTime("t2: hour specified; minute and second defaulted", t2);
23     displayTime("t3: hour and minute specified; second defaulted", t3);
24     displayTime("t4: hour, minute and second specified", t4);
25
26     // attempt to initialize t5 with invalid values
27     try {
28        Time t5{27, 74, 99}; // all bad values specified
29     }
30     catch (invalid_argument& e) {
31        cerr << "Exception while initializing t5: " << e.what() << endl;
32     }
33  }
```

Fig. 9.7 | Constructor with default arguments. (Part 2 of 3.)

```
Constructed with:

t1: all arguments defaulted
Universal time: 00:00:00
Standard time: 12:00:00 AM

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

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

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

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

Fig. 9.7 | Constructor with default arguments. (Part 3 of 3.)

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

Notes Regarding 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.

Software Engineering Observation 9.5

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

Common Programming Error 9.2

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.*

Software Engineering Observation 9.6

Making data members `private` *and controlling access, especially write access, to those data members through* `public` *member functions helps ensure data integrity.*

Error-Prevention Tip 9.4

The benefits of data integrity are not automatic simply because data members are made `private`*—you must provide appropriate validity checking.*

9.6.2 Overloaded Constructors and C++11 Delegating Constructors

- In Figs. 9.5–9.7, 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 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.

9.6.2 Overloaded Constructors and C++11 Delegating Constructors

- The first three of the four `Time` constructors 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) {}
Time::Time(int hour) : Time(hour, 0, 0) {}
Time::Time(int hour, int minute) : Time(hour, minute, 0)
```

9.7 Destructors

- The name of the destructor 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.

9.8 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
  - Global and `static` objects can alter the order in which destructors are called.

9.8.1 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 programmer-defined cleanup code to be called.

9.8.2 Constructors and Destructors for Non-static Local Objects

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

9.8.3 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`.

9.8.4 Demonstrating When Constructors and Destructors Are Called

- The program of Figs. 9.8–9.10 demonstrates the order in which constructors and destructors are called for objects of class `CreateAndDestroy` (Fig. 9.8 and Fig. 9.9) of various global, local and local `static` objects.

```
1  // Fig. 9.8: CreateAndDestroy.h
2  // CreateAndDestroy class definition.
3  // Member functions defined in CreateAndDestroy.cpp.
4  #include <string>
5
6  #ifndef CREATE_H
7  #define CREATE_H
8
9  class CreateAndDestroy {
10 public:
11    CreateAndDestroy(int, std::string); // constructor
12    ~CreateAndDestroy(); // destructor
13 private:
14    int objectID; // ID number for object
15    sdt::string message; // message describing object
16 };
17
18 #endif
```

Fig. 9.8 | CreateAndDestroy class definition.

```
1  // Fig. 9.9: 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    cout << "Object " << objectID << "   constructor runs   "
11       << message << endl;
12 }
13
14 // destructor
15 CreateAndDestroy::~CreateAndDestroy() {
16    // output newline for certain objects; helps readability
17    cout << (objectID == 1 || objectID == 6 ? "\n" : "");
18
19    cout << "Object " << objectID << "   destructor runs    "
20       << message << endl;
21 }
```

Fig. 9.9 | CreateAndDestroy class member-function definitions.

```
1  // Fig. 9.10: fig09_10.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(); // prototype
9
10 CreateAndDestroy first{1, "(global before main)"}; // global object
11
12 int main() {
13    cout << "\nMAIN FUNCTION: EXECUTION BEGINS" << endl;
14    CreateAndDestroy second{2, "(local in main)"};
15    static CreateAndDestroy third{3, "(local static in main)"};
16
17    create(); // call function to create objects
18
19    cout << "\nMAIN FUNCTION: EXECUTION RESUMES" << endl;
20    CreateAndDestroy fourth{4, "(local in main)"};
21    cout << "\nMAIN FUNCTION: EXECUTION ENDS" << endl;
22 }
```

Fig. 9.10 | Order in which constructors and destructors are called. (Part 1 of 3.)

```
23
24   // function to create objects
25   void create() {
26      cout << "\nCREATE FUNCTION: EXECUTION BEGINS" << endl;
27      CreateAndDestroy fifth{5, "(local in create)"};
28      static CreateAndDestroy sixth{6, "(local static in create)"};
29      CreateAndDestroy seventh{7, "(local in create)"};
30      cout << "\nCREATE FUNCTION: EXECUTION ENDS" << endl;
31   }
```

Fig. 9.10 | 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 in main)
Object 3   constructor runs   (local static in main)

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

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

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

MAIN FUNCTION: EXECUTION ENDS
Object 4   destructor runs    (local in main)
Object 2   destructor runs    (local 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. 9.10 | Order in which constructors and destructors are called. (Part 3 of 3.)

9.9 `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 object's name
  - May be used on the left side of an assignment statement
  - Acceptable *lvalue* that can receive a value
- A member function can return a reference to a `private` data member
  - If the reference return type is declared `const`, the reference is a nonmodifiable *lvalue* and cannot be used to modify the data.
  - If the reference return type is not declared `const`, subtle errors can occur.
- A reference return actually makes a call to member function badSetHour an alias for `private` data member `hour`!
  - The function call can be used in any way that the `private` data member can be used

```
1  // Fig. 9.11: 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 public:
11    void setTime(int, int, int);
12    unsigned int getHour() const;
13    unsigned int& badSetHour(int); // dangerous reference return
14 private:
15    unsigned int hour{0};
16    unsigned int minute{0};
17    unsigned int second{0};
18 };
19
20 #endif
```

Fig. 9.11 | Time class declaration.

```
1  // Fig. 9.12: 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  // set values of hour, minute and second
8  void Time::setTime(int h, int m, int s) {
9     // validate hour, minute and second
10    if ((h >= 0 && h < 24) && (m >= 0 && m < 60) && (s >= 0 && s < 60)) {
11       hour = h;
12       minute = m;
13       second = s;
14    }
15    else {
16       throw invalid_argument(
17          "hour, minute and/or second was out of range");
18    }
19 }
```

Fig. 9.12 | Time class member-function definitions. (Part 1 of 2.)

```
20
21   // return hour value
22   unsigned int Time::getHour() const {return hour;}
23
24   // poor practice: returning a reference to a private data member.
25   unsigned int& Time::badSetHour(int hh) {
26      if (hh >= 0 && hh < 24) {
27         hour = hh;
28      }
29      else {
30         throw invalid_argument("hour must be 0-23");
31      }
32
33      return hour; // dangerous reference return
34   }
```

Fig. 9.12 | `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. 9.13: fig09_13.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     Time t; // create Time object
10
11    // initialize hourRef with the reference returned by badSetHour
12    unsigned int& hourRef{t.badSetHour(20)}; // 20 is a valid hour
13
14    cout << "Valid hour before modification: " << hourRef;
15    hourRef = 30; // use hourRef to set invalid value in Time object t
16    cout << "\nInvalid hour after modification: " << t.getHour();
17
18    // Dangerous: Function call that returns
19    // a reference can be used as an lvalue!
20    t.badSetHour(12) = 74; // assign another invalid value to hour
```

Fig. 9.13 | `public` member function that returns a reference to a `private` data member.

```
21
22       cout << "\n\n**********************************************\n"
23          << "POOR PROGRAMMING PRACTICE!!!!!!!!\n"
24          << "t.badSetHour(12) as an lvalue, invalid hour: "
25          << t.getHour()
26          << "\n**********************************************" << endl;
27    }
```

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

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

Fig. 9.13 | `public` member function that returns a reference to a `private` data member.

9.10 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. 9.14: Date.h
2  // Date class declaration.  Member functions are defined in Date.cpp.
3  #include <string>
4
5  // prevent multiple inclusions of header
6  #ifndef DATE_H
7  #define DATE_H
8
9  // class Date definition
10 class Date {
11 public:
12    explicit Date(unsigned int = 1, unsigned int = 1, unsigned int = 2000);
13    std::string toString() const;
14 private:
15    unsigned int month;
16    unsigned int day;
17    unsigned int year;
18 };
19
20 #endif
```

Fig. 9.14 | Date class declaration.

```
1  // Fig. 9.15: Date.cpp
2  // Date class member-function definitions.
3  #include <sstream>
4  #include <string>
5  #include "Date.h" // include definition of class Date from Date.h
6  using namespace std;
7
8  // Date constructor (should do range checking)
9  Date::Date(unsigned int m, unsigned int d, unsigned int y)
10    : month{m}, day{d}, year{y} {}
11
12 // print Date in the format mm/dd/yyyy
13 string Date::toString() const {
14    ostringstream output;
15    output << month << '/' << day << '/' << year;
16    return output.str();
17 }
```

Fig. 9.15 | Date class member-function definitions.

```
1  // Fig. 9.16: fig09_16.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     Date date1{7, 4, 2004};
10    Date date2; // date2 defaults to 1/1/2000
11
12    cout << "date1 = " << date1.toString()
13       << "\ndate2 = " << date2.toString() << "\n\n";
14
15    date2 = date1; // default memberwise assignment
16
17    cout << "After default memberwise assignment, date2 = "
18       << date2.toString() << endl;
19 }
```

Fig. 9.16 | 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. 9.16 | Class objects can be assigned to each other using default memberwise assignment. (Part 2 of 2.)

9.10 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.

9.11 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.

Error-Prevention Tip 9.5

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.)

- 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.*

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.

9.11 `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.

9.11 `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 `toStandardString`'s prototype and definition so that we can show a compilation error.

```
1  // Fig. 9.17: fig09_17.cpp
2  // const objects and const member functions.
3  #include "Time.h" // include Time class definition
4
5  int main() {
6     Time wakeUp{6, 45, 0}; // non-constant object
7     const Time noon{12, 0, 0}; // constant object
8
9                                 // OBJECT      MEMBER FUNCTION
10    wakeUp.setHour(18);         // non-const   non-const
11    noon.setHour(12);           // const       non-const
12    wakeUp.getHour();           // non-const   const
13    noon.getMinute();           // const       const
14    noon.toUniversalString();   // const       const
15    noon.toStandardString();    // const       non-const
16 }
```

Fig. 9.17 | `const` objects and `const` member functions. (Part 1 of 2.)

Microsoft Visual C++ compiler error messages:

```
C:\examples\ch09\fig09_17\fig09_17.cpp(11): error C2662:
   'void Time::setHour(int)': cannot convert 'this' pointer from 'const Time'
   to 'Time &'
C:\examples\ch09\fig09_17\fig09_17.cpp(11): note: Conversion loses qualifiers
C:\examples\ch09\fig09_17\fig09_17.cpp(15): error C2662:
   'std::string Time::toStandardString(void)': cannot convert 'this' pointer
   from 'const Time' to 'Time &'
C:\examples\ch09\fig09_17\fig09_17.cpp(15): note: Conversion loses qualifiers
```

Fig. 9.17 | `const` objects and `const` member functions. (Part 2 of 2.)

9.12 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 (or aggregation) 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. 9.18–9.19) and `Employee` (Figs. 9.20–9.21) to demonstrate composition.

Software Engineering Observation 9.8

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 are constructed.

```
1  // Fig. 9.18: Date.h
2  // Date class definition; Member functions defined in Date.cpp
3  #include <string>
4
5  #ifndef DATE_H
6  #define DATE_H
7
8  class Date {
9  public:
10    static const unsigned int monthsPerYear{12}; // months in a year
11    explicit Date(unsigned int = 1, unsigned int = 1, unsigned int = 1900);
12    std::string toString() const; // date string in month/day/year format
13    ~Date(); // provided to confirm destruction order
14 private:
15    unsigned int month; // 1-12 (January-December)
16    unsigned int day; // 1-31 based on month
17    unsigned int year; // any year
18
19    // utility function to check if day is proper for month and year
20    unsigned int checkDay(int) const;
21 };
22
23 #endif
```

Fig. 9.18 | Date class definition.

```
1  // Fig. 9.19: Date.cpp
2  // Date class member-function definitions.
3  #include <array>
4  #include <iostream>
5  #include <sstream>
6  #include <stdexcept>
7  #include "Date.h" // include Date class definition
8  using namespace std;
9
10 // constructor confirms proper value for month; calls
11 // utility function checkDay to confirm proper value for day
12 Date::Date(unsigned int mn, unsigned int dy, unsigned int yr)
13    : month{mn}, day{checkDay(dy)}, year{yr} {
14    if (mn < 1 || mn > monthsPerYear) { // validate the month
15       throw invalid_argument("month must be 1-12");
16    }
17
18    // output Date object to show when its constructor is called
19    cout << "Date object constructor for date " << toString() << endl;
20 }
21
```

Fig. 9.19 | Date class member-function definitions. (Part 1 of 3.)

```
22  // print Date object in form month/day/year
23  string Date::toString() const {
24     ostringstream output;
25     output << month << '/' << day << '/' << year;
26     return output.str();
27  }
28
29  // output Date object to show when its destructor is called
30  Date::~Date() {
31     cout << "Date object destructor for date " << toString() << endl;
32  }
33
```

Fig. 9.19 | Date class member-function definitions. (Part 2 of 3.)

```
34  // utility function to confirm proper day value based on
35  // month and year; handles leap years, too
36  unsigned int Date::checkDay(int testDay) const {
37     static const array<int, monthsPerYear + 1> daysPerMonth{
38        0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
39
40     // determine whether testDay is valid for specified month
41     if (testDay > 0 && testDay <= daysPerMonth[month]) {
42        return testDay;
43     }
44
45     // February 29 check for leap year
46     if (month == 2 && testDay == 29 && (year % 400 == 0 ||
47        (year % 4 == 0 && year % 100 != 0))) {
48        return testDay;
49     }
50
51     throw invalid_argument("Invalid day for current month and year");
52  }
```

Fig. 9.19 | Date class member-function definitions. (Part 3 of 3.)

```
1  // Fig. 9.20: Employee.h
2  // Employee class definition showing composition.
3  // Member functions defined in Employee.cpp.
4  #include <string>
5
6  #ifndef EMPLOYEE_H
7  #define EMPLOYEE_H
8
9  #include <string>
10 #include "Date.h" // include Date class definition
11
12 class Employee {
13 public:
14    Employee(const std::string&, const std::string&,
15       const Date&, const Date&);
16    std::string toString() const;
17    ~Employee(); // provided to confirm destruction order
```

Fig. 9.20 | Employee class definition showing composition. (Part 1 of 2.)

```
18   private:
19      std::string firstName; // composition: member object
20      std::string lastName; // composition: member object
21      const Date birthDate; // composition: member object
22      const Date hireDate; // composition: member object
23   };
24
25   #endif
```

Fig. 9.20 | `Employee` class definition showing composition. (Part 2 of 2.)

```
1  // Fig. 9.21: Employee.cpp
2  // Employee class member-function definitions.
3  #include <iostream>
4  #include <sstream>
5  #include "Employee.h" // Employee class definition
6  #include "Date.h" // Date class definition
7  using namespace std;
8
9  // constructor uses member initializer list to pass initializer
10 // values to constructors of member objects
11 Employee::Employee(const string& first, const string& last,
12    const Date &dateOfBirth, const Date &dateOfHire)
13    : firstName{first}, // initialize firstName
14      lastName{last}, // initialize lastName
15      birthDate{dateOfBirth}, // initialize birthDate
16      hireDate{dateOfHire} { // initialize hireDate
17    // output Employee object to show when constructor is called
18    cout << "Employee object constructor: "
19       << firstName << ' ' << lastName << endl;
20 }
```

Fig. 9.21 | Employee class member-function definitions. (Part 1 of 2.)

```
21
22  // print Employee object
23  string Employee::toString() const {
24     ostringstream output;
25     output << lastName << ", " << firstName << "  Hired: "
26        << hireDate.toString() << "  Birthday: " << birthDate.toString();
27     return output.str();
28  }
29
30  // output Employee object to show when its destructor is called
31  Employee::~Employee() {
32     cout << "Employee object destructor: "
33        << lastName << ", " << firstName << endl;
34  }
```

Fig. 9.21 | Employee class member-function definitions. (Part 2 of 2.)

9.12 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 9.2

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

9.12 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.

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

Testing Classes `Date` and `Employee`

- Figure 9.22 creates two `Date` objects (lines 9–10) and passes them as arguments to the constructor of the `Employee` object created in line 11
- There are actually five constructor calls when an `Employee` is constructed:
  - two calls to the `string` class's constructor (lines 13–14 of Fig. 9.21),
  - two calls to the `Date` class's default copy constructor (lines 15–16 of Fig. 9.21),
  - and the call to the `Employee` class's constructor.
- When each `Date` object is created, the `Date` constructor displays a line of output to show that the constructor was called (see the first two lines of the sample output).

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

- [Note: Line 11 of Fig. 9.22 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, 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. 9.22: fig09_22.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     Date birth{7, 24, 1949};
10    Date hire{3, 12, 1988};
11    Employee manager{"Bob", "Blue", birth, hire};
12
13    cout << "\n" << manager.toString() << endl;
14 }
```

Fig. 9.22 | 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
```

Fig. 9.22 | 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.

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.*

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).

Software Engineering Observation 9.9

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 enclosing class's implementation, so member objects of class types should still be* `private`.

9.13 `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.

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

Declaring a friend

- To declare a non-member function as a `friend` of a class, place the function prototype in the class definition and precede it 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`.
- The friend declaration(s) can appear anywhere in a class and are not affected by access specifiers public or private (or protected, which we discuss in Chapter 11).

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

Declaring a friend

- 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.

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

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

- Figure 9.23 defines `friend` function `setX` to set the private data member `x` of class `Count`.
- Function `setX` is a stand-alone (global) function—it isn't a member function of class `Count`.
- 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 8, you'll receive error messages indicating that function `setX` cannot modify class `Count`'s `private` data member `x`.

```
1  // Fig. 9.23: fig09_23.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     friend void setX(Count&, int); // friend declaration
9  public:
10    int getX() const {return x;}
11 private:
12    int x{0};
13 };
14
15 // function setX can modify private data of Count
16 // because setX is declared as a friend of Count (line 8)
17 void setX(Count& c, int val) {
18    c.x = val; // allowed because setX is a friend of Count
19 }
```

Fig. 9.23 | Friends can access private members of a class. (Part 1 of 2.)

```
20
21   int main() {
22      Count counter; // create Count object
23
24      cout << "counter.x after instantiation: " << counter.getX() << endl;
25      setX(counter, 8); // set x using a friend function
26      cout << "counter.x after call to setX friend function: "
27         << counter.getX() << endl;
28   }
```

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

Fig. 9.23 | Friends can access `private` members of a class. (Part 2 of 2.)

9.13 `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 9.10

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

Software Engineering Observation 9.11

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 9.3

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

9.14 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.

9.14 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).

9.14 Using the `this` Pointer (cont.)

- Member functions use the `this` pointer implicitly 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)

Good Programming Practice 9.4

A widely accepted practice to minimize the proliferation of identifier names is to use the same name for a set *function's parameter and the data member it sets, and to reference the data member in the* set *function's body via* `this->`.

9.14 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`
  - In a non-`const` member function of class `Employee`, the this pointer has the type `Employee* const`—a constant pointer to a nonconstant `Employee`.
  - In a `const` member function, this has the type `const Employee* const`—a constant pointer to a constant `Employee`.

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

- Figure 9.24 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.

```
1  // Fig. 9.24: fig09_24.cpp
2  // Using the this pointer to refer to object members.
3  #include <iostream>
4  using namespace std;
5
6  class Test {
7  public:
8     explicit Test(int);
9     void print() const;
10 private:
11    int x{0};
12 };
13
14 // constructor
15 Test::Test(int value) : x{value} {} // initialize x to value
16
```

Fig. 9.24 | Using the `this` pointer to refer to object members. (Part 1 of 2.)

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

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

Fig. 9.24 | Using the `this` pointer to refer to object members. (Part 2 of 2.)

9.14.2 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 sequentially in the same statement
- The program of Figs. 9.25–9.27 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.
- The last statement in the body of each of these member functions returns `*` into a return type of `Time &`.
- The program of Fig. 9.27 creates `Time` object `t`, then uses it in *cascaded member-function calls*

```
1  // Fig. 9.25: Time.h
2  // Time class modified to enable cascaded member-function calls.
3  #include <string>
4
5  // Time class definition.
6  // Member functions defined in Time.cpp.
7  #ifndef TIME_H
8  #define TIME_H
9
10 class Time {
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
```

Fig. 9.25 | Time class modified to enable cascaded member-function calls. (Part 1 of 2.)

```
20      unsigned int getHour() const; // return hour
21      unsigned int getMinute() const; // return minute
22      unsigned int getSecond() const; // return second
23      std::string toUniversalString() const; // 24-hour time format string
24      std::string toStandardString() const; // 12-hour time format string
25   private:
26      unsigned int hour{0}; // 0 - 23 (24-hour clock format)
27      unsigned int minute{0}; // 0 - 59
28      unsigned int second{0}; // 0 - 59
29   };
30
31   #endif
```

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

```
1  // Fig. 9.26: Time.cpp
2  // Time class member-function definitions.
3  #include <iomanip>
4  #include <sstream>
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    setTime(hr, min, sec);
14 }
15
16 // set values of hour, minute, and second
17 Time& Time::setTime(int h, int m, int s) { // note Time& return
18    setHour(h);
19    setMinute(m);
20    setSecond(s);
21    return *this; // enables cascading
22 }
23
```

Fig. 9.26 | `Time` class member-function definitions modified to enable cascaded member-function calls. (Part 1 of 4.)

```
24  // set hour value
25  Time& Time::setHour(int h) { // note Time& return
26     if (h >= 0 && h < 24) {
27        hour = h;
28     }
29     else {
30        throw invalid_argument("hour must be 0-23");
31     }
32
33     return *this; // enables cascading
34  }
35
36  // set minute value
37  Time& Time::setMinute(int m) { // note Time& return
38     if (m >= 0 && m < 60) {
39        minute = m;
40     }
41     else {
42        throw invalid_argument("minute must be 0-59");
43     }
44
45     return *this; // enables cascading
46  }
```

Fig. 9.26 | `Time` class member-function definitions modified to enable cascaded member-function calls. (Part 2 of 4.)

```
47
48  // set second value
49  Time& Time::setSecond(int s) { // note Time& return
50     if (s >= 0 && s < 60) {
51        second = s;
52     }
53     else {
54        throw invalid_argument("second must be 0-59");
55     }
56
57     return *this; // enables cascading
58  }
59
60  // get hour value
61  unsigned int Time::getHour() const {return hour;}
62
63  // get minute value
64  unsigned int Time::getMinute() const {return minute;}
65
66  // get second value
67  unsigned int Time::getSecond() const {return second;}
68
```

Fig. 9.26 | `Time` class member-function definitions modified to enable cascaded member-function calls. (Part 3 of 4.)

```
69  // return Time as a string in universal-time format (HH:MM:SS)
70  string Time::toUniversalString() const {
71     ostringstream output;
72     output << setfill('0') << setw(2) << getHour() << ":"
73        << setw(2) << getMinute() << ":" << setw(2) << getSecond();
74     return output.str();
75
76
77  // return Time as string in standard-time format (HH:MM:SS AM or PM)
78  }
79     ostringstream output;
80     output << ((getHour() == 0 || getHour() == 12) ? 12 : getHour() % 12)
81        << ":" << setfill('0') << setw(2) << getMinute() << ":" << setw(2)
82        << getSecond() << (hour < 12 ? " AM" : " PM");
83     return output.str();
84  }
```

Fig. 9.26 | `Time` class member-function definitions modified to enable cascaded member-function calls. (Part 4 of 4.)

```
1  // Fig. 9.27: fig09_27.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     Time t; // create Time object
9
10    t.setHour(18).setMinute(30).setSecond(22); // cascaded function calls
11
12    // output time in universal and standard formats
13    cout << "Universal time: " << t.toUniversalString()
14       << "\nStandard time: " << t.toStandardString();
15
16    // cascaded function calls
17    cout << "\n\nNew standard time: "
18       << t.setTime(20, 20, 20).toStandardString() << endl;
19 }
```

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

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

Fig. 9.27 | Cascading member-function calls with the `this` pointer.

9.15 `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 "classwide" 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—such as a constant that can be shared by all objects of the class.*

9.15.2 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`.
- In C++11's all `static const` data members can have in-class initializers

9.15.3 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 9.12

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.*

9.15.4 Demonstrating `static` Data Members

- The program of Figs. 9.28–9.30 demonstrates a `private static` data member called `count` and a `public static` member function called `getCount`

```
1  // Fig. 9.28: 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 public:
11    Employee(const std::string&, const std::string&); // constructor
12    ~Employee(); // destructor
13    std::string getFirstName() const; // return first name
14    std::string getLastName() const; // return last name
15
16    // static member function
17    static unsigned int getCount(); // return # of objects instantiated
```

Fig. 9.28 | `Employee` class definition with a `static` data member to track the number of `Employee` objects in memory. (Part 1 of 2.)

```
18   private:
19      std::string firstName;
20      std::string lastName;
21
22      // static data
23      static unsigned int count; // number of objects instantiated
24   };
25
26   #endif
```

Fig. 9.28 | `Employee` class definition with a `static` data member to track the number of `Employee` objects in memory. (Part 2 of 2.)

```
1  // Fig. 9.29: 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() {return count;}
13
14 // constructor initializes non-static data members and
15 // increments static data member count
16 Employee::Employee(const string& first, const string& last)
17    : firstName(first), lastName(last) {
18    ++count; // increment static count of employees
19    cout << "Employee constructor for " << firstName
20       << ' ' << lastName << " called." << endl;
21 }
```

Fig. 9.29 | Employee class member-function definitions. (Part 1 of 2.)

```
22
23  // destructor decrements the count
24  Employee::~Employee() {
25     cout << "~Employee() called for " << firstName
26        << ' ' << lastName << endl;
27     --count; // decrement static count of employees
28  }
29
30  // return first name of employee
31  string Employee::getFirstName() const {return firstName;}
32
33  // return last name of employee
34  string Employee::getLastName() const {return lastName;}
```

Fig. 9.29 | Employee class member-function definitions. (Part 2 of 2.)

```
1  // Fig. 9.30: fig09_30.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     // no objects exist; use class name and binary scope resolution
9     // operator to access static member function getCount
10    cout << "Number of employees before instantiation of any objects is "
11       << Employee::getCount() << endl; // use class name
12
```

Fig. 9.30 | `static` data member tracking the number of objects of a class. (Part 1 of 3.)

```
13      // the following scope creates and destroys
14      // Employee objects before main terminates
15      {
16         Employee e1{"Susan", "Baker"};
17         Employee e2{"Robert", "Jones"};
18
19         // two objects exist; call static member function getCount again
20         // using the class name and the scope resolution operator
21         cout << "Number of employees after objects are instantiated is "
22            << Employee::getCount();
23
24         cout << "\n\nEmployee 1: "
25            << e1.getFirstName() << " " << e1.getLastName()
26            << "\nEmployee 2: "
27            << e2.getFirstName() << " " << e2.getLastName() << "\n\n";
28      }
29
```

Fig. 9.30 | `static` data member tracking the number of objects of a class. (Part 2 of 3.)

```
30      // no objects exist, so call static member function getCount again
31      // using the class name and the scope resolution operator
32      cout << "\nNumber of employees after objects are deleted is "
33         << Employee::getCount() << endl;
34   }
```

```
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. 9.30 | `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.*