This chapter discusses software design and introduces tools and techniques used by professional programmers to facilitate the design and maintenance (upkeep) of large-scale programs. You will see how software designers use abstraction to create models of computer systems and how they use abstract data types to encapsulate data elements and operators. You will learn how to specify the behavior of an abstract data type (ADT) using an interface and how to implement it using classes. The use of interfaces enables programmers to develop systems with a high degree of flexibility and also enables a programmer to code program systems that have interchangeable parts.
You will see how interfaces, preconditions, and postconditions serve as con-
tracts between system designers and programmers. They also serve as a contract
between application programmers who use an API (Application Programming
Interface) to develop applications software and those who design and implement the
classes that are part of the API.

Through examination of a large case study, you will learn how to follow a five-
step software development process to design and implement new software applica-
tions. These steps include problem specification, analysis, design, implementation,
and testing. You will see how to employ use cases to specify the interaction between
the system and its users during problem specification and analysis. You will see how
to use sequence diagrams and procedural abstraction to develop the algorithms for
the class member functions during the design phase. Carefully following the soft-
ware development process will lead to more efficient implementations with fewer
errors, making it easier to test and debug the completed system.

1.1 The Software Life Cycle

The goal of a college programming assignment is to give you some experience with
a particular concept, thus reinforcing your mastery of it. The resulting program is
used briefly to demonstrate to the instructor that you have mastered the assignment.
The only users of the program are you (its author) and perhaps the instructor.

In industry, on the other hand, a software product is expected to be used for an
extended period of time by someone who did not write the program and who is not
intimately familiar with its internal design. The impetus for a software project
comes from users of an existing software product or potential users of a new soft-
ware product. The users see a need for improving the operation of an existing prod-
uct or for computerizing an operation that is currently done without the use of computers. This need is communicated to the individual(s) responsible for developing new software products in the organization (normally called system analysts). A team of system analysts and programmers are responsible for guiding the development of a new software system from its inception to its completion. We discuss this process next.

A software product goes through several stages as it matures from an initial concept to a finished product in regular use. This series of stages could be compared to the stages that an insect goes through as it develops from egg, to larva, to pupa, to adult. This sequence of stages is known as the life cycle; thus, we speak of a software life cycle.

Software products can require years to develop and require the efforts of many individuals. A successful software product will be used for many years after it is first released. During the term of its use, new or updated versions may be released that contain changes to fit new situations or that fix errors that were previously undetected. For this reason, it is important to design and document software in an organized way so that it can be easily understood and maintained after its initial release. This is especially important because the person who maintains the software may not have been involved in its original design.

### Software Life Cycle Models

Many different views of the software life cycle have been proposed over the years and are in use today. There are also many different ways of organizing the activities that transform the software from one stage to another. The simplest version is the waterfall model, in which the activities are performed in sequence and the result of one flows into the next (indicated by downward-pointing arrows in Figure 1.1), much like water flowing down a cascading waterfall.

Table 1.1 describes the activities that are performed in each of the five phases of the waterfall model. We will provide more details later in this section.

<table>
<thead>
<tr>
<th>Table 1.1</th>
<th>Waterfall Model of the Software Life Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Requirements</td>
<td>The requirements for the software system are determined.</td>
</tr>
<tr>
<td>2. Analysis</td>
<td>The requirements are studied and clarified, and the overall architecture of the solution is determined. Each major subsystem is analyzed, and its component classes are determined. Also, any interaction between components is determined.</td>
</tr>
<tr>
<td>3. Design</td>
<td>Functions and data fields are defined for classes. Detailed algorithms for the functions are defined.</td>
</tr>
<tr>
<td>4. Implementation</td>
<td>The individual classes and functions are coded in the target programming language.</td>
</tr>
<tr>
<td>5. Testing</td>
<td>The functions of each class are tested in isolation and as a class (unit testing). The functions and classes are tested together (integration testing) to verify that they work together and meet the requirements.</td>
</tr>
</tbody>
</table>
Although simple in concept, the waterfall model has proved to be unworkable in practice. The fundamental flaw in this model is the assumption that each stage can and must be completed before the next one starts. That is rarely the case in actual practice. For example, system designers may identify incomplete or inconsistent requirements during the design process, or programmers may find during the implementation phase that there are areas of the design that are incomplete or inconsistent. Sometimes it is not until the product is finished and the user first sees it that the user’s requirements are “specified” in the form of the complaint, “That’s not what I meant!”

Various alternatives have been proposed. The common theme is to develop a software product in stages or cycles. Each cycle is a mini-version of the waterfall, with varying amounts of emphasis on the different activities for the particular cycles. At the end of each cycle there is a review with the users to obtain feedback, which will be taken into account in the next cycle.

One example of such a model is the Unified Model, shown in Figure 1.2. The cycles, called phases and iterations, are shown along the horizontal axis, and the activities, called workflows, are shown down the vertical axis. The four phases are inception, elaboration, construction, and transition (switching over to the new system). Time moves across the horizontal axis from iteration 1 to iteration $n$, and each iteration is a mini-waterfall. The five activities are the same as in the simple waterfall model. The shaded areas under the curves next to each activity are intended to show the relative amount of effort spent on that activity during each iteration. For example, during the inception phase (iterations 1 and 2), most effort is spent on specifying requirements. In fact, requirements specification is the only activity performed during iteration 1. During iteration 2, some effort is also spent on analysis, with a tiny amount on design and implementation. As the software developers move into the
elaboration phase, they continue to work on requirements and analysis but also start to spend more time on design and implementation, particularly toward the end of the elaboration phase. In the construction phase, the requirements specification and analysis activities are completed, and most effort is spent on design and implementation. The diagram also shows that some time is spent on testing during all phases after inception, but more time is spent on testing during the construction and transition phases. We will discuss testing in more detail in Chapter 2.

**Software Life Cycle Activities**

Independently of how they are organized, there is general consensus that the activities shown in Table 1.2 are essential for the development of a software product. We will consider activities 1 through 5 in the rest of this chapter; activities 6 through 8 will be the subject of the next chapter.

Because our case studies are relatively small compared to a commercial software project, we generally follow the simple waterfall model. However, we do revisit some case study solutions in later chapters and iterate through the five activities, much as in the Unified Model.

---

**FIGURE 1.2**
The Unified Software Life Cycle Model

### Requirements Specification

Because the potential users of a new system are often naïve as to what can be accomplished, the initial specification for a software product may be incomplete or ambiguous. The specification is clarified through extensive interaction between the users of the software and the system analyst. Through this interaction, the system analyst determines precisely what the users want the proposed software to do, and the users learn what to expect from the software product. This way there are no surprises in the end.

Although it may seem like common sense to proceed in this way, very often a software product does not perform as its users expected. The reason is usually a communication gap between those responsible for the product’s design and its eventual users; generally, both parties are at fault when the software fails to meet expectations. To avoid this possibility, it is imperative that a complete, written description of the requirements—a requirements specification—for a new software product be generated at the beginning of the project and that both users and designers review and approve the document.

The system analyst works with the software users to clarify the detailed system requirements. Some of the questions that need to be answered deal with the format of the input data, the desired form of any output screens or printed forms, and the need for data validation. You often need to mimic this process by interrogating your

<table>
<thead>
<tr>
<th></th>
<th><strong>Requirements specification</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td><strong>Requirements specification</strong> The requirements for the software product are determined and documented.</td>
</tr>
<tr>
<td>2.</td>
<td><strong>Architectural design</strong> The architecture of the solution is determined. This breaks the solution into different components, which are allocated to one or more processing resources.</td>
</tr>
<tr>
<td>3.</td>
<td><strong>Component design</strong> For each component, classes are identified, with specified roles and responsibilities.</td>
</tr>
<tr>
<td>4.</td>
<td><strong>Detailed design</strong> Functions and data fields are defined for classes. Detailed algorithms for the functions are defined.</td>
</tr>
<tr>
<td>5.</td>
<td><strong>Implementation</strong> The individual functions are coded in the target programming language.</td>
</tr>
<tr>
<td>6.</td>
<td><strong>Unit test</strong> Each class and its functions are tested individually.</td>
</tr>
<tr>
<td>7.</td>
<td><strong>Integration test</strong> Groups of classes are tested together to verify that they work together and meet the requirements.</td>
</tr>
<tr>
<td>8.</td>
<td><strong>Acceptance test</strong> The product as a whole is tested against its requirements to demonstrate that the product meets its requirements.</td>
</tr>
<tr>
<td>9.</td>
<td><strong>Installation</strong> The product is installed in its end-use (production) environment.</td>
</tr>
<tr>
<td>10.</td>
<td><strong>Maintenance</strong> Based upon experience with the software, enhancements and corrections are made to the product.</td>
</tr>
</tbody>
</table>
instructor or teaching assistant to determine the precise details of a programming assignment.

For example, assume that your instructor has given you the following incomplete specification of a programming assignment.

Problem: Write an interactive telephone directory program that will contain a collection of names and telephone numbers. You should be able to insert new entries in the directory, retrieve an entry in the directory, or change a directory entry.

Some of the questions that come to mind and might require clarification are the following:

- Is there an initial list of names and numbers to be stored in the directory beforehand, or are all entries inserted at the same time?
- If there is an initial list, is it stored in a file, or should it be entered interactively?
- If the initial directory is stored in a file, is the file a text file (file of characters) or a binary file (file of binary values)?
- If the file is a text file, are there any formatting conventions (for example, the name starts in column 1 and the phone number starts in column 20). Are the name and number on the same data line or on separate lines? How are the names stored (for example, last, first or first last)?
- Is it possible for there to be more than one telephone number associated with a particular name? If so, should the first number be retrieved, the last number, or all numbers?
- Is it possible to change a person’s name as well as the person’s phone number?
- When a number is retrieved, should both the person’s name and number be displayed or just the number? What form should this display take?
- What action should be taken if a “new” entry has the same name as a person already in the directory? Should this be flagged as an error?

As you can see, there are plenty of questions left unanswered by the initial problem statement. To complete the requirements specification, you should answer these questions and more. Many of the questions deal with details of input data, handling of potential errors in input data, and formats of input data and output lists.

Analysis

Once the system requirements are specified, the analysis stage begins. Before you can embark on the design of a program solution, you should make sure that you completely understand the problem. If the requirements specification has been carefully done, this will be easy. If there are any questions remaining, they should be cleared up at this time.

The next step is to evaluate different approaches to the program design. In industry, the system analyst and users may consider whether there are commercial software packages that can be purchased to satisfy their requirements (as an alternative to developing the software in-house). They must also determine the impact of the new software product on existing computer systems and what new hardware or software
will be needed to develop and run the new system. They determine the feasibility of each approach by estimating its cost and anticipated benefits. The analysis stage culminates with the selection of what appears to be the best design approach.

In your coursework, you do not have this flexibility. You must design and implement each program. During the analysis, your goal is to carefully determine the input/output requirements for the system and its interaction with the user. You will also find it helpful to break your system up into a set of small and manageable components, which you can then design and code separately. To do so, you need to identify the modules or components that will constitute the system and specify the interactions between them. You complete this process in the design phase, which we discuss next.

**Design**

Once you understand the problem and have selected the overall approach, it is time to develop a high-level design of the system. Professional software engineers make use of several design approaches in their work. The top-down approach, in which a system is broken into a set of smaller subsystems, each of those subsystems is broken into smaller components, and so forth until components are small and simple enough to be coded easily, has been used successfully for many years. More recently, the object-oriented approach, in which the developer identifies a set of objects and specifies their interaction, has been increasingly used. In this text we will utilize both the top-down and object-oriented approaches to software design.

**Top-Down Design**

The *top-down* approach to software design (also called *stepwise refinement*) instructs you to start at the top level (the original problem) and divide it into subproblems. For each subproblem, identify a subsystem with the responsibility of solving that subproblem. You can use a *structure chart* to indicate the relationship between the subproblems (and subsystems). For example, a structure chart for the telephone directory program is shown in Figure 1.3.

Figure 1.3 shows the two top levels of the structure chart, which include the original problem and its major subproblems. Each of these subproblems may be further refined and divided into still smaller subproblems. Figure 1.4 shows that to solve the subproblem “Read the initial directory” you must be able to “Read an entry from file” and “Store an entry in the directory”. Figure 1.5 shows that to solve the subproblem “Retrieve and display an entry” you must be able to “read a name”, “find
Figure 1.3 indicates that you can solve the original problem (level 0) by providing solutions to four level 1 subproblems. Figures 1.4 and 1.5 represent the solutions to two level 1 subproblems in terms of six level 2 subproblems.

Object-Oriented Design

Top-down design is process-oriented in that it focuses on the actions that are needed rather than the data structures. In contrast, object-oriented design (OOD) focuses on the data elements that are needed and operations to be performed on those data elements. In OOD, you first identify the objects that participate in your problem, and then you identify how they interact to form a solution. The common characteristics of a collection of similar objects define a class, and the interactions are identified as messages that one object sends to another. For an object to receive a message, its class must provide an operator (usually a function) to process that message. Looking at the nouns in the problem statement can help you identify objects, and looking at the verbs can point to the operators. We will discuss how to write C++ classes later in this chapter.

In looking at the phone directory problem statement, we have ascertained that there is a directory and that the directory contains entries. An entry consists of a name and a number. There is also a user, and there is a data file that contains entries. The user is external to the program and is called an actor in object-oriented terminology. The user selects the operations that will be performed on the directory and data file.

Object-oriented design incorporates some of the elements of top-down design. It also incorporates some of the elements of bottom-up design, which focuses on the design of individual system components. So OOD is a combination of both of these earlier techniques.
UML as a Design Tool

In this textbook we will use Unified Modeling Language™ (UML) diagrams as a design tool to illustrate the interaction between classes and between classes and external entities (users). We will introduce several features of UML in this textbook on an as-needed basis. Appendix B provides a summary of UML diagrams.

Figure 1.6 is a UML class diagram for the telephone directory program, showing the classes that are used and their interaction and relationships. The classes (Directory, Entry, and File) are identified by rectangles, whereas lines show the interaction and relationships. The lines ending with an arrow indicate an interaction between classes. For example, the class Directory has functions that read or write information from class File. External entities, called actors, are represented by small figures. This seems to imply that the external entity is a person, but that is not necessarily the case. So far, our class diagrams have no detail, because we have not yet identified their data elements and functions; we are focusing here on the interaction between classes.

UML diagrams are a standard means of documenting class relationships that is widely used in industry. We recommend that you use UML diagrams to describe the classes that you develop in solving your programming assignments.

EXERCISES FOR SECTION 1.1

SELF-CHECK

1. What are the advantages of the Unified Model over the waterfall model of the software life cycle?
2. What are the five activities in the software development life cycle followed in this book?
3. Name the four phases of the Unified Model. Name the five activities in the Unified Model.
4. Draw a structure chart showing the refinement of the subproblem “Insert a new entry”.
5. Draw a structure chart showing the refinement of the subproblem “Edit an entry”.

FIGURE 1.6
Initial Class Diagram for Phone Directory Program
1.2 Using Abstraction to Manage Complexity

Abstraction is a powerful technique that helps programmers (or problem solvers) deal with complex issues in a piecemeal fashion. An abstraction is a model of a physical entity or activity. In this book, we use abstraction to develop models of entities (objects and classes) and also of the operations performed on these objects. One example of abstraction is the use of a program variable (for example, name or number) to denote a storage location in memory for a data value. You need not be concerned with the details of the physical structure of memory or the actual bits (binary digits) that are used to represent the value of a variable; you don’t need to know those things to use variables in programming. This is analogous to driving a car. You need to know how to use a key to start the engine, how to use the accelerator and brake pedals to control speed, and how to use the steering wheel to control direction. However, you don’t need to know details of the car’s electrical system, drive train, braking system, or steering mechanism.

Procedural Abstraction

Procedural abstraction is the philosophy that procedure development should separate the concern of what is to be achieved by a procedure (a C++ function) from the details of how it is to be achieved. In other words, you can specify what you expect a function to do, then use that function in the design of a problem solution before you know how to implement the function. As an example of procedural abstraction, suppose you have functions available to perform all the Level 2 steps in Figure 1.5. You can then write the following C++ fragment to retrieve an entry from the directory using these functions.

```cpp
name = read_name(); // Reads a name
number = directory.get(name); // Gets number associated with name
if (number != "")
    // Display name and number.
    cout << "Phone number for " << name
    << " is " << number << endl;
else
    // Display message that name is not in directory.
    cout << name << " is not in directory\n";
```

Data Abstraction

In this course we will also use another type of abstraction: data abstraction. The idea of data abstraction is to specify the data objects for a problem and the operations to be performed on these data objects without being overly concerned with how they (the data objects) will be represented and stored in memory. You can describe what information is stored in the data object without being specific as to how the information is organized and represented. This is the logical view of the data object, as opposed to the physical view, its actual internal representation in memory. Once you understand the logical view, you can use the data object and its operators in your programs; however, you (or someone else) will eventually have to implement the data object and its operators before you can run any program that
uses them. In C++, the operators are called *member functions*, so we will use these terms interchangeably.

As an example, you have already practiced data abstraction in that you have used the `string` data type to represent sequences of characters without knowing how a character sequence is actually stored in memory. The C++ `string` is an abstraction for a sequence of characters. You can use `string` objects and their operators (length, at, and so on) without knowing the details of their implementation.

**Information Hiding**

One advantage of procedural abstraction and data abstraction is that they enable the designer to make implementation decisions in a piecemeal fashion. The designer can postpone making decisions regarding the actual internal representation of the data objects and the implementation of its operators. At the top levels of the design the designer focuses on how to use a data object and its operators; at the lower levels of design the designer (or perhaps a different designer or programmer on the team) works out the implementation details. In this way, the designer can control or reduce the overall complexity of the problem.

If the details of a data object’s implementation are not known when the higher-level module (a C++ class) is designed, the higher-level class can access the data object only through its member functions. From a software design viewpoint, this is an advantage rather than a limitation. It allows the designer of each class to make changes at a later date, such as to accommodate new government regulations or to implement a function in a more efficient way. If the higher-level classes reference a data object only through its functions, the higher-level class will not have to be rewritten. The process of “hiding” the details of a class’s implementation from users of the class is called *information hiding*.

As an example of how information hiding works, let’s see how you might access the name part of a new entry (called `my_entry`) for the phone directory. If you assume that `my_entry` is an object with a field called `name`, you could use the qualified identifier

```cpp
my_entry.name
```

As implementation proceeds, you might change your mind and decide to use an array data to hold each entry’s name and telephone number instead of separate data fields. In this case, you would have to go back and change the preceding reference (and all similar references) to

```cpp
my_entry.data[0]
```

It is much cleaner to hide the structure of an entry and instead use a function to retrieve the name string. If `get_name` is a function that extracts the name string from an `Entry` object, the statement

```cpp
a_name = my_entry.get_name();
```

will return the name string stored in `my_entry` and assign it to `a_name` regardless of the internal representation chosen for an entry. If you decide to change the internal representation, you only have to change the body of function `get_name`. You will not need to change any of the higher-level modules that call `get_name`. 
The rest of this chapter assumes that you are familiar with array processing and I/O fundamentals in C++. If you are not, you can learn about them in the first chapter, *A C++ Primer*. We next discuss C++ classes and show you how to define your own classes.

EXERCISES FOR SECTION 1.2

SELF-CHECK

1. How is information hiding related to data abstraction?
2. How does information hiding relate to procedural abstraction?
3. How does the logical view of a data object differ from its physical view?
4. Explain why type `int` is an abstraction for integers in mathematics.

1.3 Defining C++ Classes

A C++ program is a collection of functions and classes; consequently, when you write a C++ program, you will develop one or more classes. We will show you how to write a C++ class next.

A class represents a set of objects that have common properties. A class can be thought of as a template for specifying or creating objects. A class also represents a type. As discussed in *A C++ Primer*, a type determines what set of values an item of data can have and the operations that can be performed on it. In C++ there are two kinds of types: the primitive or built-in types and the user-defined or class types.

To define a class completely, we must provide two things: a *class definition* and a *class implementation*. The class definition describes the class to its users—the names of the operations and the internal data contents. The class implementation provides the implementation of the operations.

Class Definition

A class consists of members (not to be confused with the objects, or instances, of the class). Members can be either *data members* (also called data fields or attributes) or member functions (also called operators, functions, or methods). Data members are sometimes called *instance variables* because each class instance (object) has its own storage for them. The data members may be either primitive types or class types. As we discussed earlier, we want the operations to be visible to the users of the class, but we want to keep the implementation details hidden. Therefore, we specify that the member functions are *public* and the data members are *private*. Before getting into the formal syntax of a class definition, let us examine a simple example.
TABLE 1.3  
The Clock Member Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>void set_clock(int hr, int min, int sec)</td>
<td>Sets the clock to the specified values.</td>
</tr>
<tr>
<td>int get_hours()</td>
<td>Gets the number of hours since midnight or noon.</td>
</tr>
<tr>
<td>int get_minutes()</td>
<td>Gets the number of minutes in the current hour.</td>
</tr>
<tr>
<td>int get_seconds()</td>
<td>Gets the number of seconds in the current minute.</td>
</tr>
<tr>
<td>void tick()</td>
<td>Advances the clock by one second.</td>
</tr>
</tbody>
</table>

EXAMPLE 1.1  
The class Clock describes a group of objects, each of which represents the time using a 12-hour clock. Operations that we want to perform on Clock objects are setting the time, determining the value of hours, minutes, and seconds, and advancing the clock by one second. These operations are described in Table 1.3.

So that a class can be used by other classes or functions, which become the class’s clients, we place the class definition into a header file. Listing 1.1 shows the header file Clock.h for the class Clock.

LISTING 1.1  
Clock.h

```c++
#ifndef CLOCK_H_
#define CLOCK_H_

/** The class Clock represents the time of day. */
class Clock {

  public:
  /** The interface is defined here */
  /** Set the clock. */
  void set_clock(int hr, int min, int sec);

  /** Get the current hour. */
  int get_hours() const;

  /** Get the current minute. */
  int get_minutes() const;

  /** Get the current second. */
  int get_seconds() const;

  /** Advance the clock by one second. */
  void tick();

#endif
```
private:
    // The implementation details are defined here
};
#endif

This header file consists of only the member function declarations (the name and parameters). The implementation of these functions is provided separately—part of the hidden information. Notice that the definition for class Clock is enclosed in braces ({, }) and there is a semicolon after the closing brace.

Notice that the comments above each function declaration are introduced by /** rather than simply /* as described in the C++ Primer. The reason is discussed at the end of this section.

### Preprocessor Directives in Clock.h

The preprocessor directives #ifndef, #define, and #endif are used to prevent multiple definitions of the identifiers in file Clock.h, which could happen if it were included more than once (as when it is included in two different files that are both included by a third file). The first directive,

    ifndef CLOCK_H_

tests to see whether the identifier CLOCK_H_ is not defined. If it is defined, the preprocessor skips the lines in the source file through the directive

    endif

If CLOCK_H_ is not defined, the preprocessor reads the second directive,

    define CLOCK_H_

and it defines the identifier CLOCK_H_, giving it the value of the empty string. Consequently, this directive and the lines that constitute the definition for class Clock will be processed just one time, so the identifiers declared in this class cannot be multiply defined.

The identifier, CLOCK_H_, used in these directives was chosen arbitrarily; any identifier can be used. However, it is a common convention to use the file name in uppercase, replacing the period with an underscore and appending a trailing underscore.

### The public and private Parts

The keywords public and private are known as access specifiers. Members declared in the public part of the class definition can be accessed by any other class. Members declared in the private part can be accessed directly only within class Clock. So far, our class definition is incomplete, because we have shown only the public part. Although the implementation details are hidden within the private part of the class definition, we still must provide them. Thus, when we talk about information hiding, we do not mean that the client programmer (that is, the programmer who uses this class) cannot see this information; we mean that the client programmer cannot take advantage of this information, because this data is not directly accessible in the client program.
In our Clock class we have at least two choices for how to store the time. We can provide data fields for the hours, minutes, and seconds separately, or we can provide a data field that contains the number of seconds since midnight. We will choose the former because it simplifies the implementation of the member functions, except for tick. Thus, after the keyword `private` in Listing 1.1, we would have the following:

```cpp
/** The hours since midnight or noon. */
int hours;
/** The minutes within the current hour. */
int minutes;
/** The seconds within the current minute. */
int seconds;
```

**SYNTAX**  
**Class Definition**

**FORM:**
```cpp
class name {
    public:
        member declarations
    private:
        member declarations
};
```

**EXAMPLE:**
See Listing 1.1.

**MEANING:**
The class `name` is defined to have the members listed. Members may be either data members (also called data fields), member functions (also called functions), or nested classes. Members that are declared following the access specifier `public:` are accessible by functions outside of the class, while those following the access specifier `private:` are accessible only by functions that are members of the class or are declared to be friends of the class. There may be multiple public and private regions, but generally all public members are declared together, followed by all private members. Also, declarations that precede the first access specifier are considered to be private.

**Private Data Fields, Public Functions**
The access specifier `private` sets the visibility of each declaration that follows it to private visibility. This means that whatever is declared after `private:` can be accessed only within the class definition. All the data members for the Clock class are private. Ordinarily, only class members with public visibility can be accessed outside of the class. The access specifier `public` sets the visibility of each declaration that follows it to public. In Listing 1.1, all the members declared after `public:` and before `private:` are member functions.
The reason for having private visibility for data fields is to control access to an object’s data and to prevent improper use and processing of an object’s data. If a data field is private, it can be processed outside of the class only by invoking one of the public member functions that are part of the class. Therefore, the programmer who writes the public member functions controls how the data field is processed. Also, the details of how the private data is represented and stored can be changed at a later time by the programmer who implements the class; the other programs that use the class (called the class’s clients) will not need to be changed.

Constant Member Functions

The declarations for the functions get_hours, get_minutes, and get_seconds in Listing 1.1 each have the keyword const immediately preceding the semicolon. This tells the users and the compiler that these functions will not modify the object on which they operate. This serves three purposes: First, it documents for the user that calling this function will not change its value, and second, it permits the compiler to incorporate certain optimizations when it generates the machine language code. Also, if you declare a member function to be const, the compiler will enforce this intention by flagging as an error any operation that could potentially modify the object. Therefore, you should declare as constant a member function that is not supposed to modify an object.

Class Implementation

Member function definitions are the same as regular function definitions except that the class name is placed in front of the function name, separated from the name by a pair of colons. Thus, we can implement the get_hours function in our clock class as

```cpp
int Clock::get_hours() const {
    return hours;
}
```

### SYNTAX

**Member Function Definition**

**FORM:**

```
return-type class-name::function-name(parameter list) const_opt { body }
```

**EXAMPLE:**

```cpp
int Clock::get_hours() const { return hours; }
```

**MEANING:**

The function `function-name` that is a member of the class `class-name` is defined by the sequence of statements provided in the `body`. The declaration portion of the definition must be the same as that given in the class definition with respect to the return type, function name, types of parameters, and `const` specifier. The subscript `opt` following the `const` specifier indicates that it is optional. However, if it is present in the function declaration, it must also be present in the corresponding function definition. If these do not match, the compiler will report an error such as “`function-name` is not a member of `class-name`.”
Usually all of the member function definitions for a class are grouped into a single source (.cpp) file. Listing 1.2 shows the implementation for the Clock class. The source file must begin with the line

```
#include "Clock.h"
```

which includes the header file for class Clock, thereby defining the class members. Notice that the header file name is surrounded by quotes, not angle brackets (<>). Angle brackets are used only with system-defined classes.

Listing 1.2
Clock.cpp

// Implementation of the Clock class.
#include "Clock.h"

/** Set the clock. */
void Clock::set_clock(int hr, int min, int sec) {
    hours = hr;
    minutes = min;
    seconds = sec;
}

/** Get the current hour. */
int Clock::get_hours() const {
    return hours;
}

/** Get the current minute. */
int Clock::get_minutes() const {
    return minutes;
}

/** Get the current second. */
int Clock::get_seconds() const {
    return seconds;
}

/** Advance the clock by one second. */
void Clock::tick() {
    seconds++;
    if (seconds >= 60) {
        minutes++;
        seconds -= 60;
    }
    if (minutes >= 60) {
        hours++;
        minutes -= 60;
    }
    if (hours > 12) {
        hours -= 12;
    }
}
Using Class Clock

Let’s consider how class Clock might be used. The program in Listing 1.3 first includes the header file for class Clock. The main function begins by declaring an object \( c \) of type Clock. The remaining statements use dot notation to apply one of the Clock member functions to object \( c \). First, the clock is set to 12:59:58 (12 hours, 59 minutes, and 58 seconds). Next, the clock is advanced by four seconds, and the updated time is displayed. The output line displayed should be

\[
\text{The new time is } 01:00:02
\]

```
LISTING 1.3
Test_Clock.cpp

/** Program to test the Clock class. */
#include "Clock.h"
#include <iostream>
#include <iomanip>
using namespace std;

int main() {
    Clock c;
    c.set_clock(12, 59, 58);
    c.tick();
    c.tick();
    c.tick();
    c.tick();
    cout << setfill('0');
    cout << "The new time is "
         << setw(2) << c.get_hours() << ':'
         << setw(2) << c.get_minutes() << ':'
         << setw(2) << c.get_seconds() << '\n';
    return 0;
}
```

The Class Person

We now consider a more complicated example of a class definition and implementation. A class Person might describe a group of objects, each of which is a particular human being. For example, instances of class Person would be yourself, your mother, and your father. A Person object could store the following data:

- Given name
- Family name
- ID number
- Year of birth

The following are a few of the operations that can be performed on a Person object:

- Calculate the person’s age
- Test whether two Person objects refer to the same person
- Determine whether the person is old enough to vote
• Determine whether the person is a senior citizen
• Get one or more of the data fields for the Person object
• Set one or more of the data fields for the Person object

Figure 1.7 shows a diagram of class Person. This figure uses the Unified Modeling Language (UML) to represent the class. UML diagrams are a standard way of documenting class relationships that is widely used in industry. The class is represented by a box. The top compartment of the box contains the class name. The data fields (data members or instance variables) are shown in the middle compartment, and some of the functions are shown in the bottom compartment. We discuss UML further in Appendix B.

Figure 1.8 shows how two objects or instances of the class Person (author1 and author2) are represented in UML. Each object is represented by a box in which the top compartment contains the class name (Person), underlined, followed by the object name (author1 or author2), also underlined, and the bottom compartment contains the data fields and their values.

Listing 1.4 shows class Person and the member functions for this class.

We declare four data fields and two constants (all uppercase letters) after the functions (although some C++ programmers prefer to declare member functions after data fields). In the constant declarations, the modifier const indicates that the constant value may not be changed. The modifier static indicates that the constant is being defined for the class and does not have to be replicated in each instance. In other words, storage for the constant VOTE_AGE is allocated once, regardless of how many instances of Person are created.

The file Person.h contains the class declaration. As with the Clock class, the member functions are declared in this file, but, in several cases, they are also implemented. We discuss the reason for this next.
LISTING 1.4
Person.h

#ifndef PERSON_H_
#define PERSON_H_
#include <string>
#include <iostream>

/** Person is a class that represents a human being. */
class Person {
    public:
        // Constructors
        /** Construct a person with given values.
         * @param first The given name
         * @param family The family name
         * @param ID The ID number
         * @param birth The birth year
         */
        Person(std::string first, std::string family,
                 std::string ID, int birth) :
            given_name(first), family_name(family), ID_number(ID),
            birth_year(birth) { }

        /** Construct a default person. */
        Person() : given_name(""), family_name(""), ID_number(""),
                    birth_year(1900) { }

        // Modifier Functions
        /** Sets the given_name field.
         * @param given The given name
         */
        void set_given_name(std::string given) {
            given_name = given;
        }

        /** Sets the family_name field.
         * @param family The family name
         */
        void set_family_name(std::string family) {
            family_name = family;
        }

        /** Sets the birth_year field.
         * @param birth The year of birth
         */
        void set_birth_year(int birth) {
            birth_year = birth;
        }

        // Accessor Functions
        /** Gets the person's given name.
         * @return the given name as a string
         */
        std::string get_given_name() const { return given_name; }
};

1.3 Defining C++ Classes
/** Gets the person's family name. 
 * @return the family name as a string 
 */
std::string get_family_name() const { return family_name; }

/** Gets the person's ID number. 
 * @return the ID number as a string 
 */
std::string get_ID_number() const { return ID_number; }

/** Gets the person's year of birth. 
 * @return the year of birth as an int value 
 */
int get_birth_year() const { return birth_year; }

// Other Functions
/** Calculates a person's age at this year's birthday. 
 * @param year The current year 
 * @return the year minus the birth year 
 */
int age(int year) const;

/** Determines whether a person can vote. 
 * @param year The current year 
 * @return true if the person's age is greater than or 
 *         equal to the voting age 
 */
bool can_vote(int year) const;

/** Determines whether a person is a senior citizen. 
 * @param year the current year 
 * @return true if person's age is greater than or 
 *         equal to the age at which a person is 
 *         considered to be a senior citizen 
 */
bool is_senior(int year) const;

/** Compares two Person objects for equality. 
 * @param per The second Person object 
 * @return true if the Person objects have the same 
 *         ID number; false if they don't 
 */
bool operator==(const Person& per) const;

/** Compares two Person objects for inequality. 
 * @param per The second Person object 
 * @return the negation of the equals operator 
 */
bool operator!=(const Person& per) const;

/** Declaration of the stream insertion operator for Person. 
 * @param os The target ostream 
 * @param per The Person object being output 
 * @return The updated output stream 
 */
friend std::ostream& operator<<(std::ostream& os, const Person& per);
private:
// Data Fields
/** The given name. */
std::string given_name;
/** The family name. */
std::string family_name;
/** The ID number. */
std::string ID_number;
/** The birth year. */
int birth_year;

// Constants
/** The age at which a person can vote. */
static const int VOTE_AGE = 18;
/** The age at which a person is considered a senior citizen. */
static const int SENIOR_AGE = 65;
};
#endif

---

### Program Style

### Explicit Namespace Qualification in Include Files

You will notice that the include file included the standard headers `<string>` and `<iostream>`, but it did not contain a `using` directive such as `using namespace std;` Instead we specifically qualified the classes `string` and `ostream` with the prefix `std::`. This is because we do not know whether the client program will include the `using` directive. In Chapter 3 we will see reasons why this may not be desirable. It is considered good program style to always use explicit namespace qualification in header files.

### Constructors

In Listing 1.4, there are two constructors that begin with `Person`. Exactly one of these is invoked automatically when a new class instance is created. The constructor with four parameters is called if the values of all data fields are known before the object is created. For example, the statement

```cpp
Person author1("Elliot", "Koffman", "010-55-0123", 1942);
```

creates the first object shown in Figure 1.8, initializing its data fields to the values passed as arguments. Note that a constructor differs from a member function in that it has no return type.
**SYNTAX**

**Constructor Definition**

FORM:

```
class-name(parameters) : initialization-expressions { body }
```

EXAMPLES:

```
Person(std::string first, std::string family, std::string ID, int birth) :
  given_name(first), family_name(family), ID_number(ID),
  birth_year(birth) { }
Person() : given_name("") , family_name("") , ID_number("") ,
  birth_year(1900) { }
```

MEANING:

A constructor has the same name as its class and has no return type. A constructor is called automatically when an object is created. Its purpose is to initialize the memory allocated to the object to initial values. Constructors may have parameters, and constructors having different parameter lists may be defined. The *initialization-expressions* are of the form *data-field-name*(*initial-value*). As the object is constructed, the named data fields are set to the given initial values. After the object is constructed, the body of the constructor is then executed.

**PROGRAM STYLE**

**Use of Initialization Expressions in Constructors**

It is preferable to use initialization expressions to initialize data fields rather than using assignment statements in the body. Initialization expressions have the form

```
data-member(initial-value-expression)
```

An initialization expression such as

```
given_name(first)
```

is evaluated as the object is being constructed. If there is no initialization expression for a class data field, it may be initialized to a default value. The body is executed after the object is constructed. Thus, if you initialize a class data field in the body using an assignment statement

```
given_name = first;
```

you are effectively initializing it twice. While this does not matter for the built-in types because they have no default initialization, it is significant for class types, which may have several data fields of their own to initialize.
The second constructor, the no-parameter constructor, is called when the data field values are not known at the time the object is created.

    Person author2;

In this case, the data fields are initialized to the default values given (for example, 1900 for birth_year). The three string data fields are initialized to the empty string. You can use modifier functions (discussed next) at a later time to set or modify the values of the data fields.

The no-parameter constructor is sometimes called the *default constructor* because C++ automatically defines this constructor for a class that has no constructor definitions. Therefore, the following constructor was automatically defined for class Clock.

    /** Default constructor for class Clock. */
    public Clock::clock() { }

However, if you define one or more constructors for a class, you must also explicitly define the no-parameter constructor, or it will be undefined for that class.

**Modifier and Accessor Member Functions**

Because the data fields have private visibility, we need to provide public member functions to access them. Normally, we want to be able to get or retrieve the value of a data field, so each data field in class Person has an accessor member function (also called a *getter*) that begins with the word get and ends with the name of the data field (for example, get_family_name). If we want to allow a class user to update or modify the value of a data field, we provide a modifier function (also called a *mutator* or *setter*) beginning with the word set and ending with the name of the data field (for example, set_given_name). Currently, there is an accessor for each data field in this example. However, there is no set_ID_number member function because it would not be a good practice to allow a user program to modify a person's ID number. It might also be a good practice to have get_ID_number return just a portion of the value stored in data field ID_number (say, the last four digits) rather than the whole string.

The modifier member functions are type *void* because they are executed for their effect (to update a data field), not to return a value. In the member function set_birth_year,

    void set_birth_year(int birth) {
        birth_year = birth;
    }

the assignment statement stores the integer value passed as an argument in data field birth_year.

The accessor function for data field given_name,

    std::string get_given_name() const { return given_name; }

is type std::string because it returns a copy of the string object given_name. The *const* in the function declaration indicates that this function will not change the value of the object on which it is called.
Operators

In addition to the functions defined in class Person, we declare the operators == and !=.

```cpp
bool operator==(const Person& per) const;
bool operator!=(const Person& per) const;
```

Declaring and defining these operators in class Person enables us to use them with operands of type Person. For example, we can write the Boolean expressions author1 == author2 and author1 != author2 to compare two Person objects. The word operator precedes the operator symbol being defined. The left-hand operand is the current object; the right-hand operand is declared in parentheses following the operator symbol. We say that C++ permits us to overload these operators so that they can be used with operands other than the primitive types.

SYNTAX

Operator Declaration

FORM:

```cpp
return-type operator operator-symbol(parameter) const;
```

EXAMPLES:

```cpp
bool operator==(const Person& per) const;
bool operator!=(const Person& per) const;
```

MEANING:

The operator specified by `operator-symbol` is defined for this class. The left-hand operand is the current object; the right-hand operand is specified by `parameter` in the same way as a function parameter. If the word `const` appears at the end of the declaration, the left-hand operand is not changed by the operator.
**Friends**

The insertion operator that you have been using to output data to an `ostream`, such as `cout`, is really an overloading of the left shift operator. The class `ostream` contains several overloaded member functions named `operator<<` that each take a primitive type as the parameter. Each `operator<<` function formats its primitive type and outputs the resulting string to the `ostream`. To output a class type, we must define this operator for that class. Class `Person` has the following declaration:

```cpp
friend std::ostream& operator<<(std::ostream&, const Person&);
```

This function is not a member of either the class `ostream` or the class `Person`. It is a stand-alone function that takes as its left-hand operand an object of type `ostream` and as its right-hand operand an object of type `Person` (for example, `cout << author1`). It returns a reference to the output stream that has been modified. This declaration appears within the class definition prefixed with the keyword `friend`. Declaring it a `friend` gives it the same rights as member functions, so it can access the private members of the class.

**Implementing the Person Class**

The implementation file for the `Person` class (file `Person.cpp`) is given in Listing 1.5. We define all the functions that were declared in Listing 1.4 but were not written as inline functions.

**LISTING 1.5**

```
Person.cpp
/** Implementation file for the class Person. */
#include "Person.h"
#include <ostream>
using std::ostream;
/** Calculates a person's age at this year's birthday.
 @param year The current year
 @return the year minus the birth year */
int Person::age(int year) const {
    return year - birth_year;
}
/** Determines whether a person can vote.
 @param year The current year
 @return true if the person's age is greater than or equal to the voting age */
bool Person::can_vote(int year) const {
    int the_age = age(year);
    return the_age >= VOTE_AGE;
}
/** Determines whether a person is a senior citizen.
 @param year the current year
 @return true if person's age is greater than or equal to the age at which a person is considered to be a senior citizen
```
bool Person::is_senior(int year) const {
    return age(year) >= SENIOR_AGE;
}

bool Person::operator==(const Person& per) const {
    return ID_number == per.ID_number;
}

bool Person::operator!=(const Person& per) const {
    return !(this == per);
}

ostream& operator<<(ostream& os, const Person& per) {
    os << "Given name: " << per.given_name << '
'
        << "Family name: " << per.family_name << '
'
        << "ID number: " << per.ID_number << '
'
        << "Year of birth: " << per.birth_year << '
';
    return os;
}

Declarating Local Variables in Class Person

Functions age, can_vote, and is_senior are all passed the current year as an argument. Function can_vote calls member function age to determine the person’s age. The result is stored in local variable the_age. The result of calling function can_vote is the value of the Boolean expression following the keyword return.

bool Person::can_vote(int year) const {
    int the_age = age(year);
    return the_age >= VOTE_AGE;
}

It really was not necessary to introduce local variable the_age; the call to member function age could have been placed directly in the return statement (as it is in function is_senior). We wanted, however, to show you how to declare local variables in a C++ function. The scope of the local variable the_age and the parameter year is the body of member function can_vote.

Implementing the Operators

We will assume that two Persons are the same if they have the same ID number. This leads to the following definition for operator ==.
bool Person::operator==(const Person& per) const {
    return ID_number == per.ID_number;
}

Notice that we can look at parameter per’s private ID_number because per references an object of this class (type Person). Because ID_number is type string, the equality operator of class string is invoked with the ID_number of the second object as an argument. If the two ID_number data fields have the same contents, the string operator== function will return true; otherwise, it will return false. The Person operator== member function returns the result of the string operator== function.

Because we want operator!= to return the opposite of what operator== returns, we define it as follows:

```cpp
bool Person::operator!=(const Person& per) const {
    return !(operator==(per));
}
```

**The this Parameter**

Each member function has an implicit parameter named this whose value is a pointer to the object for which the member function was called. Normally in writing a member function you do not need to use the this parameter, because references to members are implicitly understood to be to the object for which the member function was called. However, in the case of operator! we want to apply the == operator between the object on which the function was called and the other object, per. Since this is a pointer to the object we are interested in, the expression *this is a reference to that object. Thus the expression *this == per does what we want. We could also have written operator!= this way:

```cpp
bool Person::operator!=(const Person& per) const {
    return !(operator==(per));
}
```

---

**Program Style**

**Returning a Boolean Value**

Some programmers unnecessarily write if statements to return a Boolean value. For example, instead of writing

```cpp
return ID_number == per.ID_number;
```

they write

```cpp
if (ID_number == per.ID_number)
    return true;
else
    return false;
```

Resist this temptation. The return statement by itself returns the value of the if statement condition, which must be true or false. It does this in a clear and succinct manner using one line instead of four.
Defining the Extraction Operator

In the following definition for `operator<<`, the function body outputs data field values to output stream `os`. It then returns a reference to its output stream parameter.

```cpp
ostream& operator<<(ostream& os, const Person& per) {
    os << "Given name: " << per.given_name << '
'
        << "Family name: " << per.family_name << '
'
        << "ID number: " << per.ID_number << '
'
        << "Year of birth: " << per.birth_year << '
';
    return os;
}
```

If `author1` is the first type `Person` object in Fig. 1.8, the statement

```cpp
cout << author1;
```

would display the following.

```
Given name: Elliot
Family name: Koffman
ID number: 010-55-0123
Year of birth: 1942
```

An Application That Uses Class Person

To test class `Person` we need to write a C++ application program that consists of a main function. The main function should create one or more instances of class `Person` and display the results of applying the class functions. Listing 1.6 shows the program `Test_Person` that does this. To execute the main function, you must compile `Test_Person.cpp` and `Person.cpp` and link them. Figure 1.9 shows a sample run.

A main function that is written primarily to test another class is often called a **driver program**. A test that demonstrates that certain requirements of a particular class are met is called a **unit test**. We will have more to say about testing in Chapter 2.
**Listing 1.6**

Test_Person.cpp

```cpp
/** Test_Person is an application that tests class Person. */
#include <iostream>
#include "Person.h"
using std::cout;
using std::endl;

int main() {
    Person p1("Sam", "Jones", "1234", 1930);
    cout << "p1: " << p1 << endl;
    Person p2("Jane", "Jones", "5678", 1990);
    cout << "p2: " << p2 << endl;
    cout << "Age of " << p1.get_given_name() << " is " << p1.age(2004) << endl;
    if (p1.is_senior(2004))
        cout << p1.get_given_name() << " can ride the subway for free\n";
    else
        cout << p1.get_given_name() << " must pay to ride the subway\n";
    if (p2.can_vote(2004))
        cout << p2.get_given_name() << " can vote\n";
    else
        cout << p2.get_given_name() << " can't vote\n";
    // Make Sam younger
    p1.set_birth_year(1950);
    // Now see whether he has to pay to ride the subway.
    cout << "Age of " << p1.get_given_name() << " is " << p1.age(2004) << endl;
    if (p1.is_senior(2004))
        cout << p1.get_given_name() << " can ride the subway for free\n";
    else
        cout << p1.get_given_name() << " must pay to ride the subway\n";
}
```

**Figure 1.9**

Sample Run of Class Test_Person

```
F:\C++Book\programs\CH01\Test_Person
p1: Given name: Sam
    Family name: Jones
    ID number: 1234
    Year of birth: 1930

p2: Given name: Jane
    Family name: Jones
    ID number: 5678
    Year of birth: 1990

Age of Sam is 74
Sam can ride the subway for free
Age of Jane is 54
Jane can vote
Age of Sam is 54
Sam must pay to ride the subway
```
Classes as Components of Other Classes

Class Person has three data fields that are type string, so string objects are components of a Person object. In Figure 1.10 this component relationship is indicated by the solid diamond symbol at the end of the lines drawn from the box representing class string to the box representing class Person. The names given_name, family_name, and ID_number above the lines and near the box representing the string class correspond to the data member names within the Person class. Figure 1.10 is a UML diagram, this one showing the relationships between classes. We will follow UML’s set of conventions for documenting class relationships in this book.

Array Data Fields

It is very common in C++ to encapsulate an array, together with the functions that process it, within a class. Rather than allocate storage for a fixed-size array, we would like the client to be able to specify the array size when an object is created. Therefore, we should define a constructor with the array size as a parameter and have the constructor allocate storage for the array. Class Company in Listing 1.7 has a data field employees that references an array of Person objects. Both constructors allocate storage for a new array when a Company object is created. The client of this class can specify the size of the array by passing a type int value to the constructor parameter size. If no argument is passed, the no-parameter constructor sets the array size to DEFAULT_SIZE.

Listing 1.7

Company.h

```cpp
#ifndef COMPANY_H_
#define COMPANY_H_

#include "Person.h"  // Definition of the Person class
#include <iostream>  // Declaration of I/O classes

/** Company is a class that represents a company. 
   The data field employees provides storage for 
   an array of Person objects. 
*/
class Company {
   public:
      // Constructors
      /** Creates a new array of Person objects. 
          @param size The size of array employees 
*/
      Company(int size) :
         num_employees(size), employees(new Person[size]) {}
};
```

Figure 1.10
UML Diagram Showing That string Objects Are Components of Class Person
Company() :
    num_employees(DEFAULT_SIZE), employees(new Person[DEFAULT_SIZE]) {} 

// Functions
/** Sets an element of employees.
   @param index The position of the employee
   @param emp The employee
*/
void set_employee(int index, Person emp);

/** Gets an employee.
   @param index The position of the employee
   @return the employee object or if not defined
       a default Person object
*/
Person get_employee(int index);

/** Builds and outputs a string consisting of all employees'
data, with newline characters between employees.
   @param os The destination ostream
   @param company The company object to be output
   @return The updated ostream object
*/
friend std::ostream& operator<<(std::ostream& os,
    const Company& company);

private:
    // Data Fields
    /** The number of employees. */
    int num_employees;
    /** The array of employees. */
    Person* employees;

    /** The default size of the array. */
    static const int DEFAULT_SIZE = 100;
};

#endif

LISTING 1.8
Company.cpp
#include "Company.h"
#include <ostream>
using std::ostream;
using std::endl;

/** Sets an element of employees.
   @param index The position of the employee
   @param emp The employee
*/
void Company::set_employee(int index, Person emp) {
    if (index >= 0 && index < num_employees) 
        employees[index] = emp;
}
/** Gets an employee. 
@param index The position of the employee
@return The employee object or if not defined return a default Person object */
Person Company::get_employee(int index) {
    if (index >= 0 && index < num_employees)
        return employees[index];
    else
        return Person();
}

/** Builds and inserts a string consisting of all employees' 
data, with newline characters between employees. 
@param os The destination ostream
@param company The Company object to be output
@return The updated ostream */
ostream& operator<<(ostream& os, const Company& company) {
    for (int i = 0; i < company.num_employees; i++)
        os << company.employees[i] << endl;
    return os;
}

There are modifier and accessor functions that process individual elements of array 
Company (set_employee and get_employee). Function get_employee returns the type 
Person object at position index, or a default Person object if the value of index is 
out of bounds.

The ostream insertion operator << creates a string representing the contents of array 
employees. The for loop calls the ostream insertion operator for each Person object 
in the array.

The following main function illustrates the use of class Company and displays the state 
of object comp. It stores two Person objects in the array and then displays the con- 
tents of the array.

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

int main() {
    Company comp(2);
    comp.set_employee(0, Person("Elliot", "K", "123", 1942));
    comp.set_employee(1, Person("Paul", "W", "234", 1945));
    cout << comp;
    return 0;
}

**Documentation Style for Classes and Functions**

In this book we write documentation comments for classes, member functions, and 
data members in a standard format that was originally developed for the Java lan- 
guage. The Java software developer's kit contains a program called Javadoc that reads 
a Java program file and writes neatly formatted HTML documentation pages if the 
program contains documentation comments in the standard form. Using Javadoc
forces the programmer to describe the function in an orderly manner. As a result, that format has gained popularity for documenting programs in other languages, such as C++, as well. You can run a variety of programs on your C++ files to generate a set of HTML pages describing each class and its data fields and functions as Javadoc does for Java. Two such programs that are freely available are DOC++ and Doxygen.

These programs focus on text that is enclosed within the delimiters /** and */. The introductory comment that describes the class is displayed on the HTML page exactly as it is written, so you should write that carefully. The lines that begin with the symbol @ are tags. They are described in Table 1.4. We will use one @param tag for each function parameter. We will not use a @return tag for void functions. The first line of the comment for each function appears in the function summary part of the HTML page. The information provided in the tags will appear in the function detail part. Figure 1.11 shows part of the documentation generated by running Doxygen for class Person.

<table>
<thead>
<tr>
<th>Tag and Example of Use</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>@param given The given name</td>
<td>Identifies a function parameter</td>
</tr>
<tr>
<td>@return The person's age</td>
<td>Identifies a function return value</td>
</tr>
</tbody>
</table>

FIGURE 1.11
Generated Documentation for Class Person
EXERCISES FOR SECTION 1.3

SELF-CHECK

1. Explain why member functions have public visibility but data fields have private visibility.

2. Trace the execution of the following statements.
   ```
   Person p1("Adam", "Jones", "wxyz", 0);
   p1.set_birth_year(1990);
   Person p2;
   p2.set_given_name("Eve");
   p2.set_family_name(p1.get_family_name());
   p2.set_birth_year(p1.get_birth_year() + 10);
   if (p1 == p2)
       cout << p1 << " is the same person as " << p2 << endl;
   else
       cout << p1 << " is not the same person as " << p2 << endl;
   ```

PROGRAMMING

1. Overload the `ostream` insertion operator for the `Clock` class to output the time in the form `hh:mm:ss`.

2. Write a member function `get_initials` that returns a string representing a `Person` object’s initials. There should be a period after each initial. Write documentation comments for the function.

3. Add a data field `mother_maiden_name` to `Person`. Write an accessor and a modifier function for this data field. Modify the `ostream` insertion operator and the equality operator to include this data field. Assume two `Person` objects are equal if they have the same ID number and mother’s maiden name.

4. Write the member functions `operator<`, `operator>`, `operator<=`, and `operator>=` that compare two `Person` objects and return an appropriate result based on a comparison of the ID numbers. For `operator<`, if the ID number of the object that the operator is applied to is less than the ID number of the argument object, the result should be true. Write documentation comments for the member functions.

5. Write a member function `switch_names` that exchanges a `Person` object’s given and family names. Write documentation comments for the member function.

1.4 Abstract Data Types, Interfaces, and Pre- and Postconditions

One of the goals of object-oriented programming is to write reusable code, which is code that can be reused in many different applications, preferably without having to be recompiled. One way to make code reusable is to encapsulate or combine a group of data elements representing an object together with its operations (functions and operators) in a separate program module (a class). As we discussed in the previous section, a new program can use the functions to manipulate the data without being
concerned about details of the data representation and the implementation of its operations. In this way, the class can be used as a building block to construct new application programs. The combination of data together with its operations is called an abstract data type (ADT).

Figure 1.12 shows a diagram of an abstract data type. The data values stored in the ADT are hidden inside the circular wall. The bricks around this wall are used to indicate that these data values cannot be accessed except by going through the ADT’s operations.

A class provides one way to implement an ADT in C++. If the data fields are private, they can be accessed only through public functions. Therefore, the functions control access to the data and determine the manner in which the data is manipulated.

A primary goal of this text is to show you how to write and use ADTs in programming. As you progress through this book, you will create a large collection of ADT implementations (classes) in your own program library. Because each ADT implementation in your library will already have been coded, debugged, and tested, using them would make it much easier for you to design and implement new application programs.

Also, the C++ Standard Library provides a rich collection of ADT implementations. You will be introduced to many of these ADTs and their implementations as you progress through this book. Many of the classes you study will implement these ADTs in whole or in part.

Abstract Data Types (ADTs) and Interfaces

The public part of a class definition is a way to specify (but not implement) an ADT. The public part of a class definition specifies the names, parameters, and return values of the ADT operations without specifying how the operations are performed and without specifying how the data is internally represented. The public part of a class definition defines the interface for that class. (If you are familiar with the Java programming language, take note that C++ does not have Java’s syntactic element called an interface).

The classes that declare the data fields and code member functions that perform the operations are said to implement the ADT. There may be more than one way to implement the operations; hence, there may be more than one class that implements the ADT. Therefore, an ADT describes a set of classes that perform the operations and define the internal data.

Each class that implements an ADT must provide the complete definition (implementation) of all operations declared in the ADT. In addition, it may declare data fields and define other member functions.

An ADT for a Telephone Directory Class

There are many different ways to represent a telephone directory. One of our goals in this course is to show you several ways to do this task and to compare their performance. Consequently, rather than provide a single class of type Phone_Directory, we will define an ADT that specifies the operations required of all classes that are
intended to implement this ADT. This will enable other programmers (including ourselves) to invoke the telephone directory operations without knowing exactly how the directory is represented or how its operations are implemented. We describe this ADT in Table 1.5.

### Contracts and ADTs

An ADT is a contract between the ADT designer and the programmer who codes a class that implements the ADT. This programmer must code functions that perform the operations specified in the ADT. Therefore, any programmer who uses a class that implements the ADT knows exactly what functions are available in that class and what operations they will perform. This allows that programmer to proceed to write application programs without needing to coordinate with the person who is coding classes that implement the ADT. Similarly, the programmer who is coding a class that implements the ADT can proceed independently of what the application programmers are doing.

There may be several classes that implement the ADT, and each class can code the data and the functions in a different way. One class may be more efficient than another class at performing certain kinds of operations. For example, one class may be more efficient at retrieving information from a directory, so that class will be used if retrieval operations are more likely in a particular application.

### Preconditions and Postconditions

At a lower level, we communicate the intent of an operation through its preconditions and postconditions, which are part of the behavior description provided in the specification. A precondition is a statement of any assumptions or constraints on the operation’s data (input parameters) that are expected to apply before the operation is performed. A postcondition is a statement that describes the result of performing

---

**Table 1.5**

Specification of the Phone_Directory ADT

<table>
<thead>
<tr>
<th>Function</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void load_data(const string&amp; source_name)</code></td>
<td>Load the data file containing the directory, or establish a connection with the data source.</td>
</tr>
<tr>
<td><code>string lookup_entry(const string&amp; name)</code></td>
<td>Look up an entry in the directory. If there is no such entry, return an empty string.</td>
</tr>
<tr>
<td><code>string add_or_change_entry(const string&amp; name, const string&amp; number)</code></td>
<td>Changes the number associated with the given name to the new value. Returns the previous value or an empty string if this is a new entry.</td>
</tr>
<tr>
<td><code>string remove_entry(const string&amp; name)</code></td>
<td>Removes the entry with the specified name from the directory. Returns the name or an empty string if the name was not in the directory.</td>
</tr>
<tr>
<td><code>void save()</code></td>
<td>Writes the directory to the data file.</td>
</tr>
</tbody>
</table>
an operation. An operation’s preconditions and postconditions serve as a contract between a function caller (who requires the operation to be performed) and the function programmer (who implements the operation); if a caller satisfies the precondition, the function result will satisfy the postcondition. If the precondition is not satisfied, there is no guarantee that the function will do what is expected, and it may even fail. We discuss this further in Chapter 2. The preconditions and postconditions allow a function user and function implementor both to proceed without further coordination. When implementing an ADT, we document the preconditions and postconditions in the multiline documentation comments associated with the class and each function.

Although some programmers write preconditions and postconditions for every function that they write, we will use them only when they provide additional information that is not readily apparent. If an operation has specific requirements for its arguments (for example, an argument must be positive), we will express these requirements through a precondition. We will use postconditions to describe the change in object state caused by executing a mutator operation. As a general rule, you should write a postcondition comment for operations that change an object’s state. (In C++ these operations are generally implemented by functions that have a `void` return type.) If an operation returns a value, you do not usually need a postcondition because the `@return` tag describes the effect of performing the operation.

**EXAMPLE 1.2** A class `Bank_Account` might define the following function `process_deposit`. The precondition (after `pre:`) shows that the deposit amount must be positive. The postcondition (after `post:`) shows that data field `balance` is increased by the value of `amount`. The words `pre:` and `post:` are not part of the language recognized by Javadoc or Doxygen, so these comments are not processed by the comment-processing program.

```java
/** Processes a deposit in a bank account.
 pre: amount is positive.
 post: Adds the specified amount to balance.
 */
 public void process_deposits(double amount) {
     balance = balance + amount;
 }
```

**EXERCISES FOR SECTION 1.4**

**SELF-CHECK**

1. What are the two parts of an ADT? Which part is accessible to a user and which is not? Explain the relationships between an ADT and a class.
2. What are two different uses of the term `interface` in programming?
**PROGRAMMING**

1. Write a documentation comment for the following member function of a class `Person`. Assume that class `Person` has two `string` data fields `last_name` and `first_name` with the obvious meanings. Provide preconditions and postconditions if needed.

   ```cpp
   bool operator<(const Person& per) {
       if (last_name == per.last_name)
           return first_name < per.last_name;
       else
           return last_name < per.last_name;
   }
   ```

2. Write a documentation comment for the following member function of a class `Person`. Provide preconditions and postconditions if needed.

   ```cpp
   void change_last_name(bool just_married, string new_last) {
       if (just_married)
           last_name = new_last;
   }
   ```

### 1.5 Requirements Analysis, Use Cases, and Sequence Diagrams

In this section, we will illustrate how to solve a programming problem similar to the telephone directory assignment introduced earlier. The solution will have multiple classes and interfaces. Our goal in this case study is to show you the process that would be followed in the software design and implementation. Don’t be concerned at this point if you do not understand all the details of the final program. In this section, we show the requirements specification and analysis and introduce two new tools: the use case and sequence diagram.

#### CASE STUDY  Designing a Telephone Directory Program

**Problem**

You have a client who wants to store a simple telephone directory in her computer that she can use for storage and retrieval of names and numbers. She has a data file that contains the names and numbers of her friends. She wants to be able to insert new names and numbers, change the number for an entry, and retrieve selected telephone numbers. She also wants to save any changes in her data file.

**Input/Output Requirements**

Earlier we discussed some questions that would have to be answered in order to complete the specification of the requirements for the phone directory problem. Most of the questions dealt with input and output considerations. We will list some answers to these questions next.
INPUTS
Initial phone directory Each name and number will be read from separate lines of a text file. The entries will be read in sequence until all entries are read.
Additional entries Each entry is typed by the user at the keyboard when requested.

OUTPUTS
Names and phone numbers The name and number of each person selected by the program user are displayed on separate output lines.
Updated phone directory Each name and number will be written to separate lines of a text file. The entries will be written in sequence until all entries are written.

Analysis The first step in the analysis is to study the problem input and output requirements carefully to make sure that they are understood and make sense. You can use a tool called a use case to help you refine the system requirements.

Use Cases
A use case is a list of the user actions and system responses for a particular sub-problem in the order that they are likely to occur.

The following four subproblems were identified for the telephone directory program:
- Read the initial directory from an existing file
- Insert a new entry
- Edit an existing entry
- Retrieve and display an entry

The use case (Table 1.6) for the first subproblem (“Read the initial directory”) shows that the user issues a single command and the system responds either by reading a directory from a file or by creating an empty directory if there is no file. The second use case (Table 1.7) is for the subproblems “Insert a new entry” and “Edit
Because the names in the directory must be unique, inserting a new entry and editing an existing entry require a search to determine whether the name is already present. Thus, from the user’s point of view, the insert and edit processes are the same. The last use case (Table 1.8) shows the user interaction for the last subproblem (“Retrieve and display an entry”).

The steps shown in each use case flesh out the user interaction with the program. The use cases should be reviewed by the client to make sure that your intentions are the same as hers. For most of the problems we study in this book, the user interaction is straightforward enough that use cases will not be required.

**Table 1.7**
Use Case for Inserting a New Entry or Editing an Existing Entry

<table>
<thead>
<tr>
<th>Step</th>
<th>User’s Action</th>
<th>System’s Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>User issues the command to insert or change an entry.</td>
<td></td>
</tr>
<tr>
<td>2.</td>
<td>System prompts for the name.</td>
<td></td>
</tr>
<tr>
<td>3.</td>
<td>User enters name.</td>
<td>If user cancels entry of name, process terminates.</td>
</tr>
<tr>
<td>4.</td>
<td>System prompts for the number.</td>
<td></td>
</tr>
<tr>
<td>5.</td>
<td>User enters number.</td>
<td>If user cancels entry of number, process terminates.</td>
</tr>
<tr>
<td>6.</td>
<td></td>
<td>The directory is updated to contain the new name and number. If the name was not already in the directory, the user is notified that a new name was entered. If the name already exists, the user is notified that the number was changed and is shown both the old and new numbers.</td>
</tr>
</tbody>
</table>

**Table 1.8**
Use Case for Retrieving and Displaying an Entry

<table>
<thead>
<tr>
<th>Step</th>
<th>User’s Action</th>
<th>System’s Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>User issues the command to retrieve and display an entry.</td>
<td></td>
</tr>
<tr>
<td>2.</td>
<td>System prompts for the name.</td>
<td></td>
</tr>
<tr>
<td>3.</td>
<td>User enters name.</td>
<td>If user cancels entry of name, process terminates.</td>
</tr>
<tr>
<td>4.</td>
<td></td>
<td>The system retrieves the entry from the directory. If found, the name and number are displayed; otherwise, a message is displayed indicating that the name is not in the directory.</td>
</tr>
</tbody>
</table>

**Refinement of Initial Class Diagram**

Earlier we used the top-down design approach to identify the subproblems to be solved (see Figures 1.3 through 1.5) and came up with the list of level 1 subproblems shown in the previous section. As discussed, you can combine the second and
third subproblems ("Insert a new entry", "Edit an existing entry") and add a sub-
problem to save the directory. The modified list follows:

- Read the initial directory from an existing file.
- Insert a new entry or edit an existing entry.
- Retrieve and display an entry.
- Save the modified directory back to the file.

The directory should be saved whenever the program is exited. The phone directory
has limited usefulness if updates to the directory cannot be saved from one run to
the next.

There is another way to split this problem into subproblems. Overall, we want a
phone directory application that combines a user interface as the front end and a
persistent (permanent) storage of data as the back end. Thus, Figure 1.6 can be
refined as shown in Figure 1.13. The black diamond in Figure 1.13 indicates that a
PD_Application object has an object of type Phone_Directory as its component and
that it is created by the PD_Application object. The arrowhead shows that
PD_Application updates the Phone_Directory object.

We have identified the abstract data type: the Phone_Directory. It is shown in the
class diagram as an interface. (UML does not have a notation for ADTs, but its
notation for interface is equivalent.) In Section 1.6 a class that implements this ADT
will be designed. By splitting the design between the user interface (PD_Application)
and the directory, we can work on them independently. As long as the requirements
defined by the interfaces are met, the front-end user interface does not care which
back end it is dealing with, and the back-end directory does not care which front
end it is dealing with.

### Design Overview of Classes and Their Interaction

Next, we identify all classes that will be part of the problem solution and describe
their interaction. Besides the class that implements the interface shown in Figure
1.13, classes from the C++ standard library will be used to perform input/output.

Note that the PD_Application class is artificial. In C++ the main function is a stand-
alone function—that is, it is not a member function of any class—and all programs
must have a main function. UML, however, does not have a notation for stand-alone
functions. Thus, we show the main function encapsulated in a class that represents
the program PD_Application. Table 1.9 shows a summary of some of the classes and
interfaces that will be used in our solution.
### Table 1.9
Summary of Classes and Objects Used in Phone Directory Solution

<table>
<thead>
<tr>
<th>Class/Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PD_Application</td>
<td>Contains the main function that provides the user interface.</td>
</tr>
<tr>
<td>Directory_Entry</td>
<td>Contains a name-number pair.</td>
</tr>
<tr>
<td>Phone_Directory</td>
<td>Class that implements the Phone_Directory ADT.</td>
</tr>
<tr>
<td>ifstream</td>
<td>A class in the C++ standard library that breaks a stream of input characters from a file into lines or data values.</td>
</tr>
<tr>
<td>istream cin</td>
<td>An object in the C++ standard library that breaks a stream of input characters from the console into lines or data values.</td>
</tr>
<tr>
<td>ofstream</td>
<td>A class in the C++ standard library that formats objects for output to a file.</td>
</tr>
<tr>
<td>ostream cout</td>
<td>An object in the C++ standard library that formats objects for output to the console.</td>
</tr>
</tbody>
</table>

From the use case in Table 1.6, we know that this function must create the Phone_Directory object and read in the initial directory. It then reads and executes the user’s commands.

The following algorithm for the main function is written in pseudocode, a combination of English and C++ language constructs.

**Algorithm for main Function**

1. Send the Phone_Directory a message to read the initial directory data from a file.
2. Read a command from the user.
3. **do**
4. Prompt the user for a command
5. Read the command
6. Send the appropriate message to the Phone_Directory to execute the command
7. Report the result to the user
8. **while** the command is not exit.

Next we show the UML sequence diagram for the main function. A sequence diagram (see Appendix B) is an OOD tool that documents the interaction between the objects in a program. Sequence diagrams are used to show the flow of information through the program and to identify the messages that are passed from one object to another.

**Sequence Diagram for main Function**

The sequence diagram for the main function is shown in Figure 1.14. The first (and only) parameter for main will be the name of the file containing the directory data. We show this event in the sequence diagram as the user issuing the message
main(source_name) to the PD_Application class. This corresponds to the user typing the command

    PD_Application source_name

on the command line.

The sequence diagram shows all the objects involved in this use case across the horizontal axis, with each object’s type underlined. Time is shown along the vertical axis. There is a dashed line coming down from each object that represents the object’s life line. When a function of this object is called, the dashed line becomes a solid line indicating that the function is executing. All interaction with an object is indicated by a horizontal line that terminates at the object’s life line.

For purposes of the diagram, we show the PD_Application object as being created when the application begins execution. Tracing down its life line, you can see that the main function first sends the Phone_Directory message to a class that implements the Phone_Directory ADT. This corresponds to the declaration statement

    Phone_Directory the_directory;

that creates a new Phone_Directory object. Next, main sends that object the load_data message. Looking at the life line for this Phone_Directory object, you see that function load_data creates a new ifstream object and sends it two getline messages. Next, load_data sends the add message to the Phone_Directory object. (Note that add is a new function that was not identified earlier.) This is the same object as the one that received the load_data message, so this add message is known as a message to self. Although the sequence diagram cannot show looping, the process of reading lines and adding entries continues until there are no remaining entries.
The sequence diagram (Figure 1.14) shows that member function `load_data` of the `Phone_Directory` object performs most of the work for the “Read initial directory data” use case. Member function `load_data` calls all the functions shown after it on the life line for the `Phone_Directory` object.

Upon return from the `load_data` function, the `main` function calls the `process_commands` function. This is a stand-alone function that is included in the `PD_Application.cpp` source file, so we show it on the UML diagram as a member of the `PD_Application` class. Design and implementation of the `process_commands` function is discussed in Section 1.8.

**EXERCISES FOR SECTION 1.5**

**SELF-CHECK**

1. Provide a use case for saving the directory to a file.
2. Draw a sequence diagram for “Insert or change an entry”.
3. Draw a sequence diagram for “Write the directory back to the file”.

---

**1.6 Design of an Array-Based Phone Directory**

The case study continues with the design, implementation, and testing of class `Directory_Entry` and a class that implements the `Phone_Directory` ADT. We will identify data fields for these classes and provide algorithms for their member functions. As we design the member functions of each class, we will identify new functions needed for that class. The design, implementation, and testing of these classes are discussed in the remaining sections of this chapter.

**CASE STUDY  Designing a Telephone Directory Program (cont.)**

**Design**

Design of Data Structures for the Phone Directory

Next, we consider the actual data elements that will be involved in the telephone directory problem. We will define a class `Directory_Entry`, which will contain the name-number pairs, and a class `Array_Based_PD`, which implements the `Phone_Directory` ADT. This class will contain an array of `Directory_Entry` objects. In later chapters we will show alternative designs that use classes that are part of the C++ standard library (for example, class `vector`).

Our new class diagram is shown in Figure 1.15. The line from `Array_Based_PD` to `Directory_Entry` indicates that `Directory_Entry` objects are components of
Array-Based_PD objects, but they can also be associated with other objects (for example, the data file). For class Directory_Entry, we show data fields (attributes) in the light-color screen and member functions in the darker-color screen. Next, we discuss the two actual classes shown in this diagram: Directory_Entry and Array_Based_PD.

**UML Syntax**

In UML class diagrams, the + sign next to the function names indicate that these functions are public. The – sign next to the attributes name and number indicate that they are private. For the class Directory_Entry we show the types of the attributes, and the parameter types and return types of the member functions. Showing this information on the diagram is optional. We will generally show this information in separate tables such as Table 1.9. Appendix B provides a summary of UML.
Design of the Directory_Entry Class

The Directory_Entry objects will contain the name-and-number pairs. The name is immutable; that is, it cannot be changed. For the purposes of your design, if you need to change the name of a person in your directory, you must remove the old entry and create a new one. The number, however, can be changed. Thus a straightforward design consists of

- Two data fields: name and number
- A constructor that sets both name and number
- Accessor functions for both name and number
- A mutator function for number

This design is shown in Table 1.10.

**Table 1.10**
Design of the Directory_Entry Class

<table>
<thead>
<tr>
<th>Data Field</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>string name</td>
<td>The name of the individual represented in the entry.</td>
</tr>
<tr>
<td>string number</td>
<td>The phone number for this individual.</td>
</tr>
</tbody>
</table>

**Constructor Behavior**

Directory_Entry(string name, string number) Creates a new Directory_Entry with the specified name and number.

**Function Behavior**

string get_name() const Retrieves the name.

string get_number() const Retrieves the number.

void set_number(const string& new_number) Sets the number to the specified value.

**Table 1.11**
Functions Declared in ADT Phone_Directory

<table>
<thead>
<tr>
<th>Function</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>void load_data(const string&amp; source_name)</td>
<td>Loads the data from the data file whose name is given by source_name.</td>
</tr>
<tr>
<td>string add_or_change_entry(const string&amp; name, const string&amp; number)</td>
<td>Changes the number associated with the given name to the new value, or adds a new entry with this name and number.</td>
</tr>
<tr>
<td>string lookup_entry(const string&amp; name) const</td>
<td>Searches the directory for the given name.</td>
</tr>
<tr>
<td>string remove_entry(const string&amp; name)</td>
<td>Removes the entry with the specified name from the directory and returns that person’s number or an empty string if not in the directory (left as an exercise).</td>
</tr>
<tr>
<td>void save()</td>
<td>Writes the contents of the array of directory entries to the data file.</td>
</tr>
</tbody>
</table>
Design of the **Array_Based_PD Class**
The **Array_Based_PD** class implements the **Phone_Directory** ADT. We showed a portion of this ADT earlier (Table 1.5); Table 1.11 shows the functions for the ADT. Class **Array_Based_PD** must implement these functions. It must also declare a data field for storage of the phone directory. Table 1.12 describes the data fields of class **Array_Based_PD**. In addition to the array of directory entries, the class includes data fields to help keep track of the array size and capacity and whether it has been modified. The functions will be designed in the next section.

**Design of **Array_Based_PD** Member Functions**

In this section you will complete the design of the **Array_Based_PD** class. At this stage you need to specify the function algorithms. First, we will develop pseudocode descriptions of the algorithms.

**Function load_data**

Function `load_data` is used to read the initial directory from a data file. The file name is passed as an argument to `load_data` when it is called.

**Algorithm for Function load_data**

1. Create an `ifstream` for the input file.
2. Read the name of the data file.
3. **while** the `ifstream` is not in the fail state
   4. Read the number.
   5. **if** the `ifstream` is not in the fail state
      6. Add a new entry using function `add`.
   7. Read the name.

Note that we have identified a new member function, `add`, for class **Array_Based_PD**.

**Function add_or_change_entry**

Function `add_or_change_entry` is used to either add a new entry to the directory or change an existing entry if the name is already in the directory. The name and number are passed as arguments to `add_or_change_entry`.

---

**TABLE 1.12**

Data Fields of Class **Array_Based_PD**

<table>
<thead>
<tr>
<th>Data Field</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>static const int INITIAL_CAPACITY</code></td>
<td>The initial capacity of the array to hold the directory entries.</td>
</tr>
<tr>
<td><code>int capacity</code></td>
<td>The current capacity of the array to hold the directory entries.</td>
</tr>
<tr>
<td><code>int size</code></td>
<td>The number of directory entries currently stored in the array.</td>
</tr>
<tr>
<td><code>Directory_Entry* the_directory</code></td>
<td>The array of directory entries.</td>
</tr>
<tr>
<td><code>string source_name</code></td>
<td>The name of the data file.</td>
</tr>
<tr>
<td><code>bool modified</code></td>
<td>A <code>bool</code> variable to indicate whether the contents of the array have been modified since they were last loaded or saved.</td>
</tr>
</tbody>
</table>
**Algorithm for Function add_or_change_entry**

1. Call function `find` to see whether the name is in the directory.
2. **if** the name is in the directory
3. Change the number using the `set_number` function of the `Directory_Entry`.
4. Return the previous value of the number.
5. **else**
6. Add a new entry using function `add`.
7. Return an empty string.

Note that we have identified another new function, `find`, for class `Array_Based_PD`.

**Function lookup_entry**

Function `lookup_entry` is passed a person’s name as an argument. It retrieves the person’s number if the name is found and an empty string if the name is not found.

**Algorithm for lookup_entry**

1. Call function `find` to see whether the name is in the directory.
2. **if** the entry is found
3. `Directory_Entry`’s `get_number` function retrieves the number, which is returned to the caller.
4. **else**
5. The empty string is returned.

**Function save**

Function `save` creates an output file and then writes all information stored in the array to this file. The file name is stored in data field `source_name`. The algorithm for the `save` function follows.

**Algorithm for save**

1. Create an `ofstream` object associated with the file.
2. **for** each entry in the array
3. Call `get_name` to get the name from the entry.
4. Write the name on a line.
5. Call `get_number` to get the number from the entry.
6. Write the number on a line.
7. Close the `ofstream` object.

Figure 1.16 shows the final class diagram with the additional functions.
EXERCISES FOR SECTION 1.6

SELF-CHECK
1. Write the algorithm for the use case “Remove entry”.

PROGRAMMING
1.7 Implementing and Testing the Array-Based Phone Directory

In this section, the case study continues to illustrate implementation and testing of the array-based phone directory. We will describe this as the class \texttt{Array\_Based\_PD}. The class definition will be in the file \texttt{Array\_Based\_PD.h} and the implementation in the file \texttt{Array\_Based\_PD.cpp}. However, the name of the class will be \texttt{Phone\_Directory}. This allows the client programs to be written to use the interface defined by the ADT. The client program will include \texttt{Array\_Based\_PD.h} to use this particular implementation of the ADT and will link with the \texttt{Array\_Based\_PD} object file. Later we will describe other implementations of the \texttt{Phone\_Directory}. To change the client to use the different implementation, we need only to change the \#include directive and link with the correct object file.

### CASE STUDY  
Designing a Telephone Directory Program (cont.)

#### Implementation

First we show the definition of class \texttt{Phone\_Directory} (See Listing 1.9, file \texttt{Array\_Based\_PD.h}). The data field declarations for the class are shown in the private part of Listing 1.9. As previously described, we use the Javadoc or Doxygen style for commenting the data fields.

**Listing 1.9**

\texttt{Array\_Based\_PD.h}

```cpp
#ifndef ARRAY_BASED_PD_H_
#define ARRAY_BASED_PD_H_
#include <string>
/** Specification file for the array-based phone directory. */
class Phone_Directory {

public:
  /** Construct an empty phone directory. */
  Phone_Directory();

  /** Destroy the phone directory when it is no longer needed. */
  ~Phone_Directory();

  /** Load the data file containing the directory, or establish a connection with the data source. */
  void load_data(const std::string& source_name);
};
```

```cpp
#endif
```

```cpp```
/** Look up an entry.  
@param name The name of the person to look up  
@return The number associated with that person  
on an empty string if name is not in  
the directory  
*/  
std::string lookup_entry(const std::string& name) const;

/** Changes the number associated with the given name to  
the new value, or adds a new entry with this name and  
number.  
@param name The name of the person  
@param number The new number  
@return The old value of number or an empty string  
if this is a new entry  
*/  
std::string add_or_change_entry(const std::string& name,  
const std::string& number);

/** Removes the entry with the specified name from the directory.  
@param name The name of the person  
@return The person's name or an empty string if not in  
the directory  
*/  
std::string remove_entry(const std::string& name);

/** Writes the contents of the directory to the data file. */  
void save();

private:  
// Insert the definition of the Directory_Entry class here.  

    // Private Functions  
/** Searches the array of directory entries for the name.  
@param name The name to be found  
@return The index of the entry containing the name, or size  
if the name is not found  
*/  
int find(const std::string& name) const;

/** Adds a new entry with the given name and number to the array  
of directory entries.  
@param name The name to be added  
@param number The number to be added  
*/  
void add(const std::string& name, const std::string& number);

/** Removes the entry at the given index.  
@param index The index of the entry to be removed  
*/  
void remove_entry(int index);

/** Creates a new array of directory entries with twice the  
capacity of the current one.  
*/  
void reallocate();
/** The number of entries in the directory. */
int size;
/** The current capacity of the array. */
int capacity;
/** Pointer to the array containing the directory data. */
Directory_Entry* the_directory;
/** The name of the data file that contains the directory data. */
std::string source_name;
/** Boolean flag to indicate whether the directory was modified since it was either loaded or saved. */
bool modified;
};
#endif

Coding the Functions in the Implementation File

Table 1.13 reviews the private functions defined in file class Array_Based_PD.cpp. They are private because they were not declared in the Phone_Directory interface and should not be called by a client. Two of these, add and find, were discussed previously. Function reallocate will be discussed later in this section. Function remove_entry is left as an exercise.

| Private Function | Behavior |
|------------------)|----------|
| int find(string& name) const | Searches the array of directory entries for the name. |
| void add(const string& name, string& number) | Adds a new entry with the given name and number to the array of directory entries. |
| void remove_entry(int index) | Removes the entry at the given index from the directory array. |
| void reallocate() | Creates a new array of directory entries with twice the capacity of the current one. |

The load_data Function

The load_data function (Listing 1.10) is called by the main function to read the initial directory data from an input file (parameter source_name). The data entry process takes place in the while loop. The while loop implements Steps 4 through 6 of the algorithm for load_data shown earlier. This function reads each name and number from two consecutive data lines and adds that entry to the array.

Listing 1.10

Function load_data from Array_Based_PD.cpp

/** Function to load the data file. 
 pre: The directory storage has been created and it is empty.
 If the file exists, it consists of the name-number pairs on adjacent lines.
 post: The data from the file is loaded into the directory.
 @param source_name The name of the data file
 */
void Phone_Directory::load_data(const string& source_name)
{
    // Remember the source name for use by save.
    this->source_name = source_name;
    // Create an input stream for this file.
    ifstream in(source_name.c_str());
    if (in) { // Stream exists.
        string name;
        string number;
        while (getline(in, name)) {
            if (getline(in, number)) {
                add(name, number);
            }
        }
    } // Close the file.
    in.close();
}

The getline function reads a line from an istream and returns it in a string object. The istream is the returned value. If there is no more data to be read, the istream is returned in the fail state, so we exit the while loop. Note that we combined the call to getline and an implicit call to fail in the while statement condition

    while (getline(in, name)) {...}

Similarly we combined the call to getline and fail when reading the number in the if condition

    if (getline(in, number))

Therefore, we also exit the loop if a name was read but a number was not. If both a name and number are read, then a new entry is added to the directory and we continue reading and adding entries.

If a file with name source_name is not found, we immediately return (no data to read). (Later, we will write the new directory to file source_name.)

The add_or_change_entry Function

This function calls the internal function find to locate the name in the array. Function find will either return the index of the entry, or return -1 (minus 1) to indicate that the entry is not in the array. If the entry is in the array, that entry’s set_number function is called to change the number; otherwise a new entry is added by calling the add function:

    /** Add an entry or change an existing entry.
       @param name The name of the person being added or changed
       @param number The new number to be assigned
       @return The old number or, if a new entry, an empty string
    */
    string Phone_Directory::add_or_change_entry(const string& name,
                                                const string& number)
{  
    string old_number = "";
    int index = find(name);
    if (index != -1) {
        old_number = the_directory[index].get_number();
        the_directory[index].set_number(number);
    } else {
        add(name, number);
    }
    modified = true;
    return old_number;
}

The `lookup_entry` Function

This function also uses the internal `find` function to locate the entry in the array. If the entry is located, it is returned; otherwise the empty string is returned.

```cpp
/** Look up an entry.
 * @param name The name of the person
 * @return The number. If not in the directory, an empty string
 */
string Phone_Directory::lookup_entry(const string& name) const {
    int index = find(name);
    if (index != -1) {
        return the_directory[index].get_number();
    } else {
        return "";
    }
}
```

The `save` Function

If the directory has not been modified, function save does nothing. Otherwise it creates an `ofstream` object and writes all phone directory entries to it. The output file name is the same as the input file name (`source_name`), so an existing directory file will be overwritten. The `for` statement writes the entries.

```cpp
/** Function to save the directory.
 * pre: The directory has been loaded with data
 * post: Contents of directory written back to the file in the
 *       form of name-number pairs on adjacent lines.
 *       modified is reset to false.
 */
void Phone_Directory::save() {
    if (modified) {  // If not modified, do nothing
        // Create an output stream.
        ofstream out(source_name.c_str());
        for (int i = 0; i < size; i++) {
            out << the_directory[i].get_name() << endl;
            out << the_directory[i].get_number() << endl;
        }
        // Close the output stream.
        out.close();
        modified = false;
    }
}
```
The find Function

The find function uses a \texttt{for} loop to search the array for the requested name. If located, its index is returned; otherwise \texttt{-1} is returned.

\begin{verbatim}
/** Search the array for a given name.
   @param name The name to be found
   @return The index of the entry containing this name
           or -1 if the name is not present
 */
int Phone_Directory::find(const string& name) const {
    for (int i = 0; i < size; i++) {
        if (the_directory[i].get_name() == name)
            return i;
    }
    return -1;
}
\end{verbatim}

The add Function

The add function checks to see whether there is room in the array by comparing the size to the capacity. If the size is less than the capacity, the new entry is stored at the end of the array and size is incremented by one after the entry is stored (\texttt{size++}). If the size is greater than or equal to the capacity, then the \texttt{reallocate} function is called to increase the size of the array before the new item is inserted.

\begin{verbatim}
/** Add a new name-number pair to the directory.
   @param name The name to be added
   @param number The number to be added
 */
void Phone_Directory::add(const string& name, const string& number) {
    if (size == capacity) // If no room, reallocate.
        reallocate();
    // Increment size and add new entry.
    the_directory[size] = Directory_Entry(name, number);
    size++;
}
\end{verbatim}

\textbf{PITFALL}

Returning \texttt{-1} (Failure) Before Examining All Array Elements

A common logic error is to code the search loop for function find as follows:

\begin{verbatim}
for (int i = 0; i < size; i++) {
    if (the_directory[i].get_name() == name) {
        return i;
    } else {
        return -1; // Incorrect! - tests only one element.
    }
}
\end{verbatim}

This loop incorrectly returns a result after testing just the first element.
The **reallocate** Function

This function allocates a new array whose size is twice the current array. We use a `for` loop to copy the old array to the new array, and `the_directory` is changed to point to the new array. The storage allocated to the old array is then returned to the free-memory pool via the `delete[]` operator.

By doubling the size each time that a reallocation is necessary, we reduce the number of times we need to do this. Surprisingly, if we do this only fourteen times, we can store over 1 million entries.

```cpp
/** Create a new array of directory entries with twice the capacity of the current one. */
void Phone_Directory::reallocate() {
    // Double the capacity.
    capacity *= 2;
    // Create a new directory array.
    Directory_Entry* new_directory = new Directory_Entry[capacity];
    // Copy the old to the new
    for (int i = 0; i < size; i++) {
        new_directory[i] = the_directory[i];
    }
    // Return the memory occupied by the old directory.
    delete[] the_directory;
    // Set the_directory to point to the new directory.
    the_directory = new_directory;
}
```

Using a Storage Structure Without Reallocation

In Chapter 4, you will study the vector data structure, which will enable you to store a directory of increasing size without needing to reallocate storage. You will see that we can change to a different data structure for storing the directory with very little effort, because the problem solution has been so carefully designed. We will only need to code the functions declared in the `Phone_Directory` ADT so that they perform the same operations on a vector.

---

**PITFALL**

**Failure to Delete Dynamically Allocated Arrays**

If we do not return the storage allocated to the old array to the free-memory pool, this memory is lost. This is known as a memory-leak. Although modern computers seem to have infinite memory, there is in fact only a finite supply. Furthermore, programs that leak memory can impact other programs running on the same computer. A program that routinely leaks memory can eventually require you to reboot your computer.
Testing  

Class Array_Based_PD

To test this class, you should run it with data files that are empty or that contain a single name-and-number pair. You should also run it with data files that contain an odd number of lines (ending with a name but no number). You should see what happens when the array is completely filled and you try to add a new entry. Does function reallocate properly double the array’s size? When you do a retrieval or an edit operation, make sure you try to retrieve names that are not in the directory as well as names that are in the directory. If an entry has been changed, verify that the new number is retrieved. Finally, check that all new and edited entries are written correctly to the output file. We will discuss testing further in the next chapter.

EXERCISES FOR SECTION 1.7

PROGRAMMING

1. Code the remove_entry function.

1.8 Completing the Phone Directory Application

So far we have described the main function and the Phone_Directory ADT. The main function creates a Phone_Directory object, calls its load_data function, and then calls the process_commands function.

CASE STUDY  Designing a Telephone Directory Program (cont.)

Analysis

The process_commands function is declared as follows:

```c
void process_commands(Phone_Directory the_directory);
```

The interface enables clients to use function process_commands without knowing the details of its implementation (information hiding). We will introduce additional functions that are called by process_commands to perform its tasks.

The function process_commands should present a menu of choices to the user:

- Add or Change an Entry
- Look Up an Entry
- Remove an Entry
- Save the Directory Data
- Exit the Program
Design

The function process_commands will use a “menu-driven” loop to control the interaction with the user. After each command is processed, the menu of choices is displayed again. This process continues until the user selects “Exit”.

```cpp
    do {
        // Get the action to perform from the user.
        // The user's choice will be a number from 0 through 4.
        ...
        switch (choice) {
            case 0: do_add_change_entry(); break;
            case 1: do_lookup_entry(); break;
            case 2: do_remove_entry(); break;
            case 3: do_save(); break;
            case 4: do_save(); break;
        }
    } while (choice < NUM_COMMANDS - 1);
```

The function process_commands calls a function shown in the foregoing `switch` statement to perform the user’s choice. Note that function do_save is called by the last two cases. We discuss the design and coding of these functions next.

Implementation

Listing 1.11 shows the code for the PD_Application. This program uses cout to display the menu of choices and results. It also uses cin to read data from the user. It begins with forward definitions of the functions that are called in the `switch` statement. C++ functions must be declared in a function prototype that appears before the first use of the function. The prototype consists of the heading of the function only.

Function `process_commands`

The function process_commands begins by displaying the menu:

```
    Select: 0 Add/Change Entry
    Select: 1 Look Up Entry
    Select: 2 Remove Entry
    Select: 3 Save Directory
    Select: 4 Exit
```

The first statement below

```cpp
    cin >> choice;
    cin.ignore(numeric_limits<int>::max(), '\n');
```

reads the user's choice (an integer) from the console. The second statements skips over the newline character that follows the integer.

Function `do_add_change_entry`

The do_add_change_entry function requests the name followed by the number. The Phone_Directory.add_or_change_entry function is called after values are entered for the name and number. A return value of an empty string indicates that this is a new entry, and a confirmation dialog is displayed as follows

```
    Enter name: Quincy
    Enter number: 555-111-3333
    Quincy has been added to the directory
```
If the name was already in the directory, the previous value of the number is returned, and the confirmation shows both the old and new number as follows:

```
Enter name: Tom
Enter number: 123-456-7890
Tom has been changed in the directory
Old number was 444-555-6666
```

**Function do_lookup_entry**

The do_lookup_entry function uses the same prompt as do_add_change_entry to request the name. The number is looked up by calling the Phone_Directory.lookup_entry function, and the result is displayed. If the name is not in the directory, a message is displayed. Examples of both cases follow:

```
Enter name: Tom
The number for Tom is 123-456-7890

Enter name: Dick
Dick is not in the directory
```

**Function do_save**

The do_save function calls the save function of the Phone_Directory.

---

**Listing 1.11**

*PD_Application.cpp*

```cpp
/** Phone directory application that uses console I/O. */
#include "Array_Based_PD.h"
#include <iostream>
#include <istream>
#include <ostream>
#include <limits>
using namespace std;

// Forward declaration of functions
void process_commands(Phone_Directory&);
void do_add_change_entry(Phone_Directory&);
void do_lookup_entry(Phone_Directory&);
void do_remove_entry(Phone_Directory&);
void do_save(Phone_Directory&);

int main(int argc, char* argv[]) {
  if (argc < 2) {
    cerr << "Must specify the name of the data file" << " that contains the directory\n";
    return 1;
  }
  Phone_Directory the_directory;
  the_directory.load_data(argv[1]);
  process_commands(the_directory);
}
```
void process_commands(Phone_Directory& the_directory) {
    string commands[] = {
        "Add/Change Entry",
        "Look Up Entry",
        "Remove Entry",
        "Save Directory",
        "Exit" 
    };
    const int NUM_COMMANDS = 5;
    int choice = NUM_COMMANDS - 1;
    do {
        for (int i = 0; i < NUM_COMMANDS; i++) {
            cout << "Select: " << i << " " << commands[i] << "\n";
        }
        cin >> choice;
        cin.ignore(numeric_limits<int>::max(), '\n');
        switch (choice) {
            case 0: do_add_change_entry(the_directory); break;
            case 1: do_lookup_entry(the_directory); break;
            case 2: do_remove_entry(the_directory); break;
            case 3: do_save(the_directory); break;
            case 4: do_save(the_directory); break;
        }
    } while (choice < NUM_COMMANDS - 1);
}

void do_add_change_entry(Phone_Directory& the_directory) {
    string name;
    string number;
    cout << "Enter name: ";
    getline(cin, name);
    cout << "Enter number: ";
    getline(cin, number);
    string old_number =
        the_directory.add_or_change_entry(name, number);
    if (old_number != "") {
        cout << name << " has been changed in the directory\n"
        cout << "Old number was " << old_number << "\n";
    } else {
        cout << name << " has been added to the directory\n"
    }
}

void do_lookup_entry(Phone_Directory& the_directory) {
    string name;
    cout << "Enter name: ";
    getline(cin, name);
    string number = the_directory.lookup_entry(name);
    if (number != "") {
        cout << "The number for " << name << " is " << number << "\n";
    } else {
        cout << name << " is not in the directory\n"
    }
}

void do_remove_entry(Phone_Directory& the_directory) {
    // Exercise
}

void do_save(Phone_Directory& the_directory) {
    the_directory.save();
}
EXERCISES FOR SECTION 1.8

PROGRAMMING

1. Code the do_remove_entry function.

Chapter Review

- We discussed the software engineering process. We introduced two software life cycle models (waterfall and Unified) and discussed the activities performed in each stage of these models. In this text we will use a simplified five-step process for developing software:

1. Requirements
2. Analysis
3. Design
4. Implementation
5. Testing

- Although these steps are shown in sequence, in reality there is quite a bit of interaction between them. During the design phase, you may need to go back and redo the requirements and analysis steps. Of course, this is provided for in the full Unified Model.

- Procedural abstraction, data abstraction, and information hiding are tools for managing program complexity, so that programs can be designed as a collection of separate but interacting classes. Procedural abstraction focuses on the operations to be performed; data abstraction focuses on the data elements and their operations; information hiding means that users of a class or function need to know only how to use the class or function, not how it is implemented.

- An Abstract Data Type (ADT) defines a set of objects and the operations that can be performed on them. An ADT can have several classes that implement it (define its data and operations).

- Use cases summarize the interaction between the user and the system during requirements specification and analysis.

- UML class diagrams are used during the analysis and design phases to document the interaction of classes with each other and with the user.

- Sequence diagrams and pseudocode can be used to describe the sequence of actions performed by a program that is implemented as a collection of multiple interacting classes. Sequence diagrams are employed during the design phase of the software life cycle.
Quick-Check Exercises

1. The disadvantage of the ______ model is that it assumes that the software life cycle stages are performed in sequence and not revisited once completed. The Unified Model consists of ______ and ______, and the system designer can work on more than one ______ during each iteration.

2. Procedural abstraction enables a system designer to model ______; data abstraction focuses on the ______ and ______.

3. An ADT specifies a contract between the ______ and ______; a ______ implements the ADT.

4. An ADT can be implemented by multiple classes (True/False).

5. A ______ is a statement that must be true before a ______ executes, and a ______ represents the ______ of executing a ______.

6. A ______ specifies the interaction between an external entity and a system.

7. ______ means that a class ______ does not need to know its implementation details.

8. ______ is a mixture of English and ______ used to specify an algorithm.

Answers to Quick-Check Exercises

1. The disadvantage of the *waterfall* model is that it assumes that the software life cycle stages are performed in sequence and not revisited once completed. The Unified Model consists of *activities/workflows* and *phases*, and the system designer can work on more than one *activity* during each iteration.

2. Procedural abstraction enables a system designer to model *processes/operations*; data abstraction focuses on the *data elements* and *operations on that data.*

3. An ADT specifies a contract between the *developer* and *user*; a *class* implements the ADT.

4. True.

5. A *precondition* is a statement that must be true before a *function* executes, and a *post-condition* represents the *result* of executing a *function*.

6. A *use case* specifies the interaction between an external entity and a system.

7. *Information hiding* means that a class *user* does not need to know its implementation details.

8. *Pseudocode* is a mixture of English and *programming language* used to specify an algorithm.

Review Questions

1. Explain why the principle of information hiding is important to the software designer.

2. Discuss the differences between the waterfall model and Unified Model of the software life cycle.

3. Define the terms *procedural abstraction* and *data abstraction*.

4. Explain the role of function preconditions and postconditions.

5. What is the advantage of specifying an abstract data type instead of just going ahead and implementing a class?

6. Define an ADT *Money* that has operations for arithmetic operations (addition, subtraction, multiplication, and division) on real numbers having exactly two digits to the right
of the decimal point, as well as functions for representing a Money object as a string and as a real number. Also include the operators == and < for this ADT.

7. Answer Review Question 6 for an ADT Complex that has operations for arithmetic operations on a complex number (a number with a real and imaginary part). Assume that the same operations (+, −, *, /) are supported. Also define the == and != operators and overload the ostream extraction operator.

Programming Projects

1. Modify the telephone directory project in this chapter so that it could be used by a company. Each employee’s information should contain a name, job description, telephone number, and room number. Assume that the information for each person is available on a single line of a data file in the form last name, first name, job description, phone number, room number with commas as separators between data items. Provide a submenu for the edit operation that allows the user to edit any of the person’s data fields. After using the directory and updating it, save the data file in the same format as before.

2. Follow the software development model illustrated in this chapter to design and implement an array-based program application that manages a collection of DVDs. The data for each DVD will consist of a title, a category, running time, year of release, and price. Use a file to save the collection after it has been updated. The user should be able to add new DVDs to the collection, remove a DVD, edit the information stored for a DVD, list all DVDs in a specified category, and retrieve and display the information saved for a DVD given its title. Also, the user should be able to sort the collection of DVDs by year or by title.

3. Follow the software development model illustrated in this chapter to design and implement an array-based program application that manages your personal library. The data for each book will consist of a title, author, number of pages, year of publication, price, and the name of the person who has borrowed it (if any). Use a file to save the collection after it has been updated. The user should be able to add new books to the collection, remove a book, edit the information stored for a book, list all books by a specified author, list all books loaned to a particular person, and retrieve and display the information saved for a book given its title. Also, the user should be able to sort the collection of books by author or by title.

4. Assume that you have decided to loan DVDs in your DVD collection to friends (see Project 2) but just one DVD at a time to each friend. Assume that you also have a collection of friends. For each friend, you need to store the friend’s name and phone number and the number of the DVD your friend has borrowed (−1 if none). Besides managing your DVD collection as in Project 2, you should also be able to manage your list of friends (add friend, edit friend, and so on). One of the edit operations should allow your friend to return a DVD or to exchange it for another. You should also be able to display each friend and the name of the DVD that the friend currently has (if any). Draw UML class diagrams before you start to code the program.

5. Develop a program that could be used to determine election results for a town. Assume that the town is divided into a number of precincts (an input item) and that the number of candidates is variable (an input item). The votes received for each candidate by precinct should be stored in a two-dimensional array that is part of class Vote_Tabulation. Your program should allow its user to issue any of the following instructions:
• Load the vote data from a file
• Load the vote data interactively
• Edit the vote data interactively to account for absentee ballots just received
• Request a table showing the raw vote results by precinct and candidate
• Request a table showing the percentage of votes by precinct and candidate
• Request a table showing the raw vote results by candidate and precinct
• Request a table showing the percentage of votes by candidate and precinct
• Display the top \( n \) vote getters for the township and their results in decreasing order, where \( n \) is a data item

6. Develop a program that could be used as a point-of-order inventory system. Read in a database from a file that represents the store inventory. Each item consists of an ID, a description, a price, a quantity on hand, and a reorder point. Assume that the database can be updated at a terminal by the operator when a customer purchases an item. The quantity purchased for an item is deducted from that item’s quantity (if sufficient) and a register receipt is displayed after all purchases are entered. An operator can also process the return of an item by adding the quantity returned to the inventory. A register receipt should also be displayed for a return. Finally, the operator can process new items received from a supplier by adding the items to the database. It should also be possible for an operator to update the price of an item, provided the operator enters the correct security key. Also, the operator should be able to display a list of all items whose quantity on hand is less than the reorder point for that item.

7. Develop a sales-tracking program that enables a company to keep track of its sales force’s performance by quarter. The program should read in each person’s name and the past performance of that salesperson for the last \( n \) quarters, where \( n \) is a data item. The operator should be able to enter the results for each person for the current quarter interactively. The operator should be able to add a new salesperson to the system (past performances set to 0) and edit the information stored for a salesperson. The operator should be able to request any of the following tables:
• Total sales by quarter (for all quarters or for a specified quarter)
• Sales for each person by quarter (for all quarters or for a specified quarter)
• Rank of each person by quarter (for all quarters or for a specified quarter)
• A list of salespeople and amounts in decreasing order by quarter (for all quarters or for a specified quarter)