6

Classes and Data Abstraction

Objectives

• To understand the software engineering concepts of encapsulation and data hiding.
• To understand the notions of data abstraction and abstract data types (ADTs).
• To be able to create C++ ADTs, namely classes.
• To understand how to create, use, and destroy class objects.
• To be able to control access to object data members and member functions.
• To begin to appreciate the value of object orientation.

*My object all sublime*
*I shall achieve in time.*
W. S. Gilbert

*Is it a world to hide virtues in?*
William Shakespeare, Twelfth Night

*Your public servants serve you right.*
Adlai Stevenson

*Private faces in public places*
*Are wiser and nicer*
*Than public faces in private places.*
W. H. Auden
6.1 Introduction

Now we begin our introduction to object orientation in C++. Why have we deferred object-oriented programming in C++ until Chapter 6? The answer is that the objects we will build will be composed in part of structured program pieces, so we needed to establish a basis in structured programming first.

Through our “Thinking About Objects” sections at the ends of Chapters 1 through 5, we have introduced the basic concepts (i.e., “object think”) and terminology (i.e., “object speak”) of object-oriented programming in C++. In these special sections, we also discussed the techniques of object-oriented design (OOD): We analyzed a typical problem statement that required a system (an elevator simulator) to be built, determined what classes were needed to implement the systems, determined what attributes objects of these classes needed to have, determined what behaviors objects of these classes needed to exhibit, and specified how the objects needed to interact with one another to accomplish the overall goals of the system.

Let us briefly review some key concepts and terminology of object orientation. Object-oriented programming (OOP) encapsulates data (attributes) and functions (behavior) into
packages called classes; the data and functions of a class are intimately tied together. A class is like a blueprint. Out of a blueprint, a builder can build a house. Out of a class, a programmer can create an object. One blueprint can be reused many times to make many houses. One class can be reused many times to make many objects of the same class. Classes have the property of information hiding. This means that although class objects may know how to communicate with one another across well-defined interfaces, classes normally are not allowed to know how other classes are implemented—implementation details are hidden within the classes themselves. Surely it is possible to drive a car effectively without knowing the details of how engines, transmissions and exhaust systems work internally. We will see why information hiding is so crucial to good software engineering.

In C and other procedural programming languages, programming tends to be action-oriented, whereas ideally in C++ programming is object-oriented. In C, the unit of programming is the function. In C++, the unit of programming is the class from which objects are eventually instantiated (i.e., created). C programmers concentrate on writing functions. Groups of actions that perform some task are formed into functions, and functions are grouped to form programs. Data are certainly important in C, but the view is that data exist primarily in support of the actions that functions perform. The verbs in a system specification help the C programmer determine the set of functions that will work together to implement the system.

C++ programmers concentrate on creating their own user-defined types called classes. Classes are also referred to as programmer-defined types. Each class contains data as well as the set of functions that manipulate the data. The data components of a class are called data members. The function components of a class are called member functions (or methods in other object-oriented languages). Just as an instance of a built-in type such as int is called a variable, an instance of a user-defined type (i.e., a class) is called an object. [In the C++ community, the terms variable and object are often used interchangeably.] The focus of attention in C++ is on classes rather than functions. The nouns in a system specification help the C++ programmer determine the set of classes that will be used to create the objects that will work together to implement the system.

Classes in C++ are a natural evolution of the C notion of struct. Before proceeding with the specifics of developing classes in C++, we discuss structures, and we build a user-defined type based on a structure. The weaknesses we expose in this approach will help motivate the notion of a class.

6.2 Structure Definitions

Structures are aggregate data types built using elements of other types including other structs. Consider the following structure definition:

```c
struct Time {
    int hour;     // 0-23
    int minute;   // 0-59
    int second;   // 0-59
};
```

Keyword struct introduces the structure definition. The identifier Time is the structure tag that names the structure definition and is used to declare variables of the structure type. In this example, the new type name is Time. The names declared in the braces of the struc-
ture definition are the structure’s members. Members of the same structure must have unique names, but two different structures may contain members of the same name without conflict. Each structure definition must end with a semicolon. The preceding explanation is valid for classes also as we will soon see; structures and classes are quite similar in C++.

The definition of Time contains three members of type int—hour, minute and second. Structure members can be any type, and one structure can contain members of many different types. A structure cannot, however, contain an instance of itself. For example, a member of type Time cannot be declared in the structure definition for Time. A pointer to another Time structure, however, can be included. A structure containing a member that is a pointer to the same structure type is referred to as a self-referential structure. Self-referential structures are useful for forming linked data structures such as linked lists, queues, stacks and trees, as we will see in Chapter 15.

The preceding structure definition does not reserve any space in memory; rather, the definition creates a new data type that is used to declare variables. Structure variables are declared like variables of other types. The declaration

```c++
Time timeObject, timeArray[10], *timePtr,
    &timeRef = timeObject;
```

declares timeObject to be a variable of type Time, timeArray to be an array with 10 elements of type Time, timePtr to be a pointer to a Time object and timeRef to be a reference to a Time object that is initialized with timeObject.

### 6.3 Accessing Members of Structures

Members of a structure (or of a class) are accessed using the member access operators—the dot operator (.) and the arrow operator (->). The dot operator accesses a structure or class member via the variable name for the object or via a reference to the object. For example, to print member hour of structure timeObject, use the statement

```c++
cout << timeObject.hour;
```

To print member hour of the structure referenced by timeRef, use the statement

```c++
cout << timeRef.hour;
```

The arrow operator—consisting of a minus sign (-) and a greater than sign (>) with no intervening spaces—accesses a structure member or class member via a pointer to the object. Assume that the pointer timePtr has been declared to point to a Time object, and that the address of structure timeObject has been assigned to timePtr. To print member hour of structure timeObject with pointer timePtr, use the statements

```c++
timePtr = &timeObject;
cout << timePtr->hour;
```

The expression timePtr->hour is equivalent to (*timePtr).hour, which dereferences the pointer and accesses the member hour using the dot operator. The parentheses are needed here because the dot operator (.) has a higher precedence than the pointer dereferencing operator (*). The arrow operator and dot operator, along with parentheses and brackets ([ ]), have the second highest operator precedence (after the scope resolution operator introduced in Chapter 3) and associate from left to right.
Common Programming Error 6.1

The expression \((\ast timePtr).hour\) refers to the \(\text{hour}\) member of the \text{struct} pointed to by \text{timePtr}. Omitting the parentheses, as in \(\ast timePtr.hour\) would be a syntax error because \(\ast\) has a higher precedence than \(,\) so the expression would execute as if parenthesized as \(\ast(timePtr.hour)\). This would be a syntax error because with a pointer you must use the arrow operator to refer to a member.

6.4 Implementing a User-Defined Type \textit{Time} with a \textit{struct}

Figure 6.1 creates the user-defined structure type \textit{Time} with three integer members: \textit{hour}, \textit{minute} and \textit{second}. The program defines a single \textit{Time} structure called \textit{dinnerTime} and uses the dot operator to initialize the structure members with the values 18 for \textit{hour}, 30 for \textit{minute} and 0 for \textit{second}. The program then prints the time in military format (also called “universal format”) and standard format. Note that the print functions receive references to constant \textit{Time} structures. This causes \textit{Time} structures to be passed to the print functions by reference—thus eliminating the copying overhead associated with passing structures to functions by value—and the use of \textit{const} prevents the \textit{Time} structure from being modified by the print functions. In Chapter 7, we discuss \textit{const} objects and \textit{const} member functions.

```cpp
// Fig. 6.1: fig06_01.cpp
// Create a structure, set its members, and print it.
#include <iostream>
using std::cout;
using std::endl;

struct Time {      // structure definition
   int hour;     // 0-23
   int minute;   // 0-59
   int second;   // 0-59
};

void printMilitary( const Time & );  // prototype
void printStandard( const Time & );  // prototype

int main()
{
   Time dinnerTime;    // variable of new type Time
   // set members to valid values
   dinnerTime.hour = 18;
   dinnerTime.minute = 30;
   dinnerTime.second = 0;
   cout << "Dinner will be held at ";
   printMilitary( dinnerTime );
}
```

The code snippet creates a structure `Time` with three integer members: `hour`, `minute`, and `second`. It defines a variable `dinnerTime` of type `Time` and initializes its members to `18` for `hour`, `30` for `minute`, and `0` for `second`. It then prints the time in military format and standard format using print functions `printMilitary` and `printStandard`, respectively. The use of `const` ensures that the `Time` structure is not modified by the print functions.
Performance Tip 6.1
By default, structures are passed call-by-value. To avoid the overhead of copying a structure, pass the structure call-by-reference.

Software Engineering Observation 6.1
To avoid the overhead of call-by-value yet still gain the benefit that the caller’s original data are protected from modification, pass large-size arguments as const references.

There are drawbacks to creating new data types with structures in this manner. Since initialization is not specifically required, it is possible to have uninitialized data and the consequent problems. Even if the data are initialized, they may not be initialized correctly.
Invalid values can be assigned to the members of a structure (as we did in Fig. 6.1) because the program has direct access to the data. In lines 33 and 34, the program was easily able to assign bad values to the \texttt{hour} and \texttt{minute} members of the \texttt{Time} object \texttt{dinner-Time}. If the implementation of the \texttt{struct} is changed (e.g., the time could be represented as the number of seconds since midnight), all programs that use the \texttt{struct} must be changed. This is because the programmer directly manipulates the data representation. There is no “interface” to it to ensure that the programmer uses the data type’s services correctly and to ensure that the data remain in a consistent state.

\textbf{Software Engineering Observation 6.2}

It is important to write programs that are understandable and easy to maintain. Change is the rule rather than the exception. Programmers should anticipate that their code will be modified. As we will see, classes can facilitate program modifiability.

There are other problems associated with C-style structures. In C, structures cannot be printed as a unit; rather, their members must be printed and formatted one at a time. A function could be written to print the members of a structure in some appropriate format. Chapter 8, “Operator Overloading,” illustrates how to overload the \texttt{<<} operator to enable objects of a structure type or class type to be printed easily. In C, structures may not be compared in their entirety; they must be compared member by member. Chapter 8 also illustrates how to overload equality operators and relational operators to compare objects of (C++) structure and class types.

The following section reimplements our \texttt{Time} structure as a C++ class and demonstrates some of the advantages to creating so-called \textit{abstract data types} as classes. We will see that classes and structures can be used almost identically in C++. The difference between the two is in the default accessibility associated with the members of each. This will be explained shortly.

\section{6.5 Implementing a Time Abstract Data Type with a class}

Classes enable the programmer to model objects that have \textit{attributes} (represented as \textit{data members}) and \textit{behaviors} or \textit{operations} (represented as \textit{member functions}). Types containing data members and member functions are defined in C++ using the keyword \texttt{class}.

Member functions are sometimes called \textit{methods} in other object-oriented programming languages, and are invoked in response to \textit{messages} sent to an object. A message corresponds to a member-function call sent from one object to another or sent from a function to an object.

Once a class has been defined, the class name can be used to declare objects of that class. Figure 6.2 contains a simple definition for class \texttt{Time}.

Our \texttt{Time} class definition begins with the keyword \texttt{class}. The \textit{body} of the class definition is delineated with left and right braces (\{ and \}). The class definition terminates with a semicolon. Our \texttt{Time} class definition and our \texttt{Time} structure definition each contain the three integer members \texttt{hour}, \texttt{minute} and \texttt{second}.

\textbf{Common Programming Error 6.2}

\textit{Forgetting the semicolon at the end of a class (or structure) definition is a syntax error.}
The remaining parts of the class definition are new. The public: and private: labels are called member access specifiers. Any data member or member function declared after member access specifier public (and before the next member access specifier) is accessible wherever the program has access to an object of class Time. Any data member or member function declared after member access specifier private (and up to the next member access specifier) is accessible only to member functions of the class. Member access specifiers are always followed by a colon (:) and can appear multiple times and in any order in a class definition. For the remainder of the text, we will refer to the member access specifiers as public and private (without the colon). In Chapter 9 we introduce a third member access specifier, protected, as we study inheritance and the part it plays in object-oriented programming.

**Good Programming Practice 6.1**

Use each member access specifier only once in a class definition for clarity and readability. Place public members first where they are easy to locate.

The class definition contains prototypes for the following four member functions after the public member access specifier—Time, setTime, printMilitary and printStandard. These are the public member functions or public services or public behaviors or interface of the class. These functions will be used by clients (i.e., portions of a program that are users) of the class to manipulate the data of the class. The data members of the class support the delivery of the services the class provides to the clients of the class with its member functions. These services allow the client code to interact with an object of the class.

Notice the member function with the same name as the class; it is called a constructor function of that class. A constructor is a special member function that initializes the data members of a class object. A class constructor function is called automatically when an object of that class is created. We will see that it is common to have several constructors for a class; this is accomplished through function overloading. Note that no return type is specified for the constructor.

**Common Programming Error 6.3**

Specifying a return type and/or a return value for a constructor is a syntax error.

Fig. 6.2 Simple definition of class Time.
The three integer members appear after the **private** member access specifier. This indicates that these data members of the class are only accessible to member functions—and, as we will see in the next chapter, “friends”—of the class. Thus, the data members can only be accessed by the four functions whose prototypes appear in the class definition (or by friends of the class). Data members are normally listed in the **private** portion of a class and member functions are normally listed in the **public** portion. It is possible to have **private** member functions and **public** data, as we will see later; the latter is uncommon and is considered a poor programming practice.

Once the class has been defined, it can be used as a type in object, array and pointer definitions as follows:

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

The class name becomes a new type specifier. There may be many objects of a class, just as there may be many variables of a type such as `int`. The programmer can create new class types as needed. This is one reason why C++ is said to be an **extensible language**.

Figure 6.3 uses the **Time** class. The program instantiates a single object of class **Time** called `t`. When the object is instantiated, the **Time** constructor is called automatically and initializes each **private** data member to 0. The time is then printed in military and standard formats to confirm that the members have been initialized properly. The time is then set using the **setTime** member function and is printed again in both formats. Then **setTime** attempts to set the data members to invalid values, and the time is again printed in both formats.

```cpp
// Fig. 6.3: fig06_03.cpp
// Time class.
#include <iostream>
using std::cout;
using std::endl;

// Time abstract data type (ADT) definition
class Time {
 public:
  Time();                          // constructor
  void setTime( int, int, int );  // set hour, minute, second
  void printMilitary();           // print military time format
  void printStandard();           // print standard time format

 private:
  int hour;     // 0 - 23
  int minute;   // 0 - 59
  int second;   // 0 - 59
};
```

**Fig. 6.3** Abstract data type **Time** implementation as a class (part 1 of 3).
// Time constructor initializes each data member to zero.
// Ensures all Time objects start in a consistent state.
Time::Time() { hour = minute = second = 0; }

// Set a new Time value using military time. Perform validity
// checks on the data values. Set invalid values to zero.
void Time::setTime( int h, int m, int s )
{
    hour = ( h >= 0 && h < 24 ) ? h : 0;
    minute = ( m >= 0 && m < 60 ) ? m : 0;
    second = ( s >= 0 && s < 60 ) ? s : 0;
}

// Print Time in military format
void Time::printMilitary()
{
    cout << ( hour < 10 ? "0" : "" ) << hour << ":
    << ( minute < 10 ? "0" : "" ) << minute;
}

// Print Time in standard format
void Time::printStandard()
{
    cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
    << ":
    << ( minute < 10 ? "0" : "" ) << minute
    << ":
    << ( second < 10 ? "0" : "" ) << second
    << ( hour < 12 ? " AM" : " PM");
}

// Driver to test simple class Time
int main()
{
    Time t;  // instantiate object t of class Time
    cout << "The initial military time is ";
    t.printMilitary();
    cout << "\nThe initial standard time is ";
    t.printStandard();
    t.setTime( 13, 27, 6 );
    cout << "\nMilitary time after setTime is ";
    t.printMilitary();
    cout << "\nStandard time after setTime is ";
    t.printStandard();
    t.setTime( 99, 99, 99 );  // attempt invalid settings
    cout << "\nAfter attempting invalid settings:"
    << "\nMilitary time: ";
    t.printMilitary();
    cout << "\nStandard time: ";
    t.printStandard();
    cout << endl;
}

Fig. 6.3  Abstract data type Time implementation as a class (part 2 of 3).
Again, note that the data members `hour`, `minute` and `second` are preceded by the `private` member access specifier. A class’ `private` data members are normally not accessible outside the class. (Again, we will see in Chapter 7 that friends of a class may access the class’ `private` members.) The philosophy here is that the actual data representation used within the class is of no concern to the class’ clients. For example, it would be perfectly reasonable for the class to represent the time internally as the number of seconds since midnight. Clients could use the same `public` member functions and get the same results without being aware of this. In this sense, the implementation of a class is said to be hidden from its clients. Such information hiding promotes program modifiability and simplifies the client’s perception of a class.

**Software Engineering Observation 6.3**

Clients of a class use the class without knowing the internal details of how the class is implemented. If the class implementation is changed (to improve performance, for example), provided the class’ interface remains constant, the class’ client source code need not change (although the client may need to be recompiled). This makes it much easier to modify systems.

In this program, the `Time` constructor initializes the data members `hour`, `minute` and `second` to 0 (i.e., the military time equivalent of 12 AM). This ensures that the object is in a consistent state when it is created. Invalid values cannot be stored in the data members of a `Time` object because the constructor is automatically called when the `Time` object is created and all subsequent attempts by a client to modify the data members are scrutinized by function `setTime`.

**Software Engineering Observation 6.4**

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 and/or by member functions that store new data. Because the data are already in the object, the member function calls often have no arguments or at least have fewer arguments than typical function calls in non-object-oriented languages. Thus, the calls are shorter, the function definitions are shorter and the function prototypes are shorter.

Note that the data members of a class cannot be initialized where they are declared in the class body. These data members should be initialized by the class’ constructor, or they can be assigned values by “set” functions.
Common Programming Error 6.4

Attempting to initialize a data member of a class explicitly in the class definition is a syntax error.

A function with the same name as the class but preceded with a tilde character (~) is called the destructor of that class (this example does not explicitly include a destructor, so the system “plugs one in” for you). The destructor does “termination housekeeping” on each class object before the memory for the object is reclaimed by the system. Destructors cannot take arguments and hence cannot be overloaded. We will discuss constructors and destructors in more detail later in this chapter and in Chapter 7.

Note that the functions the class provides to the outside world are preceded by the public member access specifier. The public functions implement the behaviors or services the class provides to its clients—commonly referred to as the class’ interface or public interface.

Software Engineering Observation 6.5

Clients have access to a class’ interface but should not have access to a class’ implementation.

The class definition contains declarations of the class’ data members and the class’ member functions. The member function declarations are the function prototypes we discussed in earlier chapters. Member functions can be defined inside a class, but it is a good programming practice to define the functions outside the class definition.

Software Engineering Observation 6.6

Declaring member functions inside a class definition (via their function prototypes) and defining those member functions outside that class definition separates the interface of a class from its implementation. This promotes good software engineering. Clients of a class cannot see the implementation of that class’ member functions and need not recompile if that implementation changes.

Note the use of the binary scope resolution operator (::) in each member function definition following the class definition in Fig. 6.3. Once a class is defined and its member functions are declared, the member functions must be defined. Each member function of the class can be defined directly in the class body (rather than including the function prototype of the class), or the member function can be defined after the class body. When a member function is defined after its corresponding class definition, the function name is preceded by the class name and the binary scope resolution operator (::). Because different classes can have the same member names, the scope resolution operator “ties” the member name to the class name to uniquely identify the member functions of a particular class.

Common Programming Error 6.5

When defining a class’ member functions outside that class, omitting the class name and scope resolution operator on the function name is an error.

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’ scope, i.e., its name is known only to other members of the class unless referred to via an object of the class, a reference to an object of the class or a pointer to an object of the class. We will say more about class scope shortly.
If a member function is defined in a class definition, the member function is automatically inlined. Member functions defined outside a class definition may be made inline by explicitly using the keyword `inline`. Remember that the compiler reserves the right not to inline any function.

**Performance Tip 6.2**

Defining a small member function inside the class definition automatically inlines the member function (if the compiler chooses to do so). This can improve performance, but it does not promote the best software engineering because clients of the class will be able to see the implementation of the function and their code must be recompiled if the inline function definition changes.

**Software Engineering Observation 6.7**

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

It is interesting that the `printMilitary` and `printStandard` member functions take no arguments. This is because member functions implicitly know that they are to print the data members of the particular `Time` object for which they are invoked. This makes member function calls more concise than conventional function calls in procedural programming.

**Testing and Debugging Tip 6.1**

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 and/or the wrong number of arguments.

**Software Engineering Observation 6.8**

Using an object-oriented programming approach can often simplify function calls by reducing the number of parameters to be passed. This benefit of object-oriented programming derives from the fact that encapsulation of data members and member functions within an object gives the member functions the right to access the data members.

Classes simplify programming because the client (or user of the class object) need only be concerned with the operations encapsulated or embedded in the object. Such operations are usually designed to be client-oriented rather than implementation-oriented. Clients need not be concerned with a class’ implementation (although the client, of course, wants a correct and efficient implementation). Interfaces do change, but less frequently than implementations. When an implementation changes, implementation-dependent code must change accordingly. By hiding the implementation we eliminate the possibility of other program parts becoming dependent on the details of the class implementation.

**Software Engineering Observation 6.9**

A central theme of this book is “reuse, reuse, reuse.” We will carefully discuss a number of techniques for “polishing” classes to encourage reuse. We focus on “crafting valuable classes” and creating valuable “software assets.”

Often, classes do not have to be created “from scratch.” Rather, they may be derived from other classes that provide attributes and behaviors the new classes can use. Or classes can include objects of other classes as members. Such software reuse can greatly enhance
programmer productivity. Deriving new classes from existing classes is called inheritance and is discussed in detail in Chapter 9. Including class objects as members of other classes is called composition (or aggregation) and is discussed in Chapter 7.

People new to object-oriented programming often express concern at the fact that objects must be quite large because they contain data and functions. Logically, this is true—the programmer may think of objects as containing data and functions. Physically, however, this is not true.

**Performance Tip 6.3**

Actually, objects contain only data, so objects are much smaller than if they also contained functions. Applying operator `sizeof` to a class name or to an object of that class will report only the size of the class’ data. 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 of the member functions. Each object, of course, needs its own copy of the class’ data because these data can vary among the objects. The function code is nonmodifiable (also called reentrant code or pure procedure) and hence can be shared among all objects of one class.

### 6.6 Class Scope and Accessing Class Members

A class’ data members (variables declared in the class definition) and member functions (functions declared in the class definition) belong to that class’ scope. Nonmember functions are defined as file scope.

Within a class’ scope, class members are immediately accessible by all of that class’ member functions and can be referenced by name. Outside a class’ scope, 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. [We will see in Chapter 7 that an implicit handle is inserted by the compiler on every reference to a data member or member function in an object.]

Member functions of a class can be overloaded, but only by other member functions of the class. To overload a member function, simply provide in the class definition a prototype for each version of the overloaded function, and provide a separate function definition for each version of the function.

Variables defined in a member function have function scope—they are known only to that function. If a member function defines a variable with the same name as a variable with class scope, the class-scope variable is hidden by the function-scope variable in the function scope. Such a hidden variable can be accessed by preceding the operator with the class name followed by the scope resolution operator (`::`). Hidden global variables can be accessed with the unary scope resolution operator (see Chapter 3).

The operators used to access class members are identical to the operators used to access structure members. The **dot member selection operator** (`.`) is combined with an object’s name or with a reference to an object to access the object’s members. The **arrow member selection operator** (`->`) is combined with a pointer to an object to access that object’s members.

Figure 6.4 uses a simple class called Count with **public** data member `x` of type int, and **public** member function `print` to illustrate accessing the members of a class with the member selection operators. The program creates (defines) three variables related to type Count—one counter, `counterRef` (a reference to a Count object) and coun-
terPtr (a pointer to a Count object). Variable counterRef is defined to reference counter, and variable counterPtr is defined to point to counter. It is important to note that data member x has been made public here simply to demonstrate how public members are accessed off handles (i.e., a name, a reference or a pointer). As we have stated, data are typically made private, as we will do in most subsequent examples. In Chapter 9, "Inheritance," we will sometimes make data protected.

```
// Fig. 6.4: fig06_04.cpp
// Demonstrating the class member access operators . and ->

#include <iostream>

using std::cout;
using std::endl;

// Simple class Count
class Count {
public:
    int x;
    void print() { cout << x << endl; }
};

int main()
{
    Count counter,                // create counter object
    *counterPtr = &counter,      // pointer to counter
    &counterRef = counter;       // reference to counter

    cout << "Assign 7 to x and print using the object's name: ";
    counter.x = 7;                // assign 7 to data member x
    counter.print();              // call member function print

    cout << "Assign 8 to x and print using a reference: ";
    counterRef.x = 8;             // assign 8 to data member x
    counterRef.print();           // call member function print

    cout << "Assign 10 to x and print using a pointer: ";
    counterPtr->x = 10;           // assign 10 to data member x
    counterPtr->print();          // call member function print
    return 0;
}
```

Assign 7 to x and print using the object's name: 7
Assign 8 to x and print using a reference: 8
Assign 10 to x and print using a pointer: 10
### 6.7 Separating Interface from Implementation

One of the fundamental principles of good software engineering is to separate interface from implementation. This makes it easier to modify programs. As far as clients of a class are concerned, changes in the class’ implementation do not affect the client as long as the class’ interface originally provided to the client is unchanged (the class’ functionality could be expanded beyond the original interface).

**Software Engineering Observation 6.10**

*Place the class declaration in a header file to be included by any client that wants to use the class. This forms the class’ public interface (and provides the client with the function prototypes it needs to be able to call the class’ member functions). Place the definitions of the class member functions in a source file. This forms the implementation of the class.*

**Software Engineering Observation 6.11**

*Clients of a class do not need access to the class’ source code in order to use the class. The clients do, however, need to be able to link to the class’ object code. This encourages independent software vendors (ISVs) to provide class libraries for sale or license. The ISVs provide in their products only the header files and the object modules. No proprietary information is revealed—as would be the case if source code were provided. The C++ user community benefits by having more ISV-produced class libraries available.*

Actually, things are not quite this rosy. Header files do contain some portion of the implementation and hints about other portions of the implementation. Inline member functions, for example, need to be in a header file, so that when the compiler compiles a client, the client can include the inline function definition in place. Private members are listed in the class definition in the header file, so these members are visible to clients even though the clients may not access the private members. In Chapter 7, we show how to use a so-called proxy class to hide even the private data of a class from clients of the class.

**Software Engineering Observation 6.12**

*Information important to the interface to a class should be included in the header file. Information that will be used only internally in the class and will not be needed by clients of the class should be included in the unpublished source file. This is yet another example of the principle of least privilege.*

Figure 6.5 splits the program of Fig. 6.3 into multiple files. When building a C++ program, each class definition is normally placed in a header file, and that class’ member function definitions are placed in source-code files of the same base name. The header files are included (via `#include`) in each file in which the class is used, and the source-code file is compiled and linked with the file containing the main program. See your compiler’s documentation to determine how to compile and link programs consisting of multiple source files.

Figure 6.5 consists of the header file `time1.h` in which class `Time` is declared, the file `time1.cpp` in which the member functions of class `Time` are defined and the file `fig06_05.cpp` in which function `main` is defined. The output for this program is identical to the output of Fig. 6.3.
// Fig. 6.5: time1.h
// Declaration of the Time class.
// Member functions are defined in time1.cpp

// prevent multiple inclusions of header file
#ifndef TIME1_H
#define TIME1_H

// Time abstract data type definition
class Time {

public:
  Time();                        // constructor
  void setTime( int, int, int ); // set hour, minute, second
  void printMilitary();          // print military time format
  void printStandard();          // print standard time format

private:
  int hour;     // 0 - 23
  int minute;   // 0 - 59
  int second;   // 0 - 59
};

#endif

Fig. 6.5  Separating Time class interface and implementation—time1.h.

// Fig. 6.5: time1.cpp
// Member function definitions for Time class.
#include <iostream>
using std::cout;

#include "time1.h"

// Time constructor initializes each data member to zero.
// Ensures all Time objects start in a consistent state.
Time::Time() { hour = minute = second = 0; }

// Set a new Time value using military time. Perform validity
// checks on the data values. Set invalid values to zero.
void Time::setTime( int h, int m, int s )
{
  hour   = ( h >= 0 && h < 24 ) ? h : 0;
  minute = ( m >= 0 && m < 60 ) ? m : 0;
  second = ( s >= 0 && s < 60 ) ? s : 0;
}

// Print Time in military format
void Time::printMilitary()
{
  cout << ( hour < 10 ? "0" : "" ) << hour << ":" <<
  minute << ":" << second;
}

Fig. 6.5  Separating Time class interface and implementation—time1.cpp (part 1 of 2).
48       << ( minute < 10 ? "0" : "" ) << minute;
49     }
50
51     // Print time in standard format
52     void Time::printStandard()
53     {
54         cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
55             << ":" << ( minute < 10 ? "0" : "" ) << minute
56             << ":" << ( second < 10 ? "0" : "" ) << second
57             << ( hour < 12 ? " AM" : " PM" );
58     }

---

59     // Fig. 6.5: fig06_05.cpp
60     // Driver for Time1 class
61     // NOTE: Compile with time1.cpp
62     #include <iostream>
63     using std::cout;
64     using std::endl;
65     #include "time1.h"
66
67     // Driver to test simple class Time
68     int main()
69     {
70         Time t;  // instantiate object t of class Time
71
72         cout << "The initial military time is ";
73         t.printMilitary();
74         cout << "The initial standard time is ";
75         t.printStandard();
76
77         t.setTime( 13, 27, 6 );
78         cout << "Military time after setTime is ";
79         t.printMilitary();
80         cout << "Standard time after setTime is ";
81         t.printStandard();
82
83         t.setTime( 99, 99, 99 );  // attempt invalid settings
84         cout << "After attempting invalid settings:
85             Military time: ";
86         t.printMilitary();
87         cout << "Standard time: ";
88         t.printStandard();
89         cout << endl;
90     return 0;
91     }
Chapter 6  Classes and Data Abstraction  407

Note that the class declaration is enclosed in the following preprocessor code:

```c
// prevent multiple inclusions of header file
#ifndef TIME1_H
#define TIME1_H
...
#endif
```

When we build larger programs, other definitions and declarations will also be placed in header files. The preceding preprocessor directives prevent the code between `#ifndef` (if not defined) and `#endif` from being included if the name `TIME1_H` has been defined. If the header has not been included previously in a file, the name `TIME1_H` is defined by the `#define` directive and the header file statements are included. If the header has been included previously, `TIME1_H` is defined already and the header file is not included again. Attempts to include a header file multiple times (inadvertently) typically occur in large programs with many header files that may themselves include other header files. Note: The convention we use for the symbolic constant name in the preprocessor directives is simply the header file name with the underscore character replacing the period.

**Testing and Debugging Tip 6.2**

Use `#ifndef`, `#define` and `#endif` preprocessor directives to prevent header files from being included more than once in a program.

**Good Programming Practice 6.2**

Use the name of the header file with the period replaced by an underscore in the `#ifndef` and `#define` preprocessor directives of a header file.

---

6.8 Controlling Access to Members

The member access specifiers `public` and `private` (and `protected`, as we will see in Chapter 9, “Inheritance”) are used to control access to a class’ data members and member functions. The default access mode for classes is `private` so all members after the class header and before the first member access specifier are `private`. After each member access specifier, the mode that was invoked by that member access specifier applies until the next member access specifier or until the terminating right brace (`{`) of the class definition.

The member access specifiers `public`, `private` and `protected` may be repeated, but such usage is rare and can be confusing.
A class’ \texttt{private} members can be accessed only by member functions (and \texttt{friends}, as we will see in Chapter 7) of that class. The \texttt{public} members of a class may be accessed by any function in the program.

The primary purpose of \texttt{public} members is to present to the class’ clients a view of the \textit{services} (behaviors) the class provides. This set of services forms the \texttt{public} \textit{interface} of the class. Clients of the class need not be concerned with how the class accomplishes its tasks. The \texttt{private} members of a class as well as the definitions of its \texttt{public} member functions are not accessible to the clients of a class. These components form the \textit{implementation} of the class.

\textbf{Software Engineering Observation 6.13}

C++ encourages programs to be implementation independent. When the implementation of a class used by implementation-independent code changes, that code need not be modified. If any part of the interface of the class changes, the implementation-independent code must be recompiled.

\textbf{Common Programming Error 6.6}

An attempt by a function, which is not a member of a particular class (or a \texttt{friend} of that class), to access a \texttt{private} member of that class is a syntax error.

Figure 6.6 demonstrates that \texttt{private} class members are only accessible through the \texttt{public} class interface using \texttt{public} member functions. When this program is compiled, the compiler generates two errors stating that the \texttt{private} member specified in each statement is not accessible. Figure 6.6 includes \texttt{timel.h} and is compiled with \texttt{timel.cpp} from Fig. 6.5.

\textbf{Good Programming Practice 6.3}

If you choose to list the \texttt{private} members first in a class definition, explicitly use the \texttt{private} member access specifier despite the fact that \texttt{private} is assumed by default. This improves program clarity. Our preference is to list the \texttt{public} members of a class first to emphasize the class’ interface.

```cpp
// Fig. 6.6: fig06_06.cpp
// Demonstrate errors resulting from attempts
// to access private class members.

#include <iostream>
using std::cout;
#include "timel.h"

int main()
{
  Time t;
  // Error: 'Time::hour' is not accessible
  t.hour = 7;
  // Error: 'Time::minute' is not accessible
  cout << "minute = " << t.minute;
}
```

\textbf{Fig. 6.6} Erroneous attempt to access \texttt{private} members of a class (part 1 of 2).
Despite the fact that the `public` and `private` member access specifiers may be repeated and intermixed, list all the `public` members of a class first in one group and then list all the `private` members in another group. This focuses the client’s attention on the class’ `public` interface, rather than on the class’ implementation.

Software Engineering Observation 6.14

Keep all the data members of a class `private`. Provide `public` member functions to set the values of `private` data members and to get the values of `private` data members. This architecture helps hide the implementation of a class from its clients, which reduces bugs and improves program modifiability.

A client of a class may be a member function of another class or it may be a global function (i.e., a C-like “loose” or “free” function in the file that is not a member function of any class).

The default access for members of a class is `private`. Access to members of a class may be explicitly set to `public`, `protected` (as we will see in Chapter 9) or `private`. The default access for `struct` members is `public`. Access to members of a `struct` also may be explicitly set to `public`, `protected` or `private`.

```cpp
19  return 0;
20 }

Borland C++ command-line compiler error messages

Fig06_06.cpp:
Error E2247 Fig06_06.cpp 15:
  'Time::hour' is not accessible in function main()
Error E2247 Fig06_06.cpp 18:
  'Time::minute' is not accessible in function main()

*** 2 errors in Compile ***

Microsoft Visual C++ compiler error messages

Compiling...
Fig06_06.cpp
D:\Fig06_06.cpp(15) : error C2248: 'hour' : cannot access private
member declared in class 'Time'
D:\Fig06_06\timel.h(18) : see declaration of 'hour'
D:\Fig06_06.cpp(18) : error C2248: 'minute' : cannot access private
member declared in class 'Time'
D:\\timel.h(19) : see declaration of 'minute'
Error executing cl.exe.

test.exe - 2 error(s), 0 warning(s)

Fig. 6.6  Erroneous attempt to access `private` members of a class (part 2 of 2).
Classes and Data Abstraction

Software Engineering Observation 6.15
Class designers use private, protected and public members to enforce the notion of information hiding and the principle of least privilege.

Just because class data is private does not necessarily mean that clients cannot effect changes to that data. The data can be changed by member functions or friends of that class. As we will see, these functions should be designed to ensure the integrity of the data.

Access to a class’ private data should be carefully controlled by the use of member functions, called access functions (also called accessor methods). For example, to allow clients to read the value of private data, the class can provide a get function. To enable clients to modify private data, the class can provide a set function. Such modification would seem to violate the notion of private data. But a set member function can provide data validation capabilities (such as range checking) to ensure that the value is set properly. A set function can also translate between the form of the data used in the interface and the form used in the implementation. A get function need not expose the data in “raw” format; rather, the get function can edit the data and limit the view of the data the client will see.

Software Engineering Observation 6.16
The class designer need not provide set and/or get functions for each private data item; these capabilities should be provided only when appropriate. If the service is useful to the client code, that service should be provided in the class’ public interface.

Testing and Debugging Tip 6.3
Making the data members of a class private and the member functions of the class public facilitates debugging because problems with data manipulations are localized to either the class’ member functions or the friends of the class.

6.9 Access Functions and Utility Functions

Not all member functions need be made public to serve as part of the interface of a class. Some member functions remain private and serve as utility functions to the other functions of the class.

Software Engineering Observation 6.17
Member functions tend to fall into a number of different categories: functions that read and return the value of private data members; functions that set the value of private data members; functions that implement the services of the class; and functions that perform various mechanical chores for the class such as initializing class objects, assigning class objects, converting between classes and built-in types or between classes and other classes and handling memory for class objects.

Access functions can read or display data. Another common use for access functions is to test the truth or falsity of conditions—such functions are often called predicate functions. An example of a predicate function would be an isEmpty function for any container class—a class capable of holding many objects—such as a linked list, a stack or a queue. A program would test isEmpty before attempting to read another item from the container object. An isFull predicate function might test a container class object to determine if it has no additional room. A set of useful predicate functions for our Time class might be isAM and isPM.
Figure 6.7 demonstrates the notion of a utility function (also called a helper function). A utility function is not part of a class’ interface; rather, it is a private member function that supports the operation of the class’ public member functions. Utility functions are not intended to be used by clients of a class.

```cpp
// Fig. 6.7: salesp.h
// SalesPerson class definition
// Member functions defined in salesp.cpp
#ifndef SALESP_H
#define SALESP_H

class SalesPerson {
public:
    SalesPerson();                // constructor
    void getSalesFromUser(); // get sales figures from keyboard
    void setSales( int, double ); // User supplies one month’s
                                // sales figures.
    void printAnnualSales();

private:
    double totalAnnualSales();    // utility function
    double sales[ 12 ];           // 12 monthly sales figures
};

#endif
```

**Fig. 6.7** Using a utility function—`salesp.h`.

```cpp
// Fig. 6.7: salesp.cpp
// Member functions for class SalesPerson
#include <iostream>

using std::cout;
using std::cin;
using std::endl;

// Constructor function initializes array
SalesPerson::SalesPerson()
{
    for ( int i = 0; i < 12; i++ )
        sales[ i ] = 0.0;
}
```

**Fig. 6.7** Using a utility function—`salesp.cpp` (part 1 of 2)
Class `SalesPerson` has an array of 12 monthly sales figures initialized by the constructor to zero and set to user-supplied values by function `setSales`. Public member function `printAnnualSales` prints the total sales for the last 12 months. Utility function `totalAnnualSales` totals the 12 monthly sales figures for the benefit of `printAnnualSales`. Member function `printAnnualSales` edits the sales figures into dollar amount format.
Chapter 6 Classes and Data Abstraction

Note that `main` includes only a simple sequence of member function calls—there are no control structures.

Software Engineering Observation 6.18

A phenomenon of object-oriented programming is that once a class is defined, creating and manipulating objects of that class usually involves issuing only a simple sequence of member function calls—few, if any, control structures are needed. By contrast, it is common to have control structures in the implementation of a class’ member functions.

6.10 Initializing Class Objects: Constructors

When a class object is created, its members can be initialized by that class’ constructor function. A constructor is a class member function with the same name as the class. The programmer provides the constructor, which is then invoked automatically each time an object of that class is created (instantiated). Constructors may be overloaded to provide a variety of means for initializing objects of a class. Data members must either be initialized in a constructor of the class or their values may be set later after the object is created. However, it is considered a good programming and software engineering practice to ensure that an object is fully initialized before the client code invokes the object’s member functions. In general, you should not rely on the client code to ensure that an object gets initialized properly.
Common Programming Error 6.7
Data members of a class cannot be initialized in the class definition.

Common Programming Error 6.8
Attempting to declare a return type for a constructor and/or attempting to return a value from a constructor are syntax errors.

Good Programming Practice 6.5
When appropriate (almost always), provide a constructor to ensure that every object is properly initialized with meaningful values. Pointer data members, in particular, should be initialized to some legitimate pointer value or to 0.

Testing and Debugging Tip 6.4
Every member function (and friend) that modifies the private data members of an object should ensure that the data remains in a consistent state.

When an object of a class is declared, initializers can be provided in parentheses to the right of the object name and before the semicolon. These initializers are passed as arguments to the class’ constructor. We will soon see several examples of these constructor calls. [Note: Although programmers do not explicitly call constructors, programmers can still provide data that get passed to constructors as arguments.]

6.11 Using Default Arguments with Constructors
The constructor from timel.cpp (Fig. 6.5) initialized hour, minute and second to 0 (i.e., 12 midnight in military time). Constructors can contain default arguments. Figure 6.8 redefines the Time constructor function to include default arguments of zero for each variable. By providing default arguments to the constructor, even if no values are provided in a constructor call, the object is still guaranteed to be initialized to a consistent state, due to the default arguments. A programmer-supplied constructor that defaults all its arguments (or explicitly requires no arguments) is also a default constructor, i.e., a constructor that can be invoked with no arguments. There can be only one default constructor per class.

In this program, the constructor calls member function setTime with the values passed to the constructor (or the default values) to ensure that the value supplied for hour is in the range 0 to 23, and that the values for minute and second are each in the range 0 to 59. If a value is out of range, it is set to zero by setTime (this is an example of ensuring that a data member remains in a consistent state).

Note that the Time constructor could be written to include the same statements as member function setTime. This may be slightly more efficient because the extra call to setTime is eliminated. However, coding the Time constructor and member function setTime identically makes maintenance of this program more difficult. If the implementation of member function setTime changes, the implementation of the Time constructor should change accordingly. Having the Time constructor call setTime directly requires any changes to the implementation of setTime to be made only once. This reduces the likelihood of a programming error when altering the implementation. Also, the performance of the Time constructor can be enhanced by explicitly declaring the constructor inline or by defining the constructor in the class definition (which implicitly inlines the function definition).
// Fig. 6.8: time2.h
// Declaration of the Time class.
// Member functions are defined in time2.cpp
// preprocessor directives that
// prevent multiple inclusions of header file
#ifndef TIME2_H
#define TIME2_H

// Time abstract data type definition
class Time {

public:
    Time( int = 0, int = 0, int = 0 ); // default constructor
    void setTime( int, int, int ); // set hour, minute, second
    void printMilitary();          // print military time format
    void printStandard();          // print standard time format

private:
    int hour;     // 0 - 23
    int minute;   // 0 - 59
    int second;   // 0 - 59
};
#endif

Fig. 6.8 Using a constructor with default arguments—time2.h.

// Fig. 6.8: time2.cpp
// Member function definitions for Time class.
#include <iostream>

using std::cout;

#include "time2.h"

// Time constructor initializes each data member to zero.
// Ensures all Time objects start in a consistent state.
Time::Time( int hr, int min, int sec )
{ setTime( hr, min, sec ); }

// Set a new Time value using military time. Perform validity
// checks on the data values. Set invalid values to zero.
void Time::setTime( int h, int m, int s )
{
    hour   = ( h >= 0 && h < 24 ) ? h : 0;
    minute = ( m >= 0 && m < 60 ) ? m : 0;
    second = ( s >= 0 && s < 60 ) ? s : 0;
}

Fig. 6.8 Using a constructor with default arguments—time2.cpp (part 1 of 2).
// Print Time in military format
void Time::printMilitary()
{
    cout << (hour < 10 ? "0" : "") << hour << ":" << (minute < 10 ? "0" : "") << minute;
}

// Print Time in standard format
void Time::printStandard()
{
    cout << (hour == 0 || hour == 12 ? 12 : hour % 12) << ":" << (minute < 10 ? "0" : "") << minute 
    << "":" << (second < 10 ? "0" : "") << second 
    << (hour < 12 ? " AM" : " PM");
}

int main()
{
    Time t1, // all arguments defaulted
t2(2), // minute and second defaulted
t3(21, 34), // second defaulted
t4(12, 25, 42), // all values specified
t5(27, 74, 99); // all bad values specified
    cout << "Constructed with:\n" << " all arguments defaulted:\n  ";
t1.printMilitary();
cout << "\n";
t1.printStandard();
cout << "\nhour specified; minute and second defaulted:" "\n";
t2.printMilitary();
cout << "\n";
t2.printStandard();
cout << "\nhour and minute specified; second defaulted:" "\n";
t3.printMilitary();

Fig. 6.8 Using a constructor with default arguments—time2.cpp (part 2 of 2).

Fig. 6.8 Using a constructor with default arguments—fig06_08.cpp (part 1 of 2).
Software Engineering Observation 6.19

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.

Good Programming Practice 6.6

Declare default function argument values only in the function prototype within the class definition in the header file.

Common Programming Error 6.9

Specifying default initializers for the same member function in both a header file and in the member function definition.
Note: Any change to the default arguments of a method requires the client code to be recompiled. If it is likely that the default argument values will change, use overloaded functions instead. Thus, if the implementation of a member function changes, the client code need not be recompiled.

Figure 6.8 initializes five `Time` objects—one with all three arguments defaulted in the constructor call, one with one argument specified, one with two arguments specified, one with three arguments specified and one with three invalid arguments specified. The contents of each object’s data members after instantiation and initialization are displayed.

If no constructor is defined for a class, the compiler creates a default constructor. Such a constructor does not perform any initialization, so when the object is created, it is not guaranteed to be in a consistent state.

Software Engineering Observation 6.20
It is possible for a class not to have a default constructor if any constructors are defined and none of them is explicitly a default constructor.

6.12 Using Destructors

A destructor is a special member function of a class. The name of the destructor for a class is the tilde (\~) character followed by the class name. This naming convention has intuitive appeal, because as we will see in a later chapter, the tilde operator is the bitwise complement operator, and, in a sense, the destructor is the complement of the constructor.

A class’ destructor is called when an object is destroyed—e.g., for automatic objects when program execution leaves the scope in which an object of that class was instantiated. The destructor itself does not actually destroy the object—it performs termination housekeeping before the system reclaims the object’s memory so that memory may be reused to hold new objects.

A destructor receives no parameters and returns no value. A class may have only one destructor—destructor overloading is not allowed.

Common Programming Error 6.10
It is a syntax error to attempt to pass arguments to a destructor, to specify a return type for a destructor (even `void` cannot be specified), to return values from a destructor or to overload a destructor.

Notice that destructors have not been provided for the classes presented so far. We will soon see several examples of classes with useful destructors. In Chapter 8, we will see that destructors are appropriate for classes whose objects contain dynamically allocated memory (for arrays and strings, for example). In Chapter 7, we discuss how to dynamically allocate and deallocate storage.

Software Engineering Observation 6.21
As we will see (throughout the remainder of the book), constructors and destructors have much greater prominence in C++ and object-oriented programming than is possible to convey after only our brief introduction here.

6.13 When Constructors and Destructors Are Called

Constructors and destructors are called automatically. The order in which these function calls are made depends on the order in which execution enters and leaves the scope in which
objects are instantiated. Generally, destructor calls are made in the reverse order of the constructor calls. However, as we will see in Fig. 6.9, the storage class of objects can alter the order in which the destructors are called.

Constructors are called for objects defined in global scope before any other function (including `main`) in that file begins execution (although the order of execution of global object constructors between files is not guaranteed). The corresponding destructors are called when `main` terminates or the `exit` function (see Chapter 18, “Other Topics,” for more information on the function) is called. Destructors are not called for global objects if the program is terminated with a call to function `abort` (see Chapter 18, “Other Topics,” for more information on this function).

Constructors are called for automatic local objects when execution reaches the point where the objects are defined. Corresponding destructors are called when the objects leave scope (i.e., the block in which they are defined exits). Constructors and destructors for automatic objects are called each time the objects enter and leave scope. Destructors are not called for automatic objects if the program is terminated with a call to functions `exit` or `abort`.

Constructors are called for `static` local objects only once when execution first reaches the point where the objects are defined. Corresponding destructors are called when `main` terminates or the `exit` function is called. Destructors are not called for `static` objects if the program is terminated with a call to function `abort`.

The program of Fig. 6.9 demonstrates the order in which constructors and destructors are called for objects of type `CreateAndDestroy` in several scopes. The program defines `first` in global scope. Its constructor is called as the program begins execution and its destructor is called at program termination after all other objects are destroyed.

Function `main` declares three objects. Objects `second` and `fourth` are local automatic objects, and object `third` is a `static` local object. The constructors for each of these objects are called when execution reaches the point where each object is declared. The destructors for objects `fourth` and `second` are called in that order when the end of `main` is reached. Because object `third` is `static`, it exists until program termination. The destructor for object `third` is called before the destructor for `first`, but after all other objects are destroyed.

```cpp
// Fig. 6.9: create.h
// Definition of class CreateAndDestroy.
// Member functions defined in create.cpp.
#ifndef CREATE_H
#define CREATE_H

class CreateAndDestroy {
public:
    CreateAndDestroy( int ); // constructor
    ~CreateAndDestroy();     // destructor
private:
    int data;
};
#endif
```

Fig. 6.9 Demonstrating the order in which constructors and destructors are called—create.h.
Fig. 6.9: create.cpp

```cpp
#include <iostream>

using std::cout;
using std::endl;

#include "create.h"

CreateAndDestroy::CreateAndDestroy( int value )
{
    data = value;
    cout << "Object " << data << " constructor";
}

CreateAndDestroy::~CreateAndDestroy()
{
    cout << "Object " << data << " destructor " << endl;
}
```

Fig. 6.9: fig06_09.cpp

```cpp
#include <iostream>

using std::cout;
using std::endl;

#include "create.h"

void create( void ); // prototype

CreateAndDestroy first( 1 ); // global object

int main()
{
    cout << " (global created before main)" << endl;
    CreateAndDestroy second( 2 ); // local object
    cout << " (local automatic in main)" << endl;
    static CreateAndDestroy third( 3 ); // local object
    cout << " (local static in main)" << endl;
    create(); // call function to create objects
    CreateAndDestroy fourth( 4 ); // local object
    cout << " (local automatic in main)" << endl;
    return 0;
}
```

Fig. 6.9: Demonstrating the order in which constructors and destructors are called—create.cpp (part 1 of 2).
```
// Function to create objects
void create( void )
{
    CreateAndDestroy fifth( 5 );
    cout << "   (local automatic in create)" << endl;
    static CreateAndDestroy sixth( 6 );
    cout << "   (local static in create)" << endl;
    CreateAndDestroy seventh( 7 );
    cout << "   (local automatic in create)" << endl;
}
```

Object 1   constructor   (global created before main)
Object 2   constructor   (local automatic in main)
Object 3   constructor   (local static in main)
Object 5   constructor   (local automatic in create)
Object 6   constructor   (local static in create)
Object 7   constructor   (local automatic in create)
Object 7   destructor
Object 5   destructor
Object 4   constructor   (local automatic in main)
Object 4   destructor
Object 2   destructor
Object 6   destructor
Object 3   destructor
Object 1   destructor

Fig. 6.9  Demonstrating the order in which constructors and destructors are called—fig06_09.cpp (part 2 of 2).

Function `create` declares three objects—`fifth` and `seventh` are local automatic objects, and `sixth` is a `static` local object. The destructors for objects `seventh` and `fifth` are called in that order when the end of `create` is reached. Because `sixth` is `static`, it exists until program termination. The destructor for `sixth` is called before the destructors for `third` and `first`, but after all other objects are destroyed.

### 6.14 Using Data Members and Member Functions

A class’ `private` data members can be accessed only by member functions (and `friends`) of the class. A typical manipulation might be the adjustment of a customer’s bank balance (e.g., a `private` data member of a class `BankAccount`) by a member function `computeInterest`.

Classes often provide `public` member functions to allow clients of the class to `set` (i.e., write) or `get` (i.e., read) the values of `private` data members. These functions need not be called `set` and `get` specifically, but they often are. More specifically, a member function that `sets` data member `interestRate` would typically be named `setInterestRate`, and a member function that `gets` the `interestRate` would typically be called `getInterestRate`. Get functions are also commonly called “query” functions.
It may seem that providing both set and get capabilities is essentially the same as making the data members public. This is yet another subtlety of C++ that makes the language so desirable for software engineering. If a data member is public, then the data member may be read or written at will by any function in the program. If a data member is private, a public get function would certainly seem to allow other functions to read the data at will, but the get function could control the formatting and display of the data. A public set function could—and most likely would—carefully scrutinize any attempt to modify the value of the data member. This would ensure that the new value is appropriate for that data item. For example, an attempt to set the day of the month to 37 could be rejected, an attempt to set a person’s weight to a negative value could be rejected, an attempt to set a numeric quantity to an alphabetic value could be rejected, an attempt to set a grade on an exam to 185 (when the proper range is zero to 100) could be rejected, etc.

**Software Engineering Observation 6.22**

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

**Testing and Debugging Tip 6.5**

The benefits of data integrity are not automatic simply because data members are made private—the programmer must provide the validity checking. C++ does, however, provide a framework in which programmers can design better programs in a convenient manner.

**Good Programming Practice 6.7**

Member functions that set the values of private data should verify that the intended new values are proper; if they are not, the set functions should place the private data members into an appropriate consistent state.

The client of a class should be notified when an attempt is made to assign an invalid value to a data member. A class’ set functions are often written to return values indicating that an attempt was made to assign invalid data to an object of the class. This enables clients of the class to test the return values of set functions to determine if the object they are manipulating is a valid object and to take appropriate action if the object is not valid.

Figure 6.10 extends our Time class to include get and set functions for the hour, minute and second private data members. The set functions strictly control the setting of the data members. Attempts to set any data member to an incorrect value cause the data member to be set to zero (thus leaving the data member in a consistent state). Each get function simply returns the appropriate data member’s value. The program first uses the set functions to set the private data members of Time object t to valid values, then uses the get functions to retrieve the values for output. Next the set functions attempt to set the hour and second members to invalid values and the minute member to a valid value, and then the get functions retrieve the values for output. The output confirms that invalid values cause the data members to be set to zero. Finally, the program sets the time to 11:58:00 and increments the minute value by 3 with a call to function incrementMinutes. Function incrementMinutes is a nonmember function that uses the get and set member functions to increment the minute member properly. Although this works, it incurs the performance burden of issuing multiple function calls. In the next chapter, we discuss the notion of friend functions as a means of eliminating this performance burden.
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 in a consistent state. Using data members before they have been properly initialized can cause logic errors.

```cpp
// Fig. 6.10: time3.h
// Declaration of the Time class.
// Member functions defined in time3.cpp

#ifndef TIME3_H
#define TIME3_H

class Time {

public:
    Time( int = 0, int = 0, int = 0 );  // constructor

    // set functions
    void setTime( int, int, int );  // set hour, minute, second
    void setHour( int );   // set hour
    void setMinute( int ); // set minute
    void setSecond( int ); // set second

    // get functions
    int getHour();         // return hour
    int getMinute();       // return minute
    int getSecond();       // return second

    void printMilitary();  // output military time
    void printStandard();  // output standard time

private:
    int hour;              // 0 - 23
    int minute;            // 0 - 59
    int second;            // 0 - 59
};
#endif
```

```cpp
// Fig. 6.10: time3.cpp
// Member function definitions for Time class.
#include <iostream>

using std::cout;
```

Fig. 6.10 Using set and get functions—time3.h.

Fig. 6.10 Using set and get functions—time3.cpp (part 1 of 2).
```cpp
#include "time3.h"

// Constructor function to initialize private data.
// Calls member function setTime to set variables.
// Default values are 0 (see class definition).
Time::Time( int hr, int min, int sec )
    { setTime( hr, min, sec ); }

// Set the values of hour, minute, and second.
void Time::setTime( int h, int m, int s )
{
    setHour( h );
    setMinute( m );
    setSecond( s );
}

// Set the hour value
void Time::setHour( int h )
    { hour = ( h >= 0 && h < 24 ) ? h : 0; }

// Set the minute value
void Time::setMinute( int m )
    { minute = ( m >= 0 && m < 60 ) ? m : 0; }

// Set the second value
void Time::setSecond( int s )
    { second = ( s >= 0 && s < 60 ) ? s : 0; }

// Get the hour value
int Time::getHour() { return hour; }

// Get the minute value
int Time::getMinute() { return minute; }

// Get the second value
int Time::getSecond() { return second; }

// Print time is military format
void Time::printMilitary()
{
    cout << ( hour < 10 ? "0": "") << hour << ":";
    << ( minute < 10 ? "0": "") << minute;
}

// Print time in standard format
void Time::printStandard()
{
    cout << ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
    << ":" << ( minute < 10 ? "0": "") << minute
    << ":" << ( second < 10 ? "0": "") << second
    << ( hour < 12 ? " AM" : " PM" );
}
```

Fig. 6.10  Using set and get functions—*time3.cpp* (part 2 of 2).
// Fig. 6.10: fig06_10.cpp
// Demonstrating the Time class set and get functions

#include <iostream>

using std::cout;
using std::endl;

#include "time3.h"

void incrementMinutes( Time &tt, const int count )
{
    cout << "Incrementing minute " << count
    << " times:\nStart time: ";
    tt.printStandard();
    for ( int i = 0; i < count; i++ )
    {
        tt.setMinute( ( tt.getMinute() + 1 ) % 60 );
        if ( tt.getMinute() == 0 )
            tt.setHour( ( tt.getHour() + 1 ) % 24 );
        cout << "\nminute + 1: ";
    }
}

void incrementMinutes( Time &tt, const int );

int main()
{
    Time t;
    t.setHour( 17 );
    t.setMinute( 34 );
    t.setSecond( 25 );

    cout << "Result of setting all valid values:\n" << " Hour: " << t.getHour()
    << " Minute: " << t.getMinute()
    << " Second: " << t.getSecond();

    t.setHour( 234 ); // invalid hour set to 0
    t.setMinute( 43 );
    t.setSecond( 6373 ); // invalid second set to 0

    cout << "\n\nResult of attempting to set invalid hour and" << " second:\n" << " Hour: " << t.getHour()
    << " Minute: " << t.getMinute()
    << " Second: " << t.getSecond() << "\n";

    t.setTime( 11, 58, 0 );
    incrementMinutes( t, 3 );

    return 0;
}
Using `set` functions is certainly important from a software engineering standpoint because they can perform validity checking. Both `set` and `get` functions have another important software engineering advantage.

**Software Engineering Observation 6.23**

Accessing `private` data through `set` and `get` member functions not only protects the data members from receiving invalid values, but it also insulates clients of the class from the representation of the data members. Thus, if the representation of the data changes for some reason (typically to reduce the amount of storage required or to improve performance), only the member functions need to change—the clients need not change as long as the interface provided by the member functions remains the same. The clients may, however, need to be recompiled.

### 6.15 A Subtle Trap: Returning a Reference 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. One way to use this capability (unfortunately!) is to have a `public` member function of a class return a non-`const` reference to a `private` data member of that class.

Figure 6.11 uses a simplified `Time` class to demonstrate returning a reference to a `private` data member. Such a return actually makes a call to function `badSetHour` an alias for the `private` data member `hour`! 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!

**Good Programming Practice 6.8**

Never have a `public` member function return a non-`const` reference (or a pointer) to a `private` data member. Returning such a reference violates the encapsulation of the class. In fact, returning any reference or pointer to `private` data still makes the client code dependent on the representation of the class’ data. So, returning pointers or references to `private` data should be avoided.
// Fig. 6.11: time4.h
// Declaration of the Time class.
// Member functions defined in time4.cpp

// preprocessor directives that
// prevent multiple inclusions of header file
#ifndef TIME4_H
#define TIME4_H

class Time {

public:
    Time( int = 0, int = 0, int = 0 );
    void setTime( int, int, int );
    int getHour();
    int &badSetHour( int );  // DANGEROUS reference return

private:
    int hour;
    int minute;
    int second;
};
#endif

// Fig. 6.11 Returning a reference to a private data member—time4.h.

// Fig. 6.11: time4.cpp
// Member function definitions for Time class.
#include "time4.h"

// Constructor function to initialize private data.
// Calls member function setTime to set variables.
// Default values are 0 (see class definition).
Time::Time( int hr, int min, int sec )
{
    setTime( hr, min, sec );
}

// Set the values of hour, minute, and second.
void Time::setTime( int h, int m, int s )
{
    hour   = ( h >= 0 && h < 24 ) ? h : 0;
    minute = ( m >= 0 && m < 60 ) ? m : 0;
    second = ( s >= 0 && s < 60 ) ? s : 0;
}

// Get the hour value
int Time::getHour() { return hour; }

// POOR PROGRAMMING PRACTICE:
// Returning a reference to a private data member.
int &Time::badSetHour( int hh )
{
    hour = ( hh >= 0 && hh < 24 ) ? hh : 0;
}

// Fig. 6.11 Returning a reference to a private data member—time4.cpp (part 1 of 2).
The program begins by declaring `Time` object `t` and reference `hourRef` that is assigned the reference returned by the call `t.badSetHour(20)`. The program displays the value of the alias `hourRef`. Next, the alias is used to set the value of `hour` to 30 (an
invalid value) and the value is displayed again. Finally, the function call itself is used as an

`lvalue` and assigned the value 74 (another invalid value), and the value is displayed.

### 6.16 Assignment by Default Memberwise Copy

The assignment operator (\(=\)) can be used to assign an object to another object of the same
type. Such assignment is by default performed by memberwise copy—each member of one
object is copied (assigned) individually to the same member in another object (see Fig.
6.12). (Note: Memberwise copy can cause serious problems when used with a class whose
data members contain dynamically allocated storage; in Chapter 8, “Operator Overload-
ing,” we will discuss these problems and show how to deal with them.)

Objects may be passed as function arguments and may be returned from functions.
Such passing and returning is performed call-by-value by default—a copy of the object is
passed or returned (we present several examples in Chapter 8, “Operator Overloading”).

**Performance Tip 6.4**

Passing an object call-by-value is good from a security standpoint because the called func-
tion has no access to the original object, but call-by-value can degrade performance when
making a copy of a large object. An object can be passed call-by-reference by passing either
a pointer or a reference to the object. Call-by-reference offers good performance but is weaker
from a security standpoint because the called function is given access to the original ob-
ject. Call-by-\texttt{const}-reference is a safe, good-performing alternative.

```cpp
1 // Fig. 6.12: fig06_12.cpp
2 // Demonstrating that class objects can be assigned
3 // to each other using default memberwise copy
4 #include <iostream>
5
6 using std::cout;
7 using std::endl;
8
9 // Simple Date class
10 class Date {
11    public:
12        Date( int = 1, int = 1, int = 1990 ); // default constructor
13        void print();
14    private:
15        int month;
16        int day;
17        int year;
18    }
19
20 // Simple Date constructor with no range checking
21 Date::Date( int m, int d, int y )
22 {
23    month = m;
24    day = d;
25    year = y;
26 }
```

Fig. 6.12 Assigning one object to another with default memberwise copy (part 1 of 2).
6.17 Software Reusability

People who write object-oriented programs concentrate on implementing useful classes. There is a tremendous opportunity to capture and catalog classes so that they can be accessed by large segments of the programming community. Many class libraries exist and others are being developed worldwide. There are efforts to make these libraries broadly accessible. Software is increasingly being constructed from existing, well-defined, carefully tested, well-documented, portable, widely available components. This kind of software reusability speeds the development of powerful, high-quality software. Rapid applications development (RAD) through the mechanisms of reusable componentry has become an important field.

Significant problems must be solved, however, before the full potential of software reusability can be realized. We need cataloging schemes, licensing schemes, protection mechanisms to ensure that master copies of classes are not corrupted, description schemes so that designers of new systems can determine if existing objects meet their needs, browsing mechanisms to determine what classes are available and how closely they meet software developer requirement and the like. Many interesting research and development problems need to be solved. There is great motivation to solve these problems because the potential value of their solutions is enormous.
6.18 (Optional Case Study) Thinking About Objects: Starting to Program the Classes for the Elevator Simulator

In the “Thinking About Objects” sections in Chapters 1 through 5, we introduced the fundamentals of object orientation and developed an object-oriented design for an elevator simulator. In the body of Chapter 6, we introduced the details of programming with C++ classes. We now begin implementing our object-oriented design in C++. In this section, we will use our UML class diagram to outline the C++ header files that define our classes.

**Implementation: Visibility**

In the body of Chapter 6, we introduced the access specifiers `public` and `private`. Before we create the class header files, we must first consider which elements from our class diagram should be `public` and which elements should be `private`.

**Software Engineering Observation 6.24**

Each element of a class should have `private` visibility until it can be proven that the element needs `public` visibility.

In Chapter 6, we discussed how data members should generally be `private`, but what about member functions? The operations of a class are its member functions. These operations must be invoked by clients of that class; therefore, the member functions should be `public`. In the UML, `public` visibility is indicated by placing a plus sign (+) before a particular element (i.e., a member function or a data member); a minus sign (-) indicates `private` visibility. Figure 6.13 shows our updated class diagram with visibility notations included. (Note that we have added the `personArrives` operation to class `Floor` from our sequence diagram in Fig. 4.27.) As we write the C++ header files for the classes in our system, we automatically place the items designated with “+” into the `public` sections and items designated with “-” into the `private` sections of the class declarations.

**Implementation: Handles**

In order for an object of class A to communicate with an object of class B, the class A object must have a handle to the class B object. This means that either the class A object must know the name of the class B object, or the class A object must hold a reference (Section 3.17) or a pointer (Chapter 5) to the class B object. Figure 5.36 contained a list of collaborations among objects in our system. The classes in the left column of the table need a handle to each of the classes in the right column of the table to send messages to those classes. Figure 6.14 lists the handles for each class based on the information displayed in the table from Figure 5.36.

In the body of Chapter 6, we discussed how to implement handles in C++ as references and pointers to classes (and, again, we will prefer references over pointers, where appropriate). These references then become attributes (data) of the class. Until we discuss composition in Chapter 7, we cannot represent every item from Fig. 6.14 in our class header files. We will discuss these special cases shortly.

---

1. In situations where the name of the class B object is not available to the class A object, we will prefer references over pointers (where appropriate), because references are inherently safer than pointers.
### Class Handles

<table>
<thead>
<tr>
<th>Class</th>
<th>Handles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elevator</td>
<td>ElevatorButton, Bell, Floor, Door</td>
</tr>
<tr>
<td>Clock</td>
<td></td>
</tr>
<tr>
<td>Scheduler</td>
<td>Person, Floor</td>
</tr>
<tr>
<td>Person</td>
<td>FloorButton, ElevatorButton, Elevator, Floor</td>
</tr>
<tr>
<td>Floor</td>
<td>FloorButton, Light</td>
</tr>
<tr>
<td>FloorButton</td>
<td>Elevator</td>
</tr>
<tr>
<td>ElevatorButton</td>
<td>Elevator</td>
</tr>
<tr>
<td>Door</td>
<td>Person</td>
</tr>
<tr>
<td>Bell</td>
<td></td>
</tr>
<tr>
<td>Light</td>
<td></td>
</tr>
<tr>
<td>Building</td>
<td>Clock, Scheduler, Elevator</td>
</tr>
</tbody>
</table>

**Fig. 6.13** Complete class diagram with visibility notations.

**Fig. 6.14** List of handles for each class.
Implementation: Class Header Files

Now that we have discussed programming C++ classes, we are ready to begin writing the code for our elevator simulator. In this section, we examine the header files for each class in our system. In the “Thinking About Objects” section at the end of Chapter 7, we present the complete, working C++ code for the simulator. In Chapter 9, we modify that code to incorporate inheritance.

To demonstrate the order in which constructors and destructors run, we will code a constructor and destructor for each of our classes that simply display messages indicating that these functions are running. We include the constructor and destructor prototypes in our header files; we include their implementations in the .cpp files presented in Chapter 7.

Figure 6.15 lists the header file for class Bell. Working from our class diagram (Fig. 6.13), we declare a constructor, a destructor (lines 8 and 9) and the member function ringBell (line 10); each of these member functions has public visibility. We have identified no other public or private elements for this class, so our header file is complete.

Figure 6.16 lists the header file for class Clock. We include a constructor and destructor (lines 8 and 9) and the public member functions tick() and getTime() (lines 10 and 11) from Fig. 6.13. We implement the time attribute in our class header file by declaring a private data member time of type int (line 13). Once per second in our simulation, an object of class Building invokes the getTime member function of an object of class Clock to get the current value of time and invokes the tick member function to increment time.

```cpp
1 // bell.h
2 // Definition for class Bell.
3 ifndef BELL_H
4 define BELL_H
5
6 class Bell {
7 public:
8 Bell(); // constructor
9 ~Bell(); // destructor
10 void ringBell(); // ring the bell
11
12
13 } // BELL_H
```

**Fig. 6.15** Class Bell header file.

```cpp
1 // clock.h
2 // Definition for class Clock.
3 ifndef CLOCK_H
4 define CLOCK_H
5
6 class Clock {
7 public:
8 Clock(); // constructor
9 ~Clock(); // destructor
10 void tick(); // increment clock by one second
```

**Fig. 6.16** Class Clock header file (part 1 of 2).
Figure 6.17 lists the header file for class **Person**. We declare the **ID** attribute (line 16) and (in lines 12–14) the operations **stepOntoFloor**, **enterElevator** and **exitElevator** from our class diagram (Fig. 6.13). We also declare a **public getID** member function (line 10) that returns the person’s **ID** number. We will use this operation to keep track of the people in our simulation.

Objects of class **Person** are not created at the beginning of the simulation—they are randomly created dynamically as the simulation runs. For this reason, we must implement objects of class **Person** differently from objects of our other classes in our system. After we discuss how to create new objects dynamically in Chapter 7, we will add significant elements to the header file for class **Person**.

Figure 6.18 lists the header file for class **Door**. We declare a constructor and a destructor (lines 8 and 9), the **public** member functions **openDoor** and **closeDoor** in lines 11 and 12. We declare the **private** class data member **open** in line 14. The table in Fig. 6.14 states that class **Door** needs a handle to class **Person**. However, because objects of class **Person** are created dynamically in our system, we are unsure at this point how to implement handles to objects of class **Person**. After we discuss dynamically created objects in Chapter 7, we will have a better idea of how to implement handles to class **Person**.
We list the header file for class `Light` in Figure 6.19. The information from the class diagram in Fig. 6.13 leads us to declare `public` member functions `turnOn` and `turnOff` and `private` data member `on` of type `bool`. In this header file, we also introduce something new to our implementation—the need to distinguish between different objects of the same class in our system. We know that the simulation contains two objects of class `Light`: one object belongs to the first floor, and the other object belongs to the second floor. We want to be able to distinguish between these two objects for output purposes, so we need to give a name to each. Therefore, we add line 14

```cpp
char *name;          // which floor the light is on
```

to the `private` section of the class declaration. We also add a `char *` parameter to the constructor (line 8), so that it can initialize the name of each object of class `Light`. 

```cpp
// door.h
// Definition for class Door.
#ifndef DOOR_H
#define DOOR_H

class Door {
public:
  Door();                    // constructor
  ~Door();                   // destructor

  void openDoor();
  void closeDoor();

private:
  bool open;                 // open or closed
};
#endif // DOOR_H

Fig. 6.18 Class Door header file.
```

```cpp
// light.h
// Definition for class Light.
#ifndef LIGHT_H
#define LIGHT_H

class Light {
public:
  Light( char * );        // constructor
  ~Light();               // destructor
  void turnOn();          // turns light on
  void turnOff();         // turns light off

private:
  bool on;                // on or off
  char *name;
};
#endif // LIGHT_H

Fig. 6.19 Class Light header file.
```
Figure 6.20 lists the header file for class Building. The public section of the class declaration includes a constructor, a destructor and the runSimulation member function from Fig. 6.13. When we first identified the runSimulation operation in Chapter 4, we did not know what object would invoke the function to begin the simulation. Now that we have discussed classes in C++, we know that an object of class Building needs to be declared in main and that main will then invoke runSimulation. The code in the main program is:

```cpp
Building building;         // create the building object
building.runSimulation(); // invoke runSimulation
```

We also choose to include a parameter of type int in the runSimulation declaration. The building object will run the elevator simulation for the number of seconds passed to the object via this parameter. The preceding invocation of runSimulation would then include a number indicating the duration of the simulation. The table in Fig. 6.14 indicates that class Building needs handles to its composite objects. We cannot implement these handles at this point, because we have not discussed composition. Therefore, we delay the implementation of the component objects of class Building until Chapter 7 (see the comments in lines 14–18 in Fig. 6.20).

Figure 6.21 lists the header file for class ElevatorButton. We declare the pressed attribute, the pressButton and resetButton member functions (from the class diagram in Fig. 6.13) and the constructor and destructor. Figure 6.14 states that class ElevatorButton needs a handle to the elevator. In line 19

```cpp
Elevator &elevatorRef;
```

we include this handle (notice that we use a reference to implement the handle). In Chapter 7, we discuss how to send messages to the elevator using this reference.

```cpp
// building.h
// Definition for class Building.
#ifndef BUILDING_H
#define BUILDING_H

class Building {
public:
  Building();       // constructor
  ~Building();      // destructor

  // runs simulation for specified amount of time
  void runSimulation( int );

private:
  // In Chapter 7, we show how to include:
  // one object of class Clock
  // one object of class Scheduler
  // one object of class Elevator
  // two objects of class Floor
};
#endif // BUILDING_H
```

Fig. 6.20 Class Building header file.
A reference must be initialized when it is declared, but we are not allowed to assign a value to class data member in the header file. Therefore, a reference must be initialized in the constructor; we pass an `Elevator` reference to the constructor as a parameter in line 10. Line 6

```
class Elevator; // forward declaration
```

is a *forward declaration* of class `Elevator`. The forward declaration allows us to declare a reference to an object of class `Elevator` without needing to include the header file for class `Elevator` in the header file for class `ElevatorButton`.²

Figure 6.22 lists the header file for class `FloorButton`. This header file is identical to the header file for class `ElevatorButton`, except we declare a *private* data member `floorNumber` of type `int`. Objects of class `FloorButton` need to know to which floor they belong for simulator output purposes. The floor number is passed in as a constructor argument for initialization purposes (line 10).

Figure 6.23 lists the header file for class `Scheduler`. In lines 23 and 24

```
int floor1ArrivalTime;
int floor2ArrivalTime;
```

we declare the *private* data members of class `Scheduler` that correspond to the attributes we have identified for this class (Fig. 6.13). In line 12, we declare the *public* member function `processTime` that corresponds to the operation we identified in the “Thinking About Objects” section at the end of Chapter 4.

---

2. Using the forward declaration (where possible) instead of including the full header file helps avoid a preprocessor problem called a circular include. We discuss the circular include problem in more detail in Chapter 7.
1 // floorButton.h
2 // Definition for class FloorButton.
3 #ifndef FLOORBUTTON_H
4 #define FLOORBUTTON_H
5
6 class Elevator;                     // forward declaration
7
8 class FloorButton {
9 public:
10    FloorButton( int, Elevator & ); // constructor
11    ~FloorButton();                  // destructor
12
13    void pressButton();              // press the button
14    void resetButton();              // reset the button
15
16 private:
17    int floorNumber;                 // number of the button's floor
18    bool pressed;                    // state of button
19
20    // reference to button's elevator
21    Elevator &elevatorRef;
22};
23
24 #endif // FLOORBUTTON_H

Fig. 6.22 Class FloorButton header file.

1 // scheduler.h
2 // definition for class Scheduler
3 #ifndef SCHEDULER_H
4 #define SCHEDULER_H
5
6 class Floor;                        // forward declaration
7
8 class Scheduler {
9 public:
10    Scheduler( Floor &, Floor & ); // constructor
11    ~Scheduler();                  // destructor
12    void processTime( int );       // set scheduler's time
13
14 private:
15
16    // method that schedules arrivals to a specified floor
17    void scheduleTime( Floor & );
18
19    // method that delays arrivals to a specified floor
20    void delayTime( Floor & );
21
22    Floor &floor1Ref;
23    Floor &floor2Ref;
24    int floor1ArrivalTime;
25    int floor2ArrivalTime;
26};

Fig. 6.23 Class Scheduler header file (part 1 of 2).
In lines 15–19 we declare the functions we identified in the sequence diagram in Fig. 4.27. Each of these functions takes as a parameter a reference to an object of class Floor. Note that we did not list these functions as operations (i.e., public member functions), because these methods are not invoked by client objects. Instead, these methods are used only by class Scheduler to perform its own internal actions. Therefore, we place these methods in the private section of the class declaration.

In lines 21 and 22, we declare the handles identified in Fig. 6.14. Again, we implement each handle as a reference to an object of class Floor. Class Scheduler needs these handles so that it can send the isOccupied message to the two floors in the simulation (see diagram in Fig. 4.27). We must also make a forward declaration of class Floor in line 6 so that we may declare the references.

Figure 6.24 contains the header file for class Floor. We declare the public member functions elevatorArrived, isOccupied and personArrives from Fig. 6.13. We also declare the public member function elevatorLeaving in line 26. We add this member function so the elevator can tell the floor when the elevator is preparing to leave. The elevator invokes the elevatorLeaving operation, and the floor responds by turning off its light.
In line 31, we add a `private` `floorNumber` data member to the class—we add this value for output purposes, just as we did with the `floorNumber` data member of class `FloorButton`. We also add a parameter of type `int` to the constructor so the constructor can initialize that data member. We also declare the handle to class `Elevator` identified in Fig. 6.14. We defer declaration of the component members of class `Floor` (see lines 28 and 33) until Chapter 7.

We list the header file for class `Elevator` in Fig. 6.25. In the `public` section of the header file, we declare the `summonElevator`, `prepareToLeave` and `processTime` operations listed in Fig. 6.13. To differentiate between people on the floor and people in the elevator, we rename the last two operations listed under class `Elevator`. We call these operations `passengerEnters` and `passengerExits`, and we declare them in the `public` section of the header file. We also declare a reference to each of the two floors (lines 38-39); the constructor (line 10) initializes these references.
In the private section of the header file, we declare the moving, direction, currentFloor and arrivalTime attributes from Fig. 6.13. We do not need to declare the capacity attribute; instead, we will write our code to ensure that only one person may be on the elevator at a time.

**Conclusion**

In the next “Thinking About Objects” section, we present the full C++ code for our elevator simulation. We will use the concepts presented in the next chapter to implement composite relationships, dynamic creation of objects of class Person and static and const data members and functions. In the “Thinking About Objects” section at the end of Chapter 9, we use inheritance to further improve our object-oriented elevator simulator design and implementation.

**SUMMARY**

- Structures are aggregate data types built using data of other types.
- The keyword struct introduces a structure definition. The body of a structure is delineated by braces ({ and }). Every structure definition must end with a semicolon.
- A structure tag name can be used to declare variables of a structure type.
- Structure definitions do not reserve space in memory; they create new data types that are used to declare variables.
- Members of a structure or a class are accessed using the member access operators—the dot operator (.) and the arrow operator (⇒). The dot operator accesses a structure member via the object’s variable name or a reference to the object. The arrow operator accesses a structure member via a pointer to the object.
• Drawbacks to creating new data types with struct are the possibility of having uninitialized data; improper initialization; all programs using a struct must be changed if the struct implementation changes and no protection is provided to ensure that data are kept in a consistent state with proper data values.

• Classes enable the programmer to model objects with attributes and behaviors. Class types can be defined in C++ using the keywords class and struct, but keyword class is normally used for this purpose.

• The class name can be used to declare objects of that class.

• Class definitions begin with the keyword class. The body of the class definition is delineated with braces ({ and }). Class definitions terminate with a semicolon.

• Any data member or member function declared after public: in a class is visible to any function with access to an object of the class.

• Any data member or member function declared after private: is only visible to friends and other members of the class.

• Member access specifiers always end with a colon (:) and can appear multiple times and in any order in a class definition.

• Private data are not accessible from outside the class.

• The implementation of a class should be hidden from its clients.

• A constructor is a special member function with the same name as the class that is used to initialize the members of a class object. The constructor is called when an object of that class is instantiated.

• The function with the same name as the class but preceded with a tilde character (~) is called a destructor.

• The set of public member functions of a class is called the class’ interface or public interface.

• When a member function is defined outside the class definition, the function name is preceded by the class name and the binary scope resolution operator (::).

• Member functions defined using the scope resolution operator outside a class definition are within that class’ scope.

• Member functions defined in a class definition are automatically inlined. The compiler reserves the right not to inline any function.

• Calling member functions is more concise than calling functions in procedural programming because most data used by the member function are directly accessible in the object.

• Within a class’ scope, class members may be referenced simply by their names. Outside a class’ scope, class members are referenced through either an object name, a reference to an object or a pointer to an object.

• Member selection operators . and -> are used to access class members.

• A fundamental principle of good software engineering is separating interface from implementation.

• Class definitions are normally placed in header files and member function definitions are normally placed in source-code files of the same base name.

• The default access mode for classes is private so all members after the class header and before the first member access specifier are considered to be private.

• A class’ public members present a view of the services the class provides to the class’ clients.

• Access to a class’ private data can be carefully controlled via member functions called access functions. If a class wants to allow clients to read private data, the class can provide a get function. To enable clients to modify private data, the class can provide a set function.
• Data members of a class are normally made private and member functions of a class are normally made public. Some member functions may be private and serve as utility functions to the other functions of the class.

• Data members of a class cannot be initialized in a class definition. They must be initialized in a constructor, or their values may be set after their object is created.

• Constructors can be overloaded.

• Once a class object is properly initialized, all member functions that manipulate the object should ensure that the object remains in a consistent state.

• When an object of a class is declared, initializers can be provided. These initializers are passed to the class’ constructor.

• Constructors can specify default arguments.

• Constructors may not specify return types, nor may they attempt to return values.

• If no constructor is defined for a class, the compiler creates a default constructor. A default constructor supplied by the compiler does not perform any initialization, so when the object is created, it is not guaranteed to be in a consistent state.

• The destructor of an automatic object is called when the object goes out of scope. The destructor itself does not actually destroy the object, but it does perform termination housekeeping before the system reclaims the object’s storage.

• Destructors do not receive parameters and do not return values. A class may have only one destructor (destructors cannot be overloaded).

• The assignment operator (\textbf{=} ) is used to assign an object to another object of the same type. Such assignment is normally performed by default memberwise copy. Memberwise copy is not ideal for all classes.

**TERMINOLOGY**

\texttt{&} reference operator
abstract data type (ADT)
access function
arrow member selection operator (\texttt{\textasciitilde\textgreater\textgreater})
attribute
behavior
binary scope resolution operator (\texttt{::})
class
class definition
class member selector operator (\texttt{.})
class scope
client of a class
consistent state for a data member
constructor
data member
data type
default constructor
destructor
dot member selection operator (\texttt{.})
encapsulation
extensibility
file scope

get function
global object
header file
helper function
implementation of a class
information hiding
initialize a class object
inline member function
instance of a class
instantiate an object of a class
interface to a class
member access control
member access specifiers
member function
member initializer
member selection operator (\texttt{. and \textasciitilde\textgreater\textgreater})
memberwise copy
message
nonmember function
nonstatic local object
object
object-oriented design (OOD)
COMMON PROGRAMMING ERRORS

6.1 The expression \((\ast\text{timePtr}).\text{hour}\) refers to the \text{hour} member of the \textit{struct} pointed to by \texttt{timePtr}. Omitting the parentheses, as in \texttt{\ast\ast\text{timePtr}.\text{hour}} would be a syntax error because \texttt{.} has a higher precedence than \texttt{\ast}, so the expression would execute as if parenthesized as \texttt{\ast\ast\text{(timePtr.\text{hour})}}. This would be a syntax error because with a pointer you must use the arrow operator to refer to a member.

6.2 Forgetting the semicolon at the end of a class (or structure) definition is a syntax error.

6.3 Specifying a return type and/or a return value for a constructor is a syntax error.

6.4 Attempting to initialize a data member of a class explicitly in the class definition is a syntax error.

6.5 When defining a class’ member functions outside that class, omitting the class name and scope resolution operator on the function name is an error.

6.6 An attempt by a function, which is not a member of a particular class (or a \texttt{friend} of that class), to access a \texttt{private} member of that class is a syntax error.

6.7 Data members of a class cannot be initialized in the class definition.

6.8 Attempting to declare a return type for a constructor and/or attempting to return a value from a constructor are syntax errors.

6.9 Specifying default initializers for the same member function in both a header file and in the member function definition.

6.10 It is a syntax error to attempt to pass arguments to a destructor, to specify a return type for a destructor (even \texttt{void} cannot be specified), to return values from a destructor or to overload a destructor.

6.11 A constructor can call other member functions of the class such as \textit{set} or \textit{get} functions, but because the constructor is initializing the object, the data members may not yet be in a consistent state. Using data members before they have been properly initialized can cause logic errors.
GOOD PROGRAMMING PRACTICES

6.1 Use each member access specifier only once in a class definition for clarity and readability. Place `public` members first where they are easy to locate.

6.2 Use the name of the header file with the period replaced by an underscore in the `#ifndef` and `#define` preprocessor directives of a header file.

6.3 If you choose to list the `private` members first in a class definition, explicitly use the `private` member access specifier despite the fact that `private` is assumed by default. This improves program clarity. Our preference is to list the `public` members of a class first to emphasize the class' interface.

6.4 Despite the fact that the `public` and `private` member access specifiers may be repeated and intermixed, list all the `public` members of a class first in one group and then list all the `private` members in another group. This focuses the client's attention on the class' `public` interface, rather than on the class' implementation.

6.5 When appropriate (almost always), provide a constructor to ensure that every object is properly initialized with meaningful values. Pointer data members, in particular, should be initialized to some legitimate pointer value or to 0.

6.6 Declare default function argument values only in the function prototype within the class definition in the header file.

6.7 Member functions that set the values of private data should verify that the intended new values are proper; if they are not, the set functions should place the `private` data members into an appropriate consistent state.

6.8 Never have a `public` member function return a non-const reference (or a pointer) to a `private` data member. Returning such a reference violates the encapsulation of the class. In fact, returning any reference or pointer to `private` data still makes the client code dependent on the representation of the class' data. So, returning pointers or references to `private` data should be avoided.

PERFORMANCE TIPS

6.1 By default, structures are passed call-by-value. To avoid the overhead of copying a structure, pass the structure call-by-reference.

6.2 Defining a small member function inside the class definition automatically inlines the member function (if the compiler chooses to do so). This can improve performance, but it does not promote the best software engineering because clients of the class will be able to see the implementation of the function and their code must be recompiled if the inline function definition changes.

6.3 Actually, objects contain only data, so objects are much smaller than if they also contained functions. Applying operator `sizeof` to a class name or to an object of that class will report only the size of the class' data. 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 of the member functions. Each object, of course, needs its own copy of the class' data because these data can vary among the objects. The function code is nonmodifiable (also called reentrant code or pure procedure) and hence can be shared among all objects of one class.

6.4 Passing an object call-by-value is good from a security standpoint because the called function has no access to the original object, but call-by-value can degrade performance when making a copy of a large object. An object can be passed call-by-reference by passing either a pointer or a reference to the object. Call-by-reference offers good performance but is weaker from a security standpoint because the called function is given access to the original object. Call-by-const-reference is a safe, good-performing alternative.
SOFTWARE ENGINEERING OBSERVATIONS

6.1 To avoid the overhead of call-by-value yet still gain the benefit that the caller’s original data are protected from modification, pass large-size arguments as `const` references.

6.2 It is important to write programs that are understandable and easy to maintain. Change is the rule rather than the exception. Programmers should anticipate that their code will be modified. As we will see, classes can facilitate program modifiability.

6.3 Clients of a class use the class without knowing the internal details of how the class is implemented. If the class implementation is changed (to improve performance, for example), provided the class’ interface remains constant, the class’ client source code need not change (although the client may need to be recompiled). This makes it much easier to modify systems.

6.4 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 and/or by member functions that store new data. Because the data are already in the object, the member function calls often have no arguments or at least have fewer arguments than typical function calls in non-object-oriented languages. Thus, the calls are shorter, the function definitions are shorter and the function prototypes are shorter.

6.5 Clients have access to a class’ interface but should not have access to a class’ implementation.

6.6 Declaring member functions inside a class definition (via their function prototypes) and defining those member functions outside that class definition separates the interface of a class from its implementation. This promotes good software engineering. Clients of a class cannot see the implementation of that class’ member functions and need not recompile if that implementation changes.

6.7 Only the simplest member functions and most stable member functions (i.e., the implementation is unlikely to change) should be defined in the class header.

6.8 Using an object-oriented programming approach can often simplify function calls by reducing the number of parameters to be passed. This benefit of object-oriented programming derives from the fact that encapsulation of data members and member functions within an object gives the member functions the right to access the data members.

6.9 A central theme of this book is “reuse, reuse, reuse.” We will carefully discuss a number of techniques for “polishing” classes to encourage reuse. We focus on “crafting valuable classes” and creating valuable “software assets.”

6.10 Place the class declaration in a header file to be included by any client that wants to use the class. This forms the class’ `public` interface (and provides the client with the function prototypes it needs to be able to call the class’ member functions). Place the definitions of the class member functions in a source file. This forms the implementation of the class.

6.11 Clients of a class do not need access to the class’ source code in order to use the class. The clients do, however, need to be able to link to the class’ object code. This encourages independent software vendors (ISVs) to provide class libraries for sale or license. The ISVs provide in their products only the header files and the object modules. No proprietary information is revealed—as would be the case if source code were provided. The C++ user community benefits by having more ISV-produced class libraries available.

6.12 Information important to the interface to a class should be included in the header file. Information that will be used only internally in the class and will not be needed by clients of the class should be included in the unpublished source file. This is yet another example of the principle of least privilege.

6.13 C++ encourages programs to be implementation independent. When the implementation of a class used by implementation-independent code changes, that code need not be modified.
If any part of the interface of the class changes, the implementation-independent code must be recompiled.

6.14 Keep all the data members of a class **private**. Provide **public** member functions to set the values of **private** data members and to get the values of **private** data members. This architecture helps hide the implementation of a class from its clients, which reduces bugs and improves program modifiability.

6.15 Class designers use **private**, **protected** and **public** members to enforce the notion of information hiding and the principle of least privilege.

6.16 The class designer need not provide **set** and/or **get** functions for each **private** data item; these capabilities should be provided only when appropriate. If the service is useful to the client code, that service should be provided in the class’s **public** interface.

6.17 Member functions tend to fall into a number of different categories: functions that read and return the value of **private** data members; functions that set the value of **private** data members; functions that implement the services of the class and functions that perform various mechanical chores for the class such as initializing class objects, assigning class objects, converting between classes and built-in types or between classes and other classes and handling memory for class objects.

6.18 A phenomenon of object-oriented programming is that once a class is defined, creating and manipulating objects of that class usually involves issuing only a simple sequence of member function calls—few, if any, control structures are needed. By contrast, it is common to have control structures in the implementation of a class’ member functions.

6.19 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.

6.20 It is possible for a class not to have a default constructor if any constructors are defined and none of them is explicitly a default constructor.

6.21 As we will see (throughout the remainder of the book), constructors and destructors have much greater prominence in C++ and object-oriented programming than is possible to convey after only our brief introduction here.

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

6.23 Accessing **private** data through **set** and **get** member functions not only protects the data members from receiving invalid values, but it also insulates clients of the class from the representation of the data members. Thus, if the representation of the data changes for some reason (typically to reduce the amount of storage required or to improve performance), only the member functions need to change—the clients need not change as long as the interface provided by the member functions remains the same. The clients may, however, need to be recompiled.

**TESTING AND DEBUGGING TIPS**

6.1 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 and/or the wrong number of arguments.

6.2 Use **#ifndef**, **#define** and **#endif** preprocessor directives to prevent header files from being included more than once in a program.
6.3 Making the data members of a class **private** and the member functions of the class **public** facilitates debugging because problems with data manipulations are localized to either the class’ member functions or the **friends** of the class.

6.4 Every member function (and **friend**) that modifies the **private** data members of an object should ensure that the data remains in a consistent state.

6.5 The benefits of data integrity are not automatic simply because data members are made **private**—the programmer must provide the validity checking. C++ does, however, provide a framework in which programmers can design better programs in a convenient manner.

**SELF-REVIEW EXERCISES**

6.1 Fill in the blanks in each of the following:

a) The keyword **struct** introduces a structure definition.

b) Class members are accessed via the **dot** operator in conjunction with the name of an object of the class or via the **arrow** operator in conjunction with a pointer to an object of the class.

c) Members of a class specified as **private** are accessible only to member functions of the class and **friends** of the class.

d) A **constructor** is a special member function used to initialize the data members of a class.

e) The default access for members of a class is **private**.

f) A **set** function is used to assign values to **private** data members of a class.

g) **Default memberwise copy** (performed by the assignment operator) can be used to assign an object of a class to another object of the same class.

h) Member functions of a class are normally made **public** and data members of a class are normally made **private**.

i) A **get** function is used to retrieve values of **private** data of a class.

j) The set of **public** member functions of a class is referred to as the class’ **interface**.

k) A class implementation is said to be hidden from its clients or **encapsulated**.

l) The keywords **class** and **struct** can be used to introduce a class definition.

m) Members of a class specified as **public** are accessible anywhere an object of the class is in scope.

6.2 Find the error(s) in each of the following and explain how to correct it:

a) Assume the following prototype is declared in class **Time**:

   ```cpp
   void ~Time( int );
   ```

b) The following is a partial definition of class **Time**.

   ```cpp
   class Time {
   public:
      // function prototypes
      private:
      int hour = 0;
      int minute = 0;
      int second = 0;
   }
   ```

c) Assume the following prototype is declared in class **Employee**:

   ```cpp
   int Employee( const char *, const char * );
   ```

**ANSWERS TO SELF-REVIEW EXERCISES**

6.1  a) **struct**, b) dot (.), arrow (->), c) **private**, d) constructor, e) **private**, f) set, g) Default memberwise copy (performed by the assignment operator), h) **public**, i) **private**, j) interface, k) encapsulated, l) **class**. **struct**. m) **public**.
a) Error: Destructors are not allowed to return values or take arguments.
Correction: Remove the return type `void` and the parameter `int` from the declaration.

b) Error: Members cannot be explicitly initialized in the class definition.
Correction: Remove the explicit initialization from the class definition and initialize the data members in a constructor.

c) Error: Constructors are not allowed to return values.
Correction: Remove the return type `int` from the declaration.

EXERCISES

6.3 What is the purpose of the scope resolution operator?

6.4 Compare and contrast the notions of `struct` and `class` in C++.

6.5 Provide a constructor that is capable of using the current time from the `time()` function—declared in the C Standard Library header `time.h`—to initialize an object of the `Time` class.

6.6 Create a class called `Complex` for performing arithmetic with complex numbers. Write a driver program to test your class.

Complex numbers have the form

$$ realPart + imaginaryPart \times i $$

where $i$ is

$$ \sqrt{-1} $$

Use `double` variables to represent the private data of the class. Provide a constructor function that enables an object of this class to be initialized when it is declared. The constructor should contain default values in case no initializers are provided. Provide public member functions for each of the following:

a) Addition of two `Complex` numbers: The real parts are added together and the imaginary parts are added together.

b) Subtraction of two `Complex` numbers: The real part of the right operand is subtracted from the real part of the left operand and the imaginary part of the right operand is subtracted from the imaginary part of the left operand.

c) Printing `Complex` numbers in the form `(a, b)` where $a$ is the real part and $b$ is the imaginary part.

6.7 Create a class called `Rational` for performing arithmetic with fractions. Write a driver program to test your class.

Use integer variables to represent the private data of the class—the numerator and the denominator. Provide a constructor function that enables an object of this class to be initialized when it is declared. The constructor should contain default values in case no initializers are provided and should store the fraction in reduced form (i.e., the fraction

$$ \frac{2}{4} $$

would be stored in the object as 1 in the numerator and 2 in the denominator). Provide public member functions for each of the following:

a) Addition of two `Rational` numbers. The result should be stored in reduced form.

b) Subtraction of two `Rational` numbers. The result should be stored in reduced form.

c) Multiplication of two `Rational` numbers. The result should be stored in reduced form.
d) Division of two Rational numbers. The result should be stored in reduced form.
e) Printing Rational numbers in the form $a/b$ where $a$ is the numerator and $b$ is the
denominator.
f) Printing Rational numbers in floating-point format.

6.8 Modify the Time class of Fig. 6.10 to include a tick member function that increments the
time stored in a Time object by one second. The Time object should always remain in a consistent state. Write a driver program that tests the tick member function in a loop that prints the time in standard format during each iteration of the loop to illustrate that the tick member function works correctly. Be sure to test the following cases:
a) Incrementing into the next minute.
b) Incrementing into the next hour.
c) Incrementing into the next day (i.e., 11:59:59 PM to 12:00:00 AM).

6.9 Modify the Date class of Fig. 6.12 to perform error checking on the initializer values for data members month, day and year. Also, provide a member function nextDay to increment the day by one. The Date object should always remain in a consistent state. Write a driver program that tests the nextDay function in a loop that prints the date during each iteration of the loop to illustrate that the nextDay function works correctly. Be sure to test the following cases:
a) Incrementing into the next month.
b) Incrementing into the next year.

6.10 Combine the modified Time class of Exercise 6.8 and the modified Date class of Exercise 6.9 into one class called DateAndTime (in Chapter 9 we will discuss inheritance, which will enable us to accomplish this task quickly without modifying the existing class definitions). Modify the ticks function to call the nextDay function if the time is incremented into the next day. Modify the printStandard and printMilitary to output the date in addition to the time. Write a driver program to test the new class DateAndTime. Specifically, test incrementing the time into the next day.

6.11 Modify the set functions in the program of Fig. 6.10 to return appropriate error values if an attempt is made to set a data member of an object of class Time to an invalid value.

6.12 Create a class Rectangle. The class has attributes length and width, each of which defaults to 1. It has member functions that calculate the perimeter and the area of the rectangle. It has set and get functions for both length and width. The set functions should verify that length and width are each floating-point numbers larger than 0.0 and less than 20.0.

6.13 Create a more sophisticated Rectangle class than the one you created in Exercise 6.12. This class stores only the Cartesian coordinates of the four corners of the rectangle. The constructor calls a set function that accepts four sets of coordinates and verifies that each of these is in the first quadrant with no single x or y coordinate larger than 20.0. The set function also verifies that the supplied coordinates do, in fact, specify a rectangle. Member functions calculate the length, width, perimeter and area. The length is the larger of the two dimensions. Include a predicate function square that determines if the rectangle is a square.

6.14 Modify the Rectangle class of Exercise 6.13 to include a draw function that displays the rectangle inside a 25-by-25 box enclosing the portion of the first quadrant in which the rectangle resides. Include a setFillCharacter function to specify the character out of which the body of the rectangle will be drawn. Include a setPerimeterCharacter function to specify the character that will be used to draw the border of the rectangle. If you feel ambitious, you might include functions to scale the size of the rectangle, rotate it, and move it around within the designated portion of the first quadrant.

6.15 Create a class HugeInteger that uses a 40-element array of digits to store integers as large as 40-digits each. Provide member functions inputHugeInteger, outputHugeInteger,
addHugeIntegers and subtractHugeIntegers. For comparing HugeInteger objects, provide functions isEqualTo, isNotEqualTo, isGreaterThan, isLessThan, isGreaterThanOrEqualTo and isLessThanOrEqualTo—each of these is a “predicate” function that simply returns true if the relationship holds between the two huge integers and returns false if the relationship does not hold. Provide a predicate function isZero. If you feel ambitious, also provide member functions multiplyHugeIntegers, divideHugeIntegers and modulusHugeIntegers.

6.16 Create a class TicTacToe that will enable you to write a complete program to play the game of tic-tac-toe. The class contains as private data a 3-by-3 double array of integers. The constructor should initialize the empty board to all zeros. Allow two human players. Wherever the first player moves, place a 1 in the specified square; place a 2 wherever the second player moves. Each move must be to an empty square. After each move, determine if the game has been won or if the game is a draw. If you feel ambitious, modify your program so that the computer makes the moves for one of the players automatically. Also, allow the player to specify whether he or she wants to go first or second. If you feel exceptionally ambitious, develop a program that will play three-dimensional tic-tac-toe on a 4-by-4-by-4 board (Caution: This is an extremely challenging project that could take many weeks of effort!).