Part Five
Introduction to C++

16 Introduction to C++
17 Additional Class Capabilities
16.1 Object-Based Programming and Abstract Data Types

Besides being an improved version of C, the distinguishing characteristic of C++ is its support of object-oriented programming. Central to this object orientation is the concept of an abstract data type, which is a programmer-defined data type. In this chapter we explore the implications of permitting programmers to define their own data types and then present C++’s mechanism for constructing abstract data types. As we will see, the construction of a data type is based on both structures and functions; structures provide the means for creating new data configurations, and functions provide the means for performing operations on these structures. What C++ provides is a unique way of combining structures and functions together in a self-contained, cohesive unit from which objects can be created.

16.1 Object-Based Programming and Abstract Data Types

The emergence of the graphical screen and the subsequent interest in window applications have dramatically changed the programming environment since the mid-1990s. Programming multiple and possibly overlapping windows on the same graphical screen is extremely
difficult using standard procedural, input/output programming techniques. Similarly, providing a graphical user interface (GUI) where a user can easily move around in even a single window is a challenge in C.

Unlike a procedural approach, however, an object-oriented approach works well in a graphical windowed environment, where each window can be specified as a self-contained rectangular object that can be moved and resized in relation to other objects on the screen. Additionally, within each window, other graphical objects, such as check boxes, option buttons, labels, and text boxes, can easily be placed and moved.

To provide this object creation capability, extensions to the procedural language C were developed. These extensions became the new language named C++, which permits a programmer to both use and create new objects.

Central to the creation of objects is the concept of an abstract data type, which is simply a user-defined data type as opposed to the built-in data types provided by all languages (such as integer and floating-point types). Permitting a programmer to define new data types, such as a rectangular type, out of which specific rectangular objects can be created and displayed on a screen, forms the basis of C++’s object orientation.
Abstract Data Types

To gain a clear understanding of an abstract data type, consider the following three built-in data types supplied in C: int, float, and char. In using these data types, we typically declare one or more variables of the desired type, use them in their accepted ways, and avoid using them in ways that are not specified. Thus, for example, we do not use the modulus operator on two floating-point numbers. Because this operation makes no sense for floating-point numbers, it is never defined, in any programming language, for such numbers. Thus, although we typically don’t consider it, each data type consists of both a type of data, such as integer or float, and specific operational capabilities provided for each type.

In computer terminology, a data type is the combination of data and their associated operations. That is, a data type defines both the types of data and the types of operations that can be performed on the data. Seen in this light, the int data type, the float data type, and the char data type provided in C are all examples of built-in data types that are defined by a type of data and specific operational capabilities provided for initializing and manipulating the type. In a simplified form, this relationship can be described as

\[
\text{Data Type} = \text{Allowable Data} + \text{Operational Capabilities}
\]

Although we don’t normally associate the operations that we apply to C’s built-in data types with the data type itself, these operations are an inherent part of each data type. Clearly the designers of C had to carefully consider, and then implement, specific operations for each type of data supplied by the language.

To understand the importance of the operational capabilities provided by a programming language, let’s take a moment to list some of those supplied with C’s built-in data types (integer, floating-point, and character). The minimum set of the capabilities provided by C’s built-in data types is listed in Table 16.1.

<table>
<thead>
<tr>
<th>Capability</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Define one or more variables of the data type</td>
<td>int a, b;</td>
</tr>
<tr>
<td>Initialize a variable at definition</td>
<td>int a = 5;</td>
</tr>
<tr>
<td>Assign a value to a variable</td>
<td>a = 10;</td>
</tr>
<tr>
<td>Assign one variable’s value to another variable</td>
<td>a = b;</td>
</tr>
<tr>
<td>Perform mathematical operations</td>
<td>a + b</td>
</tr>
<tr>
<td>Convert from one data type to another</td>
<td>a = (int) 7.2;</td>
</tr>
</tbody>
</table>

Now let’s see how all of this relates to abstract data types (ADTs). By definition an abstract data type is simply a user-defined type that defines both a type of data and the operations that can be performed on it. Such user-defined data types are required when we wish to create objects that are more complex than simple integers and characters. If we are to create our own data types, we must be aware of both the type of data we are creating and the capabilities that we provide to initialize and manipulate the data.

As a specific example, assume that we are programming an application that uses dates extensively. Clearly we can define a data structure named Date that consists of three data members as follows:

```c
struct Date
{
    int month;
};
```
int day;
int year;
};

Specifically, we have chosen to store a date using three integers, one for the month, day, and year, respectively. In addition, we will require that the year be stored as a four-digit number. Thus, for example, we will store the year 1999 as 1999 and not 99. Making sure to store all years with their correct century designation eliminates a number of problems that can crop up if only the last two digits, such as 99, are stored. For example, the number of years between 2007 and 1999 can quickly be calculated as 2007 – 1999 = 8 years, while this same answer is not so easily obtained if only the year values 07 and 99 are used. Additionally, we are sure of what the year 2007 refers to, whereas a two-digit value such as 07 could refer to either 1907 or 2007.

The data structure for storing a date, unfortunately, supplies only half of the answer. We must still supply a set of operations that can be used with dates. Clearly, such operations could include assigning values to a date, subtracting two dates to determine the number of days between them, comparing two dates to determine which is earlier and which is later, or displaying a date in a form such as 6/3/06. The combination of the Date data structure with a set of Date operations would then define an abstract Date data type. Although C does not provide a direct capability for constructing abstract data types (providing this capability was the impetus for developing C++), it is possible to simulate an abstract data type in C. This is done by using a data structure for defining a type of data and then defining a set of functions to specifically operate on the data structure. We now construct a set of suitable functions for our Date data structure. Specifically we will develop two functions for operating on dates, assignment and display, and leave the development of some of the remaining operations as an exercise. Following are the specifications that our two functions must meet:

**Assignment:**

*Accept three integers as arguments*
*Assign the first number as the Date's month member*
*Assign the second number as the Date's day member*
*Assign the third number as the Date's year member*
*Return the Date*

**Display:**

*Accept a Date as a parameter*
*Display the values in the form month/day/year*

Notice that the details of each function are of interest to us only as we develop each operation. Once developed, however, we need never be concerned with how the operations are performed. When we use an operation, all we will need to know is what the operation does and how to invoke it, much as we use C's built-in operations. For example, we don't really care how the addition of two integers is performed but only that it is performed correctly.

The actual coding of our two specified functions is rather simple:

```c
/* operation to assign values to a Date object */
struct Date setdate(int mm, int dd, int yyyy)
{
    struct Date temp;
    temp.month = mm;
    temp.day = dd;
    return temp;
}
```
The first function header line
struct Date setdate(int mm, int dd, int yyyy)
defines a function that will return a Date structure and uses three integer parameters, mm, dd, and yyyy. The body of this function assigns the data members month, day, and year with the values of these parameters and returns the assigned Date structure.

The last function header line defines a function named showdate(). This function has no parameters and returns no value. The body of this function, however, needs a little more explanation.

Although we have chosen to internally store all years as four-digit values that retain century information, users are accustomed to seeing dates with the year represented as a two-digit value, such as 12/15/99. To display the last two digits of the year value, the expression year % 100 is used in the printf() function. For example, the expression 1999 % 100 yields the value 99, and if the year is 2005, the expression 2005 % 100 yields the value 1. The %02d format forces each value to be displayed in a field width of 2 with a leading zero, if necessary. Doing this ensures that a date such as December 9, 2006, appears as 12/09/06 and not 12/9/6.

To see how our simulated Date abstract data type can be used within the context of a complete program, consider Program 16.1. To make the program easier to read it has been shaded in lighter and darker areas. The lighter area contains the data structure declaration and operation implementation sections that define the abstract data type. The darker area contains the header and main() function.

The declaration and implementation sections contained in the light-shaded region of Program 16.1 should look familiar—they contain the declaration and implementations that have already been discussed. Notice, however, that this region only declares a Date type, it does not create any Date variables. This is true of all data types, including built-in types such as integer and floating point. Just as a variable of an integer type must be defined, variables of a user-declared data type must also be defined. Variables defined to be of a user-declared data type are referred to as objects.

Using this new terminology, the first statement in Program 16.1’s main() function, contained in the darker area, defines a single object, named a, to be of the type Date. The next statement in main(), a = setdate(6, 3, 2005), assigns the argument values 6, 3, and 2005 to a’s data members, resulting in the assignment:

a.month = 6
a.day = 3
a.year = 2005

Finally, the next-to-last statement in main() calls the showdate() function to display the values of a Date object. As the following output of Program 16.1 illustrates, our two functions are operating correctly.
The value of the a object is: 06/03/05

It should be reemphasized that we have not created an abstract data type in Program 16.1 but only simulated two essential pieces of a user-created data type—that is, a type of data declared by a record structure and a very limited set of operations defined as straightforward C functions. To create a true user-created data type we will need to more closely “bundle” the data with its defined set of operations and include many more operations. The capability to create a true abstract data type is provided in C++. This is because C++ provides the ability to incorporate functions as well as data members within the declaration of a structure, whereas in C, structures can contain only data members.

In addition to providing the capability of constructing true abstract data types, C++ also provides a substantial set of improvements to traditional ANSI C. Most of these extend function capabilities and add new capabilities for handling input and output. As these enhancements are almost always incorporated in constructing abstract data type, we will need to consider them if we are to fully understand and create true C++ abstract data types.

---

**Program 16.1**

```c
#include <stdio.h>

/* declaration of the data structure */
struct Date
{
    int month;
    int day;
    int year;
};

/* implementation of associated functions */

/* operation to assign values to a Date object */
struct Date setdate(int mm, int dd, int yyyy)
{
    struct Date temp;
    temp.month = mm;
    temp.day = dd;
    temp.year = yyyy;
    return(temp);
}

/* operation to display a Date object */
void showdate (struct Date a)
{
    printf("%02d/%02d/%02d", a.month, a.day, a.year % 100);
}
```
We can, however, provide a preview of a C++ abstract data type that should make sense to you. Program 16.2 shows how Program 16.1 could be written in C++. The important part of Program 16.2 to notice and compare to Program 16.1 is the middle, lightly shaded area, which contains the Date structure declaration. Notice in Program 16.2 that the C++ Date structure includes declarations for both data and function members, whereas the C Date structure includes only data declarations. Also notice the use of the private and public keywords contained within the structure. These keywords permit restricted access to the data members through the member functions and are described more fully in Section 16.4. Finally, although we have included numerous printf() function calls within Program 16.2, each of these calls would be replaced by an equivalent C++ output statement. Exactly how C++'s input and output are performed are the topic of Section 16.2.

Comments in C++
In addition to the comments specified by C's /* and */ notation, C++ provides for line comments. A line comment begins with two slashes (//) and continues to the end of the line. For example,

```
// this is a comment
// this program prints out a message
// this program calculates a square root
```

are all line comments. The symbols //, with no white space between them, designate the start of a line comment. The end of the line on which the comment is written designates the end of the comment.

A line comment can be written either on a line by itself or at the end of a line containing a program statement. For example,

```
float average; // this is a declaration statement
```

contains a line comment on the same line as a declaration statement. If a comment is too long to be contained on one line, it can be either separated into two or more line comments with each separate comment preceded by the double slash symbol set, //, or designated using C's comment notation. Thus, the multiline comment:

```
// this comment is used to illustrate a
// comment that extends across two lines
```

can also be written by placing the comment between the /* and */ symbols.
Program 16.2

```c
#include <stdio.h>

/* declaration of the data structure */
struct Date
{
private:
    int month;
    int day;
    int year;
public:
    void setdate(int, int, int); /* notice the function */
    void showdate(); /* declarations within the */
}; /* Date structure */

/* implementation of associated functions */

/* operation to assign values to a Date object */
void Date::setdate(int mm, int dd, int yyyy)
{
    month = mm;
    day = dd;
    year = yyyy;
    return;
}

/* operation to display a Date object */
void Date::showdate()
{
    printf("%02d/%02d/%02d", month, day, year % 100);
}

int main()
{
    struct Date a;
    a.setdate(6,3,2005);
    printf("\nThe value of the a object is: ");
    a.showdate();
    printf("\n");
    return 0;
}
```
**EXERCISES 16.1**

1. Modify Program 16.1 by adding a function named `convert()`. The function should access the month, year, and day data members of a `Date` and then return a long integer that is the calculated as `year * 10000 + month * 100 + day`. For example, if the date is 4/1/2006, the returned value is 20060401. (Dates in this form are useful when performing sorts, because placing the numbers in numerical order automatically places the corresponding dates in chronological order.) Make sure to declare the return value as a long integer to accommodate larger integer values.

2. Modify Program 16.1 to contain a function that compares two `Date` objects and returns the larger of the two. The function should be written according to the following algorithm:

   **Comparison function**
   
   **Accept two Date values as arguments**
   
   **Determine the later date using the following procedure:**
   
   Convert each date into an integer value having the form `yyyymmdd`. This can be accomplished by using the algorithm described in Exercise 1. Compare the corresponding integers for each date. The larger integer corresponds to the later date.
   
   **Return the later date**

3. a. Add a member function named `leapyr()` to Program 16.1. The function should return a 1 when a `Date` object falls in a leap year and a 0 if it is does not. A leap year is any year that is evenly divisible by 4 but not evenly divisible by 100, with the exception that years evenly divisible by 400 are leap years. For example, the year 1996 was a leap year because it is evenly divisible by 4 and not evenly divisible by 100. The year 2000 was a leap year because it is evenly divisible by 400.

4. Write data structures appropriate for each of the following specifications. Also list appropriate operations for each data structure.

   a. A structure named `Time` that has integer data members named `secs`, `mins`, and `hours`.

   b. A structure named `Circle` that has integer data members named `xcenter` and `ycenter` and a floating-point data member named `radius`.

   c. A structure named `Complex` that has floating-point members named `real` and `imaginary`.

   d. A structure named `System` that has character data members named `computer`, `printer`, and `screen`, each capable of holding 30 characters (including the end-of-string NULL), and floating-point data members named `compPrice`, `printPrice`, and `scrnPrice`.

5. a. Construct C functions for the function members you listed in Exercise 4a.

   b. Construct C functions for the function members you listed in Exercise 4b.

   c. Construct C functions for the function members you listed in Exercise 4c.

   d. Construct C functions for the function members you listed in Exercise 4d.
16-12 Introduction to C++

6. a. Include the declaration and function definitions for Exercises 4a and 5a in a complete working program.
b. Include the declaration and function definitions for Exercises 4b and 5b in a complete working program.
c. Include the declaration and function definitions for Exercises 4c and 5c in a complete working program.
d. Include the declaration and function definitions for Exercises 4d and 5d in a complete working program.

16.2 Input and Output in C++

Although the printf() and scanf() functions are available in C++, the standard method of C++’s input and output is stream I/O. In many situations stream I/O is much simpler to use than the comparable C functions, and it avoids many of the mistakes programmers tend to make with printf() and scanf().

Output in C++

An output display in C++ is constructed as a stream of characters using a C++-provided object named cout (pronounced “see out”). This object, whose name was derived from “Console OUTPut,” simply sends data given to it to the standard system display device. For most systems this display device is a video screen. The cout object simply gathers together all the data passed to it and sends it on for display. For example, if the data Hello there world! is passed to cout, this data is printed (or displayed on your terminal) by the cout object. The data Hello there world! is passed to the cout object by simply putting the insertion (“put to”) symbol, <<, before the message and after the object’s name.

    cout << "Hello there world!";

Within a working C++ program this statement could appear as shown in Program 16.3.

Program 16.3

```cpp
#include <iostream>
using namespace std;

int main()
{
    cout << "Hello there world!";
    return 0;
}
```

Notice the first two lines of the program:

```
#include <iostream>
using namespace std;
```
This is a required preprocessor command when `cout` is being used and effectively replaces the equivalent `stdio.h` header file used in C. In particular, the `iostream` file provides two abstract date types, `istream` and `ostream`, that contain the actual data definitions and operations used for data input and output.¹

In addition to displaying strings, the `cout` object also provides for the display of an expression's numerical value. For example, the statement

```
cout << (6 + 15);
```

yields the display 21. Strictly speaking, the parentheses surrounding the expression `6 + 15` are required to indicate that it is the value of the expression, which is 21, that is being placed on the output stream. In practice most compilers will accept and correctly process this statement without the parentheses.

Both strings and expressions may be included in the same `cout` statement. For example, the statement:

```
cout << "The total of 6 and 15 is " << (6 + 15) << '\n';
```

causes three pieces of data to be sent to `cout`: a string, a numerical value, and the newline constant. As illustrated by this statement, each piece of data is sent to `cout` preceded by its own insertion symbol (`<<`). Here, the first data item sent to the stream is the string "The total of 6 and 15 is ", the second item is the value of the expression `6 + 15`, and the third item is the newline character. The display produced by this statement is

```
The total of 6 and 15 is 21
```

¹Formally, `cout` is an object of the class `ostream`.
Notice that the space between the word is and the number 21 is caused by the space placed within the string passed to cout. As far as cout is concerned, its input is simply a set of characters that are then sent on to be displayed in the order they are received. Characters from the input are queued, one behind the other, and sent to an output stream for display. Placing a space in the input causes this space to be part of the output stream that is ultimately displayed. For example, the statement

```cpp
cout << "The sum of 12.2 and 15.754 is " << (12.2 + 15.754) << 'n';
```

yields the display

```
The sum of 12.2 and 15.754 is 27.954
```

It should be mentioned that insertion of data into the output stream can be made over multiple lines and is terminated only by a semicolon. Thus, the prior display is also produced by the statement

```cpp
cout << "The sum of 12.2 and 15.754 is "
    << (12.2 + 15.754)
    << 'n';
```

The requirements in using multiple lines are that a string contained within double quotes cannot be split across lines and that the terminating semicolon appear only on the last line. Within a line multiple insertion symbols can be used.

As the last display indicates, floating-point numbers are displayed with sufficient decimal places to the right of the decimal place to accommodate the fractional part of the number. This is true if the number has six or fewer decimal digits. If the number has more than six decimal digits, the fractional part is rounded to six decimal digits, and if the number has no decimal digits, neither a decimal point nor any decimal digits are displayed.\(^2\)

**Formatted Output**\(^3\)

The format of numbers displayed by cout can be controlled by field-width manipulators included in each output stream. Table 16.2 lists the most commonly used manipulators available for this purpose.

<table>
<thead>
<tr>
<th>Manipulator</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>setw(n)</code></td>
<td>Set the field width to ( n ). The setting remains in effect only for the next insertion.</td>
</tr>
<tr>
<td><code>setprecision(n)</code></td>
<td>Set the floating-point precision to ( n ) places. If the display format is exponential or fixed, then the precision indicates the number of digits after the decimal point; otherwise, the precision indicates the total number of displayed digits. The setting remains in effect until the next change.</td>
</tr>
</tbody>
</table>

\(^2\)It should be noted that none of this output is defined as part of the C++ language. Rather, it is defined by a set of classes and routines provided with each C++ compiler.

\(^3\)This topic may be omitted on first reading with no loss of subject continuity.
Table 16.2  Commonly used stream manipulators (continued)

<table>
<thead>
<tr>
<th>Manipulator</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>setiosflags(flags)</td>
<td>Set the format flags (see Table 16.4 for flag settings). The setting remains in effect until the next change.</td>
</tr>
<tr>
<td>setfill(n)</td>
<td>Set the fill character to be used as padding. The default is a space. The setting remains in effect until the next change.</td>
</tr>
<tr>
<td>dec</td>
<td>Set output for decimal display. The setting remains in effect until the next change.</td>
</tr>
<tr>
<td>hex</td>
<td>Set output for hexadecimal display. The setting remains in effect until the next change.</td>
</tr>
<tr>
<td>oct</td>
<td>Set output for octal display. The setting remains in effect until the next change.</td>
</tr>
<tr>
<td>endl</td>
<td>Insert a newline character and flush the stream.</td>
</tr>
</tbody>
</table>

For example, the statement

```cpp
cout << "The sum of 6 and 15 is" << setw(3) << 21 << endl;
```

causes the printout

```
The sum of 6 and 15 is 21
```

The `setw(3)` field-width manipulator included in the stream of data passed to `cout` is used to set the displayed field width. The 3 in this manipulator sets the default field-width for the next number in the stream to be three spaces wide. This field-width setting causes the 21 to be printed in a field of three spaces, which includes one blank and the number 21. As illustrated, integers are right-justified within the specified field. The `endl` manipulator inserts a \n into the stream and flushes the stream buffer.

Field-width manipulators are useful in printing columns of numbers so that the numbers in each column align correctly, as illustrated in Program 16.4.

### Program 16.4

```cpp
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
  cout << '
' << setw(3) << 6
       << '
' << setw(3) << 18
       << '
' << setw(3) << 124
       << "---" << '
' << setw(3) << (6+18+124)
       << endl;
  return 0;
}
```
The output of Program 16.4 is

| 6  
| 18  
| 124  
| 148  

Notice that the field-width manipulator must be included for each occurrence of a number inserted onto the data stream sent to cout and that the manipulator applies only to the insertion of data immediately following it. Also notice that if manipulators are to be included within an output display, the iomanip header field must be included as part of the program. This is accomplished by the preprocessor command #include <iomanip>.

Formatted floating-point numbers require the use of two field-width manipulators. The first manipulator sets the total width of the display, including the decimal point; the second manipulator determines how many digits can be printed to the right of the decimal point. For example, the statement

```cpp
cout << "|" << setw(10) << setiosflags(ios::fixed) << setprecision(3) << 25.67 << "|";
```

causes the printout

| 25.670 |

The bar symbol, |, is used to mark the beginning and end of the display field. The setw manipulator tells cout to display the number in a total field width of ten digits, while the setprecision manipulator tells cout to display a maximum of three digits to the right of the decimal point. The setiosflags manipulator using the ios::fixed flag ensures that the output is displayed in conventional decimal format; that is, as a fixed-point rather than an exponential number.

For all numbers (integer, floating point, and double precision), cout ignores the setw manipulator specification if the total specified field width is too small and allocates enough space for the integer part of the number to be printed. The fractional part of both floating-point and double-precision numbers is displayed up to the precision set with the setprecision manipulator (in the absence of a setprecision manipulator, the precision is set to the default of six decimal places). If the fractional part of the number to be displayed contains more digits than called for in the setprecision manipulator, the number is rounded to the indicated number of decimal places; if the fractional part contains fewer digits than specified, the number is displayed with the fewer digits. Table 16.3 illustrates the effect of various format manipulator combinations. Again, for clarity, the bar symbol, |, is used to clearly delineate the beginning and end of the output fields.

In addition to the setw and setprecision manipulators, a field justification manipulator is also available. As we have seen, numbers sent to cout are normally displayed right-justified in the display field, while strings are displayed left-justified. To alter the default justification for a stream of data, the setiosflags manipulator can be used. For example, the statement

```cpp
cout << "|" << setw(10) << setiosflags(ios::left) << 142 << "|";
```

causes the following left-justified display

| 142 |
As we have previously seen, since data passed to `cout` may be continued across multiple lines, the previous display would also be produced by the statement

```cpp
cout << "|" << setw(10) << setiosflags(ios::left) << 142 << "|
```

As always, the field-width manipulator is in effect only for the next single set of data passed to `cout`. Right-justification for strings in a stream is obtained by the manipulator `setiosflags(ios::right)`.

In addition to the left and right flags that can be used with the `setiosflags` manipulator, other flags may also be used to affect the output. The most commonly used flags for this manipulator are listed in Table 16.4.

**Table 16.3 Effect of format manipulators**

<table>
<thead>
<tr>
<th>Manipulator</th>
<th>Number</th>
<th>Display</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>setw(2)</code></td>
<td>3</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td><code>setw(2)</code></td>
<td>43</td>
<td></td>
<td>43</td>
</tr>
<tr>
<td><code>setw(2)</code></td>
<td>143</td>
<td></td>
<td>143</td>
</tr>
<tr>
<td><code>setw(2)</code></td>
<td>2.3</td>
<td></td>
<td>2.3</td>
</tr>
<tr>
<td><code>setw(5)</code></td>
<td>2.366</td>
<td></td>
<td>2.37</td>
</tr>
<tr>
<td><code>setiosflags(ios::fixed)</code> <code>setprecision(2)</code></td>
<td>42.3</td>
<td></td>
<td>42.3</td>
</tr>
<tr>
<td><code>setiosflags(ios::fixed)</code> <code>setprecision(2)</code></td>
<td>142.364</td>
<td></td>
<td>142.36</td>
</tr>
<tr>
<td><code>setiosflags(ios::fixed)</code> <code>setprecision(2)</code></td>
<td>142.366</td>
<td></td>
<td>142.37</td>
</tr>
<tr>
<td><code>setiosflags(ios::fixed)</code> <code>setprecision(2)</code></td>
<td>142</td>
<td></td>
<td>142</td>
</tr>
</tbody>
</table>

*If either the manipulator flags `ios::fixed` or `ios::scientific` are not in effect, the `setprecision` value indicates the total number of significant digits displayed rather than the number of digits after the decimal point.
For example, the output stream

```cpp
cout << setiosflag(ios::showpoint) 
    << setiosflags(ios::fixed) 
    << setprecision(4);
```

forces the next number sent to the output stream to be displayed with a decimal point and four decimal digits. If the number has fewer than four decimal digits it will be padded with trailing zeros.

In addition to outputting integers in decimal notation, the `ios::oct` and `ios::hex` flags permit conversions to octal and hexadecimal, respectively. Program 16.5 illustrates the use of these flags. Because decimal format is the default display, the `dec` manipulator is not required in the first output stream.

### Table 16.4 Format flags for use with `setiosflags`

<table>
<thead>
<tr>
<th>Flag</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ios::showpoint</code></td>
<td>Always display a decimal point. In the absence of the <code>ios::fixed</code> flag, a numerical value with a decimal point is displayed with a default of 6 significant digits. If the integer part of the number requires more than 6 digits the display will be in exponential notation, unless the <code>ios::fixed</code> flag is in effect. For example, the value 1234567. is displayed as 1.23457e0 unless the <code>ios::fixed</code> flag is in effect. This flag has no effect on integer values.</td>
</tr>
<tr>
<td><code>ios::showpos</code></td>
<td>Display a leading + sign when the number is positive</td>
</tr>
<tr>
<td><code>ios::fixed</code></td>
<td>Display the number in conventional fixed-point decimal notation (that is, with an integer and fractional part separated by a decimal point), and not in exponential notation.</td>
</tr>
<tr>
<td><code>ios::scientific</code></td>
<td>Use exponential display on output</td>
</tr>
<tr>
<td><code>ios::dec</code></td>
<td>Display in decimal format</td>
</tr>
<tr>
<td><code>ios::oct</code></td>
<td>Display in octal format</td>
</tr>
<tr>
<td><code>ios::hex</code></td>
<td>Display in hexadecimal format</td>
</tr>
<tr>
<td><code>ios::left</code></td>
<td>Left-justify output</td>
</tr>
<tr>
<td><code>ios::right</code></td>
<td>Right-justify output</td>
</tr>
</tbody>
</table>

### Program 16.5

```cpp
1 #include <iostream>
2 #include <iomanip>
3 using namespace std;
4 int main() // a program to illustrate output conversions
5 {
6     cout << "The decimal (base 10) value of 15 is " << 15
7     << "\nThe octal (base 8) value of 15 is "
8     << setiosflags(ios::oct) << 15
9     << "\nThe hexadecimal (base 16) value of 15 is "
```
The output produced by Program 16.5 is

The decimal (base 10) value of 15 is 15
The octal (base 8) value of 15 is 17
The hexadecimal (base 16) value of 15 is f

In place of the conversion flags ios::dec, ios::oct, and ios::hex, three simpler manipulators, dec, oct, and hex are provided in <iostream>. These simpler manipulators, unlike their longer counterparts, leave the conversion base set for all subsequent output streams. Using these simpler manipulators, Program 16.5 can be rewritten as:

```cpp
#include <iostream>

int main() // a program to illustrate output conversions
{
    cout << "The decimal (base 10) value of 15 is " << 15
         << "The octal (base 8) value of 15 is " << oct << 15
         << "The hexadecimal (base 16) value of 15 is " << hex << 15
         << endl;
    return 0;
}
```

### Input in C++

Just as the cout object displays a copy of the value stored inside a variable, the cin (pronounced “see in”) object allows the user to enter a value at the terminal. Figure 16.1 illustrates the relationship between these two objects.

For example, a statement such as cin >> num1; is a command to accept data from the keyboard. When a data item is entered, the cin object stores the item into the variable listed after the extraction (“get from”) operator, >>. The program then continues execution with the next statement after the call to cin. The use of cin is illustrated by Program 16.6.
16-20 Introduction to C++

Program 16.6

```cpp
#include <iostream>
int main()
    { float num1, num2, product;
    cout << "Please type in a number: ";
    cin >> num1;
    cout << "Please type in another number: ";
    cin >> num2;
    product = num1 * num2;
    cout << num1 << " times " << num2 << " is " << product << endl;
    return 0;
    }
```

The following sample run was made using Program 16.6:

Please type in a number: 30
Please type in another number: 0.05
30 times 0.05 is 1.5

In Program 16.6, each time `cin` is invoked it is used to store one value into a variable. The `cin` object, however, can be used to enter and store as many values as there are extraction symbols, `>>`, and variables to hold the entered data. For example, the statement

```cpp
    cin >> num1 >> num2;
```

results in two values being read from the terminal and assigned to the variables `num1` and `num2`. If the data entered at the terminal were

```
    0.052 245.79
```

the variables `num1` and `num2` would contain the values 0.052 and 245.79, respectively. Notice that when actually entering numbers such as 0.052 and 245.79, there must be at least one space between the numbers. The space between the entered numbers clearly indicates where one number ends and the next begins. Inserting more than one space between numbers has no effect on `cin`.

The same spacing is also applicable to entering character data; that is, the extraction operator, `>>`, will skip blank spaces and store the next nonblank character in a character variable. For example, in response to the statements

```cpp
    char ch1, ch2, ch3;  // declare three character variables
    cin >> ch1 >> ch2 >> ch3;  // accept three characters
```

the input

```
a    b    c
```
causes the letter a to be stored in the variable ch1, the letter b to be stored in the variable ch2, and the letter c to be stored in the variable ch3. Because a character variable can only be used to store one character the input

abc

can also be used.

The cin extraction operation, like the cout insertion operation, is “clever” enough to make a few data type conversions. For example, if an integer is entered in place of a floating-point or double-precision number, the integer will be converted to the correct data type.4 Similarly, if a floating-point or double-precision number is entered when an integer is expected, only the integer part of the number will be used. For example, assume the following numbers are typed in response to the statement cin >> num1 >> num2 >> num3;, where num1 and num3 have been declared as floating point variables and num2 is an integer variable:

56 22.879 33.923

The 56 will be converted to a floating-point number, 56.0, and stored in the variable num1. The extraction operation continues extracting data from the input stream sent to it, expecting an integer value. As far as cin is concerned, the decimal point after the 22 in the number 22.879 indicates the end of an integer and the start of a decimal number. Thus, the number 22 is assigned to num2. Continuing to process its input stream, cin takes the .879 as the expected next floating-point number and assigns it to num3. As far as cin is concerned, 33.923 is extra input and is ignored. If you do not initially type enough data, however, the cin object will continue to make the computer pause until sufficient data has been entered.

EXERCISES 16.3

1. Using cout, write and execute a C++ program that prints your name on one line, your street address on a second line, and your city, state, and zip code on the third line.

2. Write and execute a C++ program to print out the following verse:

Computers, computers everywhere
as far as I can see
I really, really like these things,
Oh joy, Oh joy for me!

3. Determine the output of the following two programs:
   a. #include <iostream>
      int main() // a program illustrating integer truncation
      {
         cout << "answer1 is the integer " << 9/4

4 Strictly speaking, what comes in from the keyboard is not of any data type, such as int or float, but is simply a sequence of characters. The extraction operation handles the conversion from the character sequence to a defined data type.
16-22 Introduction to C++

```cpp
return 0;
}
b. #include <iostream>
int main() // a program illustrating the % operator
{
    cout << "The remainder of 9 divided by 4 is " << 9 % 4
    << " The remainder of 17 divided by 3 is " << 17 % 3 << \n';
    return 0;
}
4. Write a C++ program that displays the results of the expressions 3.0 * 5.0, 7.1 * 8.3 - 2.2 and 3.2 / (6.1 * 5). Calculate the value of these expressions manually to verify that the displayed values are correct.
5. Determine the errors in each of the following statements:
a. cout << "\n << " 15)
b. cout << "setw(4)" << 33;
c. cout << "setprecision(5)" << 526.768;
d. "Hello World!" >> cout;
e. cout << 47 << setw(6);
f. cout << set(10) << 526.768 << setprecision(2);
6. Determine and write out the display produced by the following statements:
a. cout << "*" << 5 << "|
    b. cout << "*" << setw(4) << 5 << "|"
    c. cout << "*" << setw(4) << 56829 << "|
    d. cout << "*" << setw(5) << setprecision(2) << 5.26 << "|
    e. cout << "*" << setw(5) << setprecision(2) << 5.267 << "|
    f. cout << "*" << setw(5) << setprecision(2) << 53.264 << "|
    g. cout << "*" << setw(5) << setprecision(2) << 534.264 << "|
    h. cout << "*" << setw(5) << setprecision(2) << 534. << "|
7. For the following declaration statements, write a statement using the cin object that will cause the computer to pause while the appropriate data is typed by the user.
a. int firstnum;
b. float grade;
c. double secnum;
d. char keyval;
e. int month years;
    float average;
f. char ch;
    int num1,num2;
    double grade1,grade2;
g. float interest, principal, capital;
    double price,yield;
```
h. char ch, letter1, letter2;
    int num1, num2, num3;
  i. float temp1, temp2, temp3;
    double volts1, volts2;

8. Write a C++ program that displays the following prompts:

   Enter the length of the room:
Enter the width of the room:

   After each prompt is displayed, your program should use a cin object call to accept data from the keyboard for the displayed prompt. After the width of the room is entered, your program should calculate and display the area of the room. The area displayed should be included in an appropriate message and calculated using the equation \( \text{area} = \text{length} \times \text{width} \).

9. Write a C++ program that displays the following prompts:

   Enter a number:
Enter a second number:
Enter a third number:
Enter a fourth number:

   After each prompt is displayed, your program should use a cin object call to accept a number from the keyboard for the displayed prompt. After the fourth number has been entered, your program should calculate and display the average of the numbers. The average should be included in an appropriate message.

10. Write a C++ program that prompts the user to type in a number. Have your program accept the number as an integer and immediately display the integer using a cout object call. Run your program three times. The first time you run the program enter a valid integer number, the second time enter a floating-point number, and the third time enter a character. Using the output display, see what number your program actually accepted from the data you entered.

11. Repeat Exercise 10 but have your program declare the variable used to store the number as a floating-point variable. Run the program four times. The first time enter an integer, the second time enter a decimal number with fewer than six decimal places, the third time enter a number having more than six decimal places, and the fourth time enter a character. Using the output display, keep track of what number your program actually accepted from the data you typed in. What happened, if anything, and why?

12. Repeat Exercise 10 but have your program declare the variable used to store the number as a double-precision variable. Run the program four times. The first time enter an integer, the second time enter a decimal number with fewer than six decimal places, the third time enter a number having more than six decimal places, and the fourth time enter a character. Using the output display, keep track of what number your program actually accepted from the data you typed in. What happened, if anything, and why?
16.3 Function Enhancements in C++

The driving force for the development of C++ was to extend C by providing it with an abstract data type (ADT) capability. An essential component in the construction of ADTs is the design and implementation of suitable operations for each new user-defined data type. In C++ such operations are all constructed as functions. As such, C++ includes four main extensions to C’s standard function capabilities, three of which provide additional ways of using and processing function arguments. These modifications consist of providing default arguments, reference arguments, function overloading, and inline function compilation. This section describes these four enhanced function capabilities.

Default Arguments

A convenient feature of C++ is the ability to use default arguments in a function call. The default argument values are listed in the function prototype and are automatically transmitted to the called function when the corresponding arguments are omitted from the function call. For the example, the function prototype:

```cpp
void example(int, int = 5, float = 6.78);
```

provides default values for the last two arguments. If any of these arguments are omitted when the function is actually called, the C++ compiler will supply these default values. Thus, all of the following function calls are valid:

```cpp
example(7, 2, 9.3) // no defaults used
example(7, 2) // same as example(7, 2, 6.78)
example(7) // same as example(7, 5, 6.78)
```

Four rules must be followed when using default parameters. The first is that default values can only be assigned in the function prototype. The second is that if any argument is given a default value in the function prototype, all arguments following it must also be supplied with default values. The third rule is that if one argument is omitted in the actual function call, then all arguments to its right must also be omitted. These latter two rules make it clear to the C++ compiler which arguments are being omitted and permits the compiler to supply correct default values for the missing arguments. The last rule specifies that the default value used in the function prototype may be an expression consisting of both constants and previously declared variables. If such an expression is used, it must pass the compiler’s check for validly declared variables, even though the actual value of the expression is evaluated and assigned at run time.

Default arguments are extremely useful when extending an existing function to include more features that require additional arguments. Adding the new arguments to the right of the existing arguments and providing each new argument with a default value permits all existing function calls to remain as they are. Thus, the effect of the new changes are conveniently isolated from existing code in the program.

Reusing Function Names (Overloading)

In C each function requires its own unique name. While in theory this makes sense, in practice it can lead to a profusion of function names, even for functions that perform essentially the same operations. For example, consider determining and displaying the absolute value of a number. If the number passed into the function can be either an integer,
a long integer, or a double-precision value, three distinct functions must be written to correctly handle each case. In C, we would give each of these functions a unique name, such as intabs(), longabs(), and dblabs(), respectively, having the function prototypes:

```c
void intabs(int);
void longabs(long);
void dblabs(double);
```

Clearly, each of these three functions performs essentially the same operation, differing only in data type. C++ provides the capability of using the same function name for more than one function, which is referred to as **function overloading**. The only requirement in creating more than one function with the same name is that the compiler must be able to determine which function to use based on the data types of the parameters (not the data type of the return value, if any). For example, consider the three following functions, all named `showabs()`.

```c
void showabs(int x) // display the absolute value of an integer
{
 if ( x < 0 )
   x = −x;
 cout << "The absolute value of the integer is " << x << endl;
}

void showabs(long x) // display the absolute value of a long integer
{
 if ( x < 0 )
   x = -x;
 cout << "The absolute value of the long integer is " << x << endl;
}

void showabs(double x) // display the absolute value of a double
{
 if ( x < 0 )
   x = -x;
 cout << "The absolute value of the float is " << x << endl;
}
```

Which of the three functions named `showabs()` is actually called depends on the argument types supplied at the time of the call. Thus, the function call `showabs(10)`; would cause the compiler to use the function named `showabs()` that expects an integer argument, and the function call `showabs(6.28)`; would cause the compiler to use the function named `showabs()` that expects a double-valued argument.5

Notice that overloading a function's name simply means using the same name for more than one function. Each function that uses the name must still be written and exists as a separate entity. The use of the same function name does not require that the code within the functions be similar, although good programming practice dictates that functions with the same name should perform essentially the same operation. All that is formally required in using the same function name is that the compiler can distinguish which function to select based on the data types of the arguments when the function is called. Overloaded functions are extremely useful for creating multiple constructor functions, a topic that is presented in

---

5This is accomplished by **name mangling**. This is a process whereby the function name generated by the C++ compiler differs from the function name used in the source code. The compiler appends information to the source code function name depending on the type of data being passed, and the resulting name is said to be a mangled version of the source code name.
16.5 For other functions, such as our `showabs()` example, where all that is different about the overloaded functions is the parameter types, a better programming solution is to create a C++ function template.

**Function Templates**

A function template is a single, complete function that serves as a model for a family of functions. Which function from the family that is actually created depends on subsequent function calls. To make this more concrete, consider a function template that computes and displays the absolute value of a passed argument. An appropriate function template is

```cpp
template <class T>
void showabs(T number)
{
    if (number < 0)
        number = -number;
    cout << "The absolute value of the number "
        << " is " << number << endl;
    return
}
```

For the moment, ignore the first line `template <class T>`, and look at the second line, which consists of the function header `void showabs(T number)`. Notice that this header line has the same syntax that we have been using for all of our function definitions, except for the `T` in place of where a data type is usually placed. For example, if the header line were `void showabs(int number)`, you should recognize this as a function named `showabs` that expects one integer argument to be passed to it, and that returns no value. Similarly, if the header line were `void showabs(float number)`, you should recognize it as a function that expects one floating-point argument to be passed when the function is called.

The advantage in using the `T` within the function template header line is that it represents a general data type that is replaced by an actual data type, such as `int`, `float`, `double`, etc., when the compiler encounters an actual function call. For example, if a function call with an integer argument is encountered, the compiler will use the function template to construct the code for a function that expects an integer parameter. Similarly, if a call is made with a floating-point argument, the compiler will construct a function that expects a floating-point parameter. As a specific example of this, consider Program 16.7.

**Program 16.7**

```cpp
#include <iostream>
using namespace std;
template <class T>
void showabs(T number)
{
    if (number < 0)
        number = -number;
    cout << "The absolute value of the number "
        << " is " << number << endl;
    return
}
```

This topic may be omitted on first reading with no loss of subject continuity.
7 if (number < 0)  
8     number = -number;  
9 cout << "The absolute value of the number is " 
10     << number << endl;  
11  
12 return;  
13 }

int main()
15 {
17  int num1 = -4;  
18  float num2 = -4.23;  
19  double num3 = 4.23456;  
20  
21  showabs(num1);  
22  showabs(num2);  
23  showabs(num3);  
24  
25  return 0;  
26 }

First notice the three function calls made in the main() function shown in Program 16.7, which call the function showabs() with an integer, float, and double value, respectively. Now review the function template for showabs() and let us consider the first line, template <class T>. This line, which is called a template prefix, is used to inform the compiler that the function immediately following is a template that uses a data type named T. Within the function template, T is used in the same manner as any other data type, such as int, float, double, etc. Then, when the compiler encounters an actual function call for showabs(), the data type of the argument passed in the call is substituted for T throughout the function. In effect, the compiler creates a specific function, using the template, that expects the argument type in the call. Because Program 16.7 makes three calls to showabs(), each with a different argument data type, the compiler creates three separate showabs() functions. The compiler knows which function to use based on the arguments passed at the time of the call. The output displayed when Program 16.7 is executed is

The absolute value of the number is 4
The absolute value of the number is 4.23
The absolute value of the number is 4.23456

The letter T used in the template prefix template <class T> is simply a placeholder for a data type that is defined when the function is invoked. Any letter or non-keyword identifier can be used instead. Thus, the showabs() function template could have been defined as:

template <class DTYPE>
void showabs(DTYPE number)  
{
if (number < 0)
    number = -number;
    cout << "The absolute value of the number is "
        << number << endl;
    return;
}

In this regard, it is sometimes simpler and clearer to read the word `class` in the template prefix as the words `data type`. Thus, the template prefix `template <class T>` can be read as "we are defining a function template that has a data type named T." Then, within both the header line and body of the defined function, the data type T (or any other letter or identifier defined in the prefix) is used in the same manner as any built-in data type, such as `int`, `float`, `double`, etc.

Now suppose we want to create a function template to include both a return type and an internally declared variable. For example, consider the following function template:

template <class T> // template prefix
t T abs(T value) // header line
{
    T absnum; // variable declaration
    if (value < 0)
        absnum = -value;
    else
        absnum = value;
    return absnum;
}

In this template definition, we have used the data type T to declare thee items: the return type of the function, the data type of a single function parameter named `value`, and one variable declared within the function. Program 16.8 illustrates how this function template could be used within the context of a complete program.

Program 16.8

1 #include <iostream>
2 using namespace std;
3 template <class T> // template prefix
4 T abs(T value) // header line
5 {
6    T absnum; // variable declaration
7    if (value < 0)
8        absnum = -value;
9    else
10        absnum = value;
11    return absnum;
12
In the first call to \texttt{abs(0 made within \texttt{main()}, an integer value is passed as an argument. In this case, the compiler substitutes an \texttt{int} data type for the \texttt{T} data type in the function template and creates the following function:

\begin{verbatim}
int showabs(int value) // header line
{
    int absnum; // variable declaration
    if (value < 0)
        absnum = -value;
    else
        absnum = value;
    return (absnum);
}
\end{verbatim}

Similarly, in the second and third function calls, the compiler creates two more functions, one in which the data type \texttt{T} is replaced by the keyword \texttt{float}, and one in which the data type \texttt{T} is replaced by the keyword double. The output produced by Program 16.8 is

\begin{verbatim}
The absolute value of -4 is 4
The absolute value of -4.23 is 4.23
The absolute value of -4.23456 is 4.23456
\end{verbatim}

The value of using the function template is that one function definition has been used to create three different functions, each of which uses the same logic and operations but operates on different data types.

Finally, although both Programs 16.7 and 16.8 define a function template that uses a single placeholder data type, function templates with more than one data type can be defined. For example, the template prefix

\texttt{template <class DTYPE1, class DTYPE2, class DTYPE3>
can be used to create a function template that requires three different data types. As before, within the header and body of the function template, the data types DTYPE1, DTYPE2, and DTYPE3 are used in the same manner as any built-in data type, such as an int, float, double, etc. Additionally, as noted previously, the names DTYPE1, DTYPE2, and DTYPE3 can be any non-keyword identifier. Conventionally, the letter T followed by zero or more digits would be used, such as T, T1, T2, T3, etc.

**Inline Functions**

Calling a function places a certain amount of overhead on a computer: this consists of placing argument values in a reserved memory region that the function has access to (this memory region is called the stack), passing control to the function, providing a reserved memory location for any returned value (again, the stack region of memory is used for this purpose), and, finally, returning to the proper point in the calling program. Paying this overhead is well justified when a function is called many times, because it can significantly reduce the size of a program. Rather than repeating the same code each time it is needed, the code is written once, as a function, and called whenever it is needed.

For small functions that are not called many times, however, paying the overhead for passing and returning values may not be warranted. It still would be convenient, though, to group repeating lines of code together under a common function name and have the compiler place this code directly into the program wherever the function is called. This capability is provided by inline functions.

Telling the C++ compiler that a function is inline causes a copy of the function code to be placed in the program at the point the function is called. For example, consider the function tempvert() defined in Program 16.9. Since this is a relatively short function, it is an ideal candidate to be an inline function. As shown in the program, making this or any other function inline simply requires placing the reserved word inline before the function name and defining the function before any calls are made to it.

**Program 16.9**

```cpp
#include <iostream>
using namespace std;
inline double tempvert(double inTemp) // an inline function
{
  return((5.0/9.0)*(inTemp-32.0));
}

int main()
{
  int count;          // start of declarations
  double fahren;
```

---

7This topic may be omitted on first reading without loss of subject continuity.
Observe in Program 16.9 that the inline function is placed ahead of any calls to it. This is a requirement of all inline functions and obviates the need for a function prototype. Because the function is now an inline one, its code is be expanded directly into the program wherever it is called.

The advantage of using an inline function is an increase in execution speed. Because the inline function is directly expanded and included in every expression or statement calling it, there is no loss of execution time due to the call and return overhead required by a noninline function. The disadvantage is the increase in program size when an inline function is called repeatedly. Each time an inline function is referenced, the complete function code is reproduced and stored as an integral part of the program. A noninline function, however, is stored in memory only once. No matter how many times the function is called, the same code is used. Therefore, inline functions should only be used for small functions that are not extensively called in a program.

Reference Parameters

The standard method of making a function call in C is to pass parameters by value. The same is true for function calls in C++. In both languages, when a call by reference is required, pointer parameters can be used. Such parameters, as we have seen, permit the called function to directly access the calling function’s variables. In addition, C++ provides an alternative call-by-reference capability. This alternative method is accomplished by using a new type of parameter, called a reference parameter, that can be used in place of pointer arguments.

Essentially, a reference parameter is a pointer parameter with restricted capabilities that is simpler to use because it hides a lot of internal pointer manipulations from the programmer. Before seeing how reference parameters are used, however, it will be helpful to first introduce the concept of a reference variable.

Reference variables provide a means of giving a previously declared variable an additional name. This is accomplished using a reference declaration, which has the form

```cpp
data-type& new-name = existing-name;
```

For example, the reference declaration

```cpp
float& sum = total;
```

---

8 Pointer arithmetic is not provided for reference parameters and variables.
16-32 Introduction to C++

equates the name `sum` to the name `total`—both now refer to the same variable, as illustrated in Figure 16.2.

Once another name has been established for a variable using a reference declaration, the new name, which is referred to as an alias, can be used in place of the original name. For example, consider Program 16.10.

Program 16.10

```cpp
#include <iostream>
using namespace std;

int main()
{
  float total = 20.5; // declare and initialize total
  float& sum = total; // declare another name for total

  cout << "sum = " << sum << endl;
  sum = 18.6; // this changes the value in total
  cout << "total = " << total << endl;

  return 0;
}
```

The following output is produced by Program 16.10:

```
sum = 20.5
total = 18.6
```

Because the variable `sum` is simply another reference to the variable `total`, it is the value stored in `total` that is obtained by the first call to `cout` in Program 16.10. Changing the value in `sum` then changes the value in `total`, which is displayed by the second call to `cout` in Program 16.10.

In constructing references, two considerations must be kept in mind. First, the reference should be of the same data type as the variable to which it refers. For example, the sequence of declarations

```cpp
int num = 5;
double& numref = num;
```

does not equate `numref` to `num`, because they are not of the same data type. Rather, because the compiler cannot correctly associate the reference with a variable, it creates an unnamed variable of the reference type first, and then references this unnamed variable with the reference variable. Such unnamed variables are called anonymous variables. For example, consider Program 16.11, which illustrates the effect of creating an anonymous variable.

---

9Knowledgeable C programmers should not confuse the use of the ampersand symbol, `&`, in a reference declaration with the address operator or with the use of a reference variable as an address. A reference variable simply equates two variable names.
Program 16.11

```cpp
#include <iostream>
using namespace std;

int main()
{
    int num = 10;
    float& numref = num; // this does not equate numref to num
    // instead, it equates numref to an
    // anonymous floating-point variable
    numref = 23.6;
    cout << "The value of num is " << num << endl
    << "The value of numref is " << numref << endl;
    return 0;
}
```

The output produced by Program 16.11 is

The value of num is 10
The value of numref is 23.6

Notice that the value of num is not affected by the value stored in numref. This is because numref could not be created as a reference for num; rather, it is another name for an unnamed (anonymous) floating-point variable that can be reached only by using the reference name numref.

Just as declaring a reference to an incorrect data type produces an anonymous variable, so does equating a reference to a constant. For example, the declaration

```cpp
int& val = 5; // an anonymous variable is created
```

creates an anonymous variable with the number 5 stored in it. The only way to access this variable is by the reference name. Clearly, creating references to anonymous variables should be avoided. Once a reference name has been equated to either a legal or an anonymous variable, the reference cannot be changed to refer to another variable.
As with all declaration statements, multiple references may be declared in a single statement as long as each reference name is preceded by the ampersand symbol. Thus, the declaration

```cpp
float& sum = total, & mean = average;
```

creates two reference variables named `sum` and `average`.10

### Passing and Using References

The real usefulness of reference variables is in their ability to act as function parameters, because they provide a simplified means of providing a pass-by-reference capability. As always, in exchanging data between two functions we must be concerned with both the sending and receiving sides of the data exchanged. From the sending side, however, calling a function and passing a reference is exactly the same as calling a function and passing a value: the called function is summoned into action by giving its name and a list of arguments. For example, the statement

```cpp
calc(firstnum, secnum, thirdnum, sum, product);
```

calls the function named `calc()` and passes five arguments to it. Let us now write the `calc()` function so that it receives direct access to the last two variables, `sum` and `product`, which we will assume to be floating-point variables.

The parameter declarations `float &total` and `float &product` can be used to declare two reference parameters. Here `total` and `product` are declared as reference parameters to floating-point variables. As always, the choice of the parameter names `total` and `product` is up to the programmer. Including these declarations within the parameter list for `calc()`, and assuming that the function returns no value (`void`), a valid function header for `calc()` becomes

```cpp
void calc(float num1, float num2, float num3, float& total, float& product)
```

This function header includes five parameters, which include three floating-point parameters and two reference parameters. As we will see in a moment, the two reference parameters can be employed for returning values much as pointer parameters are, but using a much simpler notation.

Assume that the purpose of the `calc()` function is to accept three values, compute the sum and product of these values, and return the computed results to the calling routine. The following function provides this capability.

```cpp
void calc(float num1, float num2, float num3, float& total, float& product)
{
    total = num1 + num2 + num3;
    product = num1 * num2 * num3;
    return;
}
```

As we have seen, this function has five parameters named `num1`, `num2`, `num3`, `total`, and `product`, of which only the last two are declared as references. Within the function only the last two parameters are altered. The value of the fourth parameter, `total`, is calculated as the sum of the first three parameters and the last parameter, `product`, is computed as the product of the parameters `num1`, `num2`, and `num3`. Program 16.12 includes this function in a complete program.

---

10Reference declarations may also be written in the form `dataType &newName = existing-name;`, where a space is placed between the data type and ampersand symbol. This form is not used much, however, because it does not clearly distinguish reference variable notation from that used in taking the address of a variable.
Program 16.12

```cpp
#include <iostream>
using namespace std;

int main()
{
    float firstnum, secnum, thirdnum, sum, product;
    void calc(float, float, float, float&, float&); // prototype
    
    cout << "Enter three numbers: ";
    cin >> firstnum >> secnum >> thirdnum;
    
calc(firstnum, secnum, thirdnum, sum, product); // function call
    
cout << "The sum of the numbers is: " << sum;
    cout << "The product of the numbers is: " << product << endl;
    
    return 0;
}

void calc(float num1, float num2, float num3, float& total, float& product)
{
    total = num1 + num2 + num3;
    product = num1 * num2 * num3;
    return;
}
```

Within `main()`, the function `calc()` is called using the five arguments `firstnum`, `secnum`, `thirdnum`, `sum`, and `product`. As required, these arguments agree in number and data type with the parameters declared by `calc()`. Of the five arguments passed, only `firstnum`, `secnum`, and `thirdnum` have been assigned values when the call to `calc()` is made. The remaining two arguments have not been initialized and will be used to receive values back from `calc()`. Depending on the compiler used in compiling the program, these arguments initially contain either zeros or "garbage" values. Figure 16.3 illustrates the relationship between argument and parameter names and the values they contain after the return from `calc()`.

In calling the `calc()` function in Program 16.12 it is important to understand the connection between the argument names, `sum` and `product`, used in the function call and the parameter names used in the function header. As shown in Figure 16.3, both refer to the same data items. The significance of this is that the values in the arguments (`sum` and `product`) can now be altered from within `calc()` by using the parameter names (`total` and `product`). This, of course, is the same correspondence provided by pointers, but with simpler notation.
Once `calc()` is called, it uses its first three parameters to calculate values for `total` and `product` and then returns control to `main()`. Because of the order of its calling arguments, `main()` knows the values calculated by `calc()` as `sum` and `product`, which are then displayed. Following is a sample run of Program 16.12.

Enter three numbers: 2.5 6.0 10.0

The sum of the entered numbers is: 18.5
The product of the entered numbers is: 150

In using reference parameters, two cautions need to be mentioned. The first is that reference parameters cannot be used to change constants. For example, calling `calc()` with five constants, such as in the call `calc(3.0, 4.0, 5.0, 6.0, 7.0)` passes five constants to the function. Although `calc()` may execute, it does not change the values of these constants.\(^\text{11}\)

The second caution to note is that a function call itself gives no indication that the called function will be using reference parameters. The convention in C++ is to make passes by value rather than passes by reference, precisely to limit a called function's ability to alter variables in the called function. This convention should be adhered to whenever possible, which means that reference parameters should be used only in very restricted situations that actually require multiple return values. The `calc()` function included in Program 16.12, while useful for illustrative purposes, could also be written as two separate functions, each returning a single value.

\(^{11}\)Most compilers will catch this error.
References versus Pointers

At this point you might be wondering when to use a reference and when to use a pointer. To answer this question we first compare the two in a rather simple case. For example, consider the statements:

```c
int b; // b is an integer variable
int &a = b; // a is a reference variable to b
a = 10; // this changes b's value to 10
```

Here, `a` is declared as a reference variable. Because the compiler knows, from the declaration, that `a` is a reference variable, it automatically equates the address of `b` (rather than the contents of `b`) to the address of `a` in the declaration statement. Finally, in the statement `a = 10;` the compiler uses the reference variable to change the value stored in `b` to 10. The advantage of using the reference is that an automatic indirect access of `b`'s value is performed without the need for explicitly using the indirection symbol, *`. This type of access is referred to as an automatic dereference.

Implementing this same correspondence between `a` and `b` using pointers is done by the following sequence of instructions:

```c
int b; // b is an integer variable
int *a = &b; // a is a pointer - store b's address in a
*a = 10; // this changes b's value to 10
```

Here, `a` is defined as a pointer that is initialized to store the address of `b`. Thus, `*a`, which can be read as either “the variable whose address is in `a`” or “the variable pointed to by `a`,” is `b`, and the expression `*a = 10` changes `b`'s value to 10. Notice in the pointer case that the stored address can be altered to point to another variable; in the reference case the reference variable cannot be altered to refer to any variable except the one to which it is initialized. To use the address in `a`, an explicit dereference must be specified using the indirection operator. Notationally, then, the reference variable is simpler to use because it does not require the indirection operator for dereferencing.

Thus, for simple cases, such as this example, the use of references over pointers is easier and clearly preferred. The same is true when a pass by reference is required—C++'s reference parameters provide a simpler notational interface and are easier to use than equivalent pointer parameters. For other situations, such as dynamically allocating memory or using alternatives to array notation where pointer arithmetic is needed, pointers are required.

EXERCISES 16.3

1. Write a function named `check()` that has three parameters. The first parameter should accept an integer number, the second parameter a floating-point number, and the third parameter a character. The default values for each passed argument should be 100, 22.4, and 'a', respectively. The function should simply display the values of its passed arguments.

Check that your `check` function works properly by making four calls to it; the first call should be made with three values passed to the function, the second call with two values, the third call with one value, and the last call with no passed values.
2. Write a function named `seconds()` that determines the total number of seconds contained in the hours and minutes arguments passed to the function. Both arguments should be integer values with defaults of 0. Check that your function works correctly by making three calls to it; the first call should be made with both an hour and a minute value, the second call with just an hour value, and the third call with no passed values.

3. Write parameter declarations for:
   a. a parameter named `amount` that will be a reference to a floating-point value
   b. a parameter named `price` that will be a reference to a double-precision number
   c. a parameter named `minutes` that will be a reference to an integer number
   d. a parameter named `key` that will be a reference to a character
   e. a parameter named `yield` that will be a reference to a double-precision number

4. Three integer arguments are to be used in a call to a function named `time()`. Write a suitable function header for `time()`, assuming that `time()` accepts these integers as the reference parameters `sec`, `min`, and `hours`, respectively, and returns no value to its calling function.

5. Rewrite the `findMax()` function in Program 6.1 so that a reference to the variable `maxnum`, declared in `main()`, is accepted by `findMax()`.

6. Write a function named `change()` that has a floating-point parameter and four integer reference parameters named `quarters`, `dimes`, `nickels`, and `pennies`, respectively. The function is to consider the floating-point passed value as a dollar amount and convert the value into an equivalent number of quarters, dimes, nickels, and pennies, respectively. Using the reference parameters the function should directly alter the respective arguments in the calling function.

7. Write a function named `time()` that has an integer parameter named `seconds` and three integer reference parameters named `hours`, `min`, and `sec`. The function is to convert the passed number of seconds into an equivalent number of hours, minutes, and seconds. Using the reference parameters the function should directly alter the respective arguments in the calling function.

---

16.4 Abstract Data Types in C++ (Classes)

In C++ an abstract data type is referred to as a class. Construction of a class is inherently easy, and we already have all the necessary tools in structures and functions. What C++ provides is a mechanism for packaging a structure and functions together in a self-contained unit. In this section we show how this is done.

**Class Construction**

Unlike a data structure, which we have used to define data, a class defines both data and functions. This is usually accomplished by constructing a class in two parts, a declaration section and an implementation section. As illustrated in Figure 16.4, the declaration section

---

12The structures presented in Chapter 12 can be expanded to include functions. In this text, however, we will only use classes for this purpose.
declares both the data type and functions of the class. The implementation section is then used to define the function whose prototypes have been declared in the declaration section.\footnote{This separation into two parts is not mandatory; the implementation can be included within the declaration section if inline functions are used.}

Variables and functions listed in the class declaration section are referred to as both \textit{data members} and \textit{class members}. The data members, as illustrated in Figure 16.4, are also referred to as \textit{instance variables}. The functions are referred to as \textit{member functions}. A member function name may not be the same as a data member name.

\begin{verbatim}
// class declaration section
class ClassName
{
    variable declarations  // instance variables
    // instance variables
    function prototypes
};
// class implementation section
function definitions

Figure 16.4 Format of a class definition
\end{verbatim}

Consider the following definition of a class named \texttt{Date}.

\begin{verbatim}
//--- class declaration section class Date
{
    private:     // notice the colon after the word private
        int month;  // a data member
        int day;    // a data member
        int year;   // a data member
    public:      // again; notice the colon here
        Date(int, int, int);   // a member function - the constructor
        void setdate(int = 7, int = 4, int = 2005); // a member function
        void showdate();        // a member function
}
//--- class implementation section
Date::Date(int mm, int dd, int yyyy)
{
    month = mm;
    day = dd;
    year = yyyy;
}
void Date::setdate(int mm, int dd, int yyyy)
{
    month = mm;
    day = dd;
    year = yyyy;
    return;
}
void Date::showdate()
\end{verbatim}
Because this definition may initially look overwhelming, first simply notice that it does consist of two sections—a declaration section and an implementation section. Now consider each of these sections individually.

A class declaration section consists of variable declarations and function prototypes. A commonly used form for this section is

```cpp
class Name
{
    private:
        a list of variable declarations
    public:
        a list of function prototypes
};
```

Notice that this format is followed by our Date class, which for convenience we have listed below with no internal comments:

```cpp
//--- class declaration section
class Date
{
    private:
        int month;
        int day;
        int year;
    public:
        Date(int = 7, int = 4, int = 2005);
        void setdate(int, int, int);
        void showdate();
}; // this is a declaration—don't forget the semicolon
```

The name of this class is Date. Although the initial capital letter is not required, it is conventionally used to designate a class. The body of the declaration section, which is enclosed within braces, consists of variable and function declarations. In this case, the variables month, day, and year are declared as integers and three functions named Date(), setdate(), and showdate() are declared via prototypes. The keywords private and public are access specifiers that define access rights. The private keyword specifies that the class members following, in this case the variables month, day, and year, cannot be accessed from outside of the class and may only be accessed by other class functions (or friend functions, as will be discussed in Section 17.2). Thus, for example, a statement made outside of the class, such as

```cpp
birth.month = 7;
```
where birth is a variable of type Date, is illegal with private class members. The purpose of the private designation is specifically to force all accesses to private data through the provided member functions. Once a class category such as private is designated, it remains in force until a new category is listed.

```
return-type ClassName::functionName (parameter list)
{
  function body
}
```

**Figure 16.5** Format of a member function

As with the equivalent C functions (see Section 16.1) we have chosen to store a date using three integers, one for the month, day, and year, respectively. In addition, we will require that the year be stored as a four-digit number. Doing so ensures that we know what the year 2007, for example, refers to, whereas a two-digit value such as 07 could refer to either 1907 or 2007. Four-digit years also permit an easy determination of the difference between two years by simply subtracting the earlier from the latter year. This is not possible for two-digit years that reside in different centuries.

Following the private class data members, the function prototypes listed in the Date class have been declared as public. This means that these class functions can be called from outside of the class. In general, all class functions should be public; as such they furnish capabilities to manipulate the class variables from outside of the class. For our Date class, we have initially provided three functions, named Date(), setdate(), and showdate(). Notice that one of these member functions has the same name, Date, as the class name. This particular function is referred to as a constructor function, and it has a specially defined purpose: it can be used to initialize class data members with values. The default argument values that are used for this function are the numbers 7, 4, and 2005, which, as we will shortly see, are used as the default month, day, and year values, respectively. The one point to notice here is that the default year is correctly represented as a four-digit integer that retains the century designation. Also notice that the constructor function has no return type, which is a requirement for this special function. The two remaining functions declared in our declaration example are setdate() and showdate(), both of which have been declared as returning no value (void). In the implementation section of the class, these three member functions will be written to permit initialization, assignment, and display capabilities, respectively.

The implementation section of a class is where the member functions declared in the declaration section are written. Figure 16.5 illustrates the general form of functions included in the implementation section. This format is correct for all functions except the constructor, which, as we have stated, has no return type.

As shown in Figure 16.5, member functions have the same format as all user-written C++ functions, with the addition of the class name and scope resolution operator, ::, which is
used to identify the function as a member of a particular class. Let us now reconsider the implementation section of our Date class, which is repeated below for convenience:

```cpp
//--- class implementation section

Date::Date(int mm, int dd, int yyyy)
{
    month = mm;
    day = dd;
    year = yyyy;
}

void Date::setdate(int mm, int dd, int yyyy)
{
    month = mm;
    day = dd;
    year = yyyy;
    return;
}

void Date::showdate()
{
    cout << "The date is " << setfill('0')
    << setw(2) << month << '/'
    << setw(2) << day << '/'
    << setw(2) << year % 100
    << endl;
    return;
}
```

Notice that the first function in this implementation section has the same name as the class, which makes it a constructor function. Hence, it has no return type. The Date:: included at the beginning of the function header line identifies this function as a member of the Date class. The rest of the header line, Date(int mm, int dd, int yyyy), defines the function as having three integer parameters. The body of this function simply assigns the data members month, day, and year with the values of the parameters mm, dd, and yyyy, respectively.

The next function header line

```cpp
void Date::setdate(int mm, int dd, int yyyy)
```

defines this as the setdate() function belonging to Date class (Date::). This function returns no value (void) and uses three integer parameters, mm, dd, and yyyy. In a manner similar to the Date() function, the body of this function assigns the data members month, day, and year with the values of these parameters. In a moment we will see the difference between Date() and setdate().

Finally, the last function header line in the implementation section defines a function named showdate(). This function has no parameters, returns no value, and is a member of the Date class. The body of this function, however, needs a little more explanation. Although we have chosen to internally store all years as four-digit values that retain century information, users are accustomed to seeing dates where the year is represented as a two-digit
value, such as 12/15/99. To display the last two digits of the year value, the expression `year % 100` can be used. For example, if the year is 1999, the expression `1999 % 100` yields the value 99, and if the year is 2005, the expression `2005 % 100` yields the value 5. Notice that if we had used an assignment such as `year = year % 100;` we would actually be altering the stored value of `year` to correspond to the last two digits of the year. Because we want to retain the year as a four-digit number, we must be careful to only manipulate the displayed value using the expression `year % 100` within the `cout` stream. The `setw` and `setfill` manipulators are used to ensure that the displayed values correspond to conventionally accepted dates. For example, the date March 9, 2006, should appear as either 3/9/06 or 03/09/06. The `setw` manipulator forces each value to be displayed in a field width of 2. Because this manipulator only remains in effect for the next insertion, we have included it before the display of each date value. As the `setfill` manipulator, however, remains in effect until the fill character is changed, we only have to include it once.\footnote{This type of information is easily obtained using Windows' online Help facility.} We have used the `setfill` manipulator here to change the fill character from its default of a blank space to the character 0. Doing this ensures that a date such as December 9, 2006, will appear as 12/09/06 and not 12/ 9/ 6.

To see how our `Date` class can be used within the context of a complete program, consider Program 16.13. To make the program easier to read it has been shaded in lighter and darker areas. The lighter area contains the class declaration and implementation sections we have already considered. The darker area contains the header and `main()` function. For convenience, we retain this shading convention for all programs using classes.\footnote{This shading is not accidental. In practice, the lightly shaded region containing the class definition would be placed in a separate file. A single `#include` statement would then be used to include this class definition into the program. Thus, the final program would consist of the two darker-shaded regions illustrated in Program 16.13 with the addition of one more `#include` statement in the first region.}

### Program 16.13

```cpp
#include <iostream>
#include <iomanip>
using namespace std;

// class declaration section
class Date
{
    private:
        int month;
        int day;
        int year;
    public:
        Date(int = 7, int = 4, int = 2005); // constructor
        void setdate(int, int, int); // member function to assign a date
        void showdate(); // member function to display a date
};
```

The declaration and implementation sections contained in the lightly shaded region of Program 16.13 should look familiar, as they contain the class declaration and implementation sections we have already discussed. Notice, however, that this region only declares the class; it does not create any variables of this class type. This is true of all C++ types, including the built-in types such as integer and floating point. Just as a variable of an integer type must be defined, variables of a user-defined class must also be defined. Variables defined to be of user-defined class are referred to as objects.
Thus, the first statement in Program 16.13’s main() function, contained in the darker area, defines three objects, named a, b, and c, to be of class type Date. In C++, whenever a new object is defined, memory is allocated for the object and its data members are automatically initialized. This is done by an automatic call to the class constructor function. For example, consider the definition Date a, b, c(4,1,1998); contained in main(). When the object named a is defined the constructor function Date() is automatically called. Because no arguments have been assigned to a, the default values of the constructor function are used, resulting in the initialization:

a.month = 7
a.day = 4
a.year = 2005

Notice the notation that we have used here. It consists of an object name and an attribute name separated by a period. This is the standard syntax for referring to an object’s attribute, namely,

objectName.attributeName

where objectname is the name of a specific object and attributeName is the name of a data member defined for the object’s class. This should be familiar to you because it is the same notation that is used in accessing structure members.

Thus, the notation a.month = 7 refers to the fact that object a’s month data member has been set to the value 7. Similarly, the notation a.day = 4 and a.year = 2005 refer to the fact that a’s day and year data members have been set to the values 4 and 2005, respectively. In the same manner, when the object named b is defined, the same default arguments are used, resulting in the initialization of b’s data members as:

b.month = 7
b.day = 4
b.year = 2005

The object named c, however, is defined with the arguments 4, 1, and 1998. These three arguments are passed into the constructor function when the object is defined, resulting in the initialization of c’s data members as:

c.month = 4
c.day = 1
c.year = 1998

The next statement in main(), b.setdate(12,25,2006), calls b’s setdate() function, which assigns the argument values 12, 25, 2006 to b’s data members, resulting in the assignment:

b.month = 12
b.day = 25
b.year = 2006
Finally, the last three statements call a, b, and c’s `showdate()` function. The first call results in the display of a’s data values, the second call in the display of b’s data value, and the third call in the display of c’s data values. Thus, the output of Program 13.11 is:

```
The date is 07/04/05
The date is 12/25/06
The date is 04/01/98
```

Notice that a statement such as `cout << a;` is invalid within `main()` because `cout` does not know how to handle an object of class `Date`. Thus, we have supplied our class with a function that can be used to access and display an object’s internal values.

**Terminology**

There is sometimes confusion between the terms “classes,” “objects,” and other terminology associated with object-oriented programming. Let us take a moment to clarify and review the terminology.

A **class** is a programmer-defined abstract data type out of which objects can be created. **Objects** are created from classes; they have the same relationship to classes as variables do to C++’s built-in data types. For example, in the declaration:

```
int a;
```

a is said to be a variable, while in Program 16.13’s declaration:

```
Date a;
```

a is said to be an object. If it helps you to initially think of an object as a variable, do so.

Objects are also referred to as **instances** of a class and the process of creating a new object is frequently referred to as an **instantiation** of the object. Each time a new object is instantiated (created), a new set of data members belonging to the object is created.\(^{19}\)

Individually, each data member represents an attribute of interest that is modeled by the class. The particular values contained in these data members for each object determine the object’s **state**.

Seen in this way, a class can be thought of as a blueprint out of which particular instances (objects) can be created. Each instance (object) of a class will have its own set of particular values for the set of data members specified in the class declaration section.

In addition to the data types allowed for an object, a class also defines **behavior**, that is, the operations that are permitted to be performed on an object’s data members. Users of the object need to know what these functions can do and how to activate them through function calls, but they do not need to know how the operation is done. The actual implementation details of an object’s operations are contained in the class implementation, which can be hidden from the user. Other names for the operations defined in a class implementation section are **procedures**, **functions**, **services**, and **methods**. We will use these terms interchangeably throughout the remainder of the text.

---

\(^{19}\)It should be noted that only one set of class functions is created. These functions are shared between objects. The mechanism for using the same function on different objects’ data members is presented in Section 17.2.
EXERCISES 16.4

1. Define the following terms:
   a. class
   b. object
   c. declaration section
   d. implementation section
   e. instance variable
   f. member function
   g. data member
   h. constructor
   i. class instance
   j. services
   k. methods
   l. interface

2. Write a class declaration section for each of the following specifications. In each case include a prototype for a constructor and a member function named showdata() that can be used to display member values.
   a. A class named Time that has integer data members named secs, mins, and hours.
   b. A class named Complex that has floating-point data members named real and imaginary.
   c. A class named Circle that has integer data members named xcenter and ycenter and a floating-point data member named radius.
   d. A class named System that has character data members named computer, printer, and screen, each capable of holding 30 characters (including the end-of-string NULL), and floating-point data members named compPrice, printPrice, and scrnPrice.

3. a. Construct a class implementation section for the constructor and showdata() function members corresponding to the class declaration created for Exercise 2a.
   b. Construct a class implementation section for the constructor and showdata() function members corresponding to the class declaration created for Exercise 2b.
   c. Construct a class implementation section for the constructor and showdata() function members corresponding to the class declaration created for Exercise 2c.
   d. Construct a class implementation section for the constructor and showdata() function members corresponding to the class declaration created for Exercise 2d.

4. a. Include the class declaration and implementation sections prepared for Exercises 2a and 3a in a complete working program.
   b. Include the class declaration and implementation sections prepared for Exercises 2b and 3b in a complete working program.
   c. Include the class declaration and implementation sections prepared for Exercises 2c and 3c in a complete working program.
   d. Include the class declaration and implementation sections prepared for Exercises 2d and 3d in a complete working program.
5. Determine the errors in the following class declaration section:

```cpp
class Employee
{
    public:
        int empnum;
    char name[31];
    private:
        class(int = 0);
    void showemp(int, char *);
};
```

6. a. Add another member function named `convert()` to Program 16.11. The function should access the month, year, and day data members and display and then return a long integer that is calculated as `year * 10000 + month * 100 + day`. For example, if the date is 4/1/2006, the returned value is 20060401 (dates in this form are useful when performing sorts, as placing the numbers in numerical order automatically places the corresponding dates in chronological order).

   b. Include the modified `Date` class constructed for Exercise 6a in a complete C++ program.

7. a. Add an additional member function to Program 16.11’s class definition named `leapyr()` that returns a 1 when the year is a leap year and a 0 if it is not. A leap year is any year that is evenly divisible by 4 but not evenly divisible by 100, with the exception that years evenly divisible by 400 are leap years. For example, the year 1996 is a leap year because it is evenly divisible by 4 and not evenly divisible by 100. The year 2000 is a leap year because it is evenly divisible by 400.

   b. Include the class definition constructed for Exercise 7a in a complete C++ program. The `main()` function should display the message `The year is a leap year` or `The year is not a leap year` depending on the `Date` object’s year value.

8. a. Add a member function to Program 16.11’s class definition named `dayOfWeek()` that determines the day of the week for any `Date` object. An algorithm for determining the day of the week, known as Zeller’s algorithm, is the following:

   This algorithm assumes a date of the form `mm/dd/ccyy`, where `mm` is the month, `dd` is the day, `cc` is the century, and `yy` is the year in the century (for example, in the date `12/5/2000`, `mm = 12`, `dd = 5`, `cc = 20`, and `yy = 0`).

   ```cpp
   cc = int(ccyy/100)
   yy = ccyy % 100
   If the mm is less than 3,
       Set mm = mm + 12 and ccyy = ccyy - 1
   EndIf
   Set the variable T = dd + int(26 * (mm + 1)/10) + yy + int(yy/4) + int(cc/4) - (2 * cc)
   dayOfWeek = T % 7
   If dayOfWeek is less than 0
       dayOfWeek = dayOfWeek + 7
   EndIf
   Using this algorithm, the variable dayOfWeek has a value of 0 if the date is a Saturday, 1 if a Sunday, etc.
   
   b. Include the class definition constructed for Exercise 8a in a complete C++ program. The `main()` function should display the name of the day (`Sun`, `Mon`, `Tue`, etc.) for the `Date` object being tested.
16.5 Constructors and Destructors

A **constructor function** is any function that has the same name as its class. More than one constructor for each class can be defined. One constructor function is automatically called each time an object is created with the intended purpose of initializing the new object’s data members. Constructor functions may also perform other tasks when they are called and can be written in a variety of ways. In this section we present the possible variations of constructor functions and introduce another function, the destructor, which is automatically called whenever an object goes out of existence.

Figure 16.6 illustrates the general format of a constructor. As shown in this figure, a constructor:

1. Must have the same name as the class to which it belongs
2. Must have no return type (not even `void`)

```cpp
ClassName::ClassName (list)
{
    function body
}
```

**Figure 16.6** Constructor format

If you do not include a constructor in your class definition, the compiler will supply one for you. The supplied constructor, however, is a do-nothing constructor. For example, consider the following class declaration:

```cpp
class Date
{
    private:
        int month, day, year;
    public:
        void setdate(int, int, int);
        void showdate();
};
```

Since no user-defined constructor has been declared here, the compiler creates a default constructor. For our `Date` class this default constructor is equivalent to the implementation `Date(void){}`, that is, the compiler-supplied default constructor expects no parameters and has an empty body. Clearly this default constructor is not very useful, but it does exist if no other constructor is declared.

The term **default constructor** is used quite frequently in C++. It refers to any constructor that does not require any arguments when it is called. This can be because no arguments are declared, which is the case for the compiler-supplied default, or because all arguments have been given default values. For example, the prototype `Date(int = 7, int = 4, int = 2005);` is also valid for a default constructor. Here, each argument has been given a default value, and, when the corresponding constructor is written, an object can be declared as type `Date` without supplying any further arguments. Using such a constructor, the declaration `Date a;` initializes the `a` object with the default values 7, 4, and 2005.
To verify that a constructor function is automatically called whenever a new object is created, consider Program 16.14. Notice that in the implementation section the constructor function uses `cout` to display the message Created a new object with data values. Thus, whenever the constructor is called this message is displayed. Since the `main()` function creates three objects, the constructor is called three times and the message is displayed three times.

Program 16.14

```cpp
#include <iostream>
using namespace std;

// class declaration section
class Date
{

private:
    int month;
    int day;
    int year;

public:
    Date(int = 4, int = 7, int = 2005); // constructor
};

// implementation section
```

Programming Note

**Constructors**

A *constructor* is any function that has the same name as its class. The primary purpose of a constructor is to initialize an object's member variables when an object is created. Hence, a constructor is automatically called when an object is declared.

A class can have multiple constructors provided that each constructor is distinguishable by having a different parameter list. A compiler error results when unique identification of a constructor is not possible. If no constructor is provided the compiler supplies a do-nothing default constructor.

Every constructor function must be declared *with no return type* (not even void). Because they are functions, constructors may also be explicitly called in nondeclarative statements. When used in this manner, the function call requires parentheses following the constructor name, even if no parameters are used. However, when used in a declaration, parentheses *must not* be included for a zero parameter constructor. For example, the declaration `Date a();` is incorrect. The correct declaration is `Date a;`. When parameters are used, however, they must be enclosed within parentheses in both declarative and nondeclarative statements. Default parameter values should be included within the constructor's prototype.

To verify that a constructor function is automatically called whenever a new object is created, consider Program 16.14. Notice that in the implementation section the constructor function uses `cout` to display the message Created a new object with data values. Thus, whenever the constructor is called this message is displayed. Since the `main()` function creates three objects, the constructor is called three times and the message is displayed three times.
The following output is produced when Program 16.14 is executed.

Created a new data object with data values 7, 4, 2005
Created a new data object with data values 7, 4, 2005
Created a new data object with data values 4, 1, 2006

Although any legitimate C++ statement can be used within a constructor function such as the cout statement used in Program 16.14, it is best to keep constructors simple and use them only for initializing purposes. One further point needs to be made with respect to the constructor function in Program 16.14. According to the rules of C++, object data members are initialized in the order they are declared in the class declaration section, not in the order they may appear in the function’s definition within the implementation section. Usually, this is not an issue, unless one member is initialized using another data member’s value.

**Calling Constructors**

As we have seen, constructors are called whenever an object is created. The actual declaration, however, can be made in a variety of ways.

For example, the declaration

```
Date c(4,1,2006);
```

used in Program 16.14 could also have been written as

```
Date c = Date(4,1,2006);
```

This second form declares c as being of type Date and then makes a direct call to the constructor function with the arguments 4, 1, and 2006. This second form can be simplified
when only one argument is passed to the constructor. For example, if only the month data member of the c object needed to be initialized with the value 8 and the day and year members can use the default values, the object can be created using the declaration

```c
Date c = 8;
```

Because it resembles declarations in C, this form and its more complete equation form shown previously are said to be the C style of initialization. The nonequation form of declaration in Program 13.14 is called the C++ style of initialization and is the form we will use predominantly throughout the remainder of the text.

Regardless of which initialization form you use, in no case should an object be declared with empty parentheses. For example, the declaration

```c
Date a();
```

is not the same as the declaration

```c
Date a;
```

The latter declaration uses the default constructor values while the former declaration results in no object being created.

### Overloaded and Inline Constructors

The primary difference between a constructor and other user-written functions is how the constructor is called: constructors are called automatically each time an object is created, whereas other functions must be explicitly called by name.\(^\text{20}\) As a function, however, a constructor must still follow all of the rules applicable to user-written functions. This means that constructors may have default argument values, as illustrated in Program 16.14, may be overloaded, and may be written as inline functions.

Recall from Section 16.3 that function overloading permits the same function name to be used with different parameter lists. Based on the supplied argument types, the compiler determines which function to use when the call is encountered. Let’s see how this can

---

\(^{20}\)This is true for all other functions except destructors, which are described later in this section. A destructor function is automatically called each time an object is destroyed.
be applied to our Date class. For convenience, the appropriate class declaration is repeated below:

```cpp
// class declaration section
class Date {
  private:
    int month;
    int day;
    int year;
  public:
    Date(int = 7, int = 4, int = 2005); // constructor
};
```

Here, the constructor prototype specifies three integer values that are used to initialize the month, day, and year data members.

An alternate method of specifying a date is to use a long integer in the form `year * 10000 + month * 100 + day`. For example, using this form the date 12/24/1999 is 19991224 and the date 2/5/2006 is 20060205. A suitable prototype for a constructor that uses dates of this form is

```cpp
Date(long); // an overloaded constructor
```

Here, the constructor is declared as receiving one long integer value. The code for this new Date function must, of course, correctly convert its single argument value into a month, day, and year, and would be included within the class implementation section. The actual code for such a constructor is

```cpp
Date::Date(long yyyymmdd) // a second constructor
{
  year = (int) (yyyymmdd/10000.0); // extract the year
  month = (int) (yyyymmdd - year * 10000.0) / 100.00; // extract the month
  day = (int) (yyyymmdd - year * 10000.0 - month * 100.0); // extract the day
}
```

Do not be overly concerned with the conversion code within the function’s body. The important point here is the concept of overloading the Date() function to provide two constructors. Program 16.15 contains the complete class definition within the context of a working program.

---

**Program 16.15**

```cpp
1 #include <iostream>
2 #include <iomanip>
3 using namespace std;
4 // class declaration
5 class Date
```

---

21. The reasons for specifying dates in this manner are that only one number needs to be used per date and that sorting the numbers automatically puts the corresponding dates into chronological order.
private:
    int month;
    int day;
    int year;
public:
    Date(int = 7, int = 4, int = 2005); // constructor
    Date(long);
    void showdate(); // member function to display a date
};

// implementation section
Date::Date(int mm, int dd, int yyyy)
{
    month = mm;
    day = dd;
    year = yyyy;
}

Date::Date(long yyyymmdd) // here is the overloaded constructor
{
    year = (int)(yyyymmdd/10000.0); // extract the year
    month = (int)( (yyyymmdd - year*10000.0)/100.00 ); // extract the month
    day = (int)(yyyymmdd - year*10000.0 - month*100.0); // extract the day
}

void Date::showdate()
{
    cout << "The date is "
    << setfill('0')
    << setw(2) << month << '/'
    << setw(2) << day << '/'
    << setw(2) << year % 100
    << endl;
}

int main()
{
    Date a, b(4,1,1998), c(20060515L); // declare three objects
    a.showdate(); // display object a's values
    b.showdate(); // display object b's values
    c.showdate(); // display object c's values
    return 0;
}
The output provided by Program 16.15 is

The date is 07/04/05
The date is 04/01/98
The date is 05/15/06

Three objects are created in Program 16.15’s main() function. The first object, a, is initialized with the default constructor using its default argument values. Object b is also initialized with the default constructor but uses the argument values 4, 1, and 1998. Finally, object c, which is initialized with a long integer, uses the second constructor in the class implementation section. The compiler knows to use this second constructor because the argument specified, 20060515L, is clearly designated as a long integer. It is worthwhile to point out that a compiler error occurs if both Date constructors had default values. In such a case a declaration such as Date d; is ambiguous to the compiler, as it is not able to determine which constructor to use. Thus, in each implementation section, only one constructor can be written as the default.

Just as constructors may be overloaded, they also may be written as inline functions. Doing so simply means defining the function in the class declaration section. Making both the constructors in Program 16.15 inline is accomplished by the declaration section

```cpp
// class declaration
class Date
{
    private:
        int month;
        int day;
        int year;
    public:
        Date(int mm = 7, int dd = 4, int yyyy = 2005)
        {
            month = mm;
            day = dd;
            year = yyyy;
        }
        Date(long yyyymmdd) // here is the overloaded constructor
        {
            year = (int)(yyyymmdd/10000.0); // extract the year
            month = (int)((yyyymmdd - year * 10000.0)/100.00); // extract the month
            day = (int)(yyyymmdd - year * 10000.0 - month * 100.0); // extract the day
        }

    The keyword inline is not required in this declaration because member functions defined inside the class declaration are inline by default.

    Generally, only functions that can be coded on one or two lines are good candidates for inline functions. This reinforces the convention that inline functions should be small. Thus, the first constructor is more conventionally written as

    Date(int mm = 7, int dd = 4, int yyyy = 2005)
    {
        month = mm;
        day = dd;
        year = yyyy;
    }

    The second constructor, which extends over three lines, should not be written as an inline function.
```
Destructors

The counterpart to constructor functions is destructor functions. Destructors are functions having the same class name as constructors, but preceded with a tilde (~). Thus, for our Date class, the destructor name is ~Date(). Like constructors, a default do-nothing destructor is provided by the C++ compiler in the absence of an explicit destructor. Unlike constructors, however, there can be only one destructor function per class. This is because destructors take no arguments and return no values.

Destructors are automatically called whenever an object goes out of existence and are meant to “clean up” any undesirable effects that might be left by the object. Generally, such effects only occur when an object contains a pointer member.

Arrays of Objects

The importance of default constructors becomes evident when arrays of objects are created. Because a constructor is called each time an object is created, the default constructor provides an elegant way of initializing all objects to the same state.

Declaring an array of objects is the same as declaring an array of any built-in type. For example, the declaration

```cpp
Date thedate[5];
```

will create five objects named thedate[0] through thedate[4], respectively. Member functions for each of these objects are called by listing the object name followed by a dot (.) and the desired function. An example using an array of objects is provided by Program 16.16, which also includes cout statements within both the constructor and destructor. As illustrated by the output of this program, the constructor is called for each declared object, followed by five member function calls to showdate(), followed by five destructor calls. The destructor is called when the objects go out of scope. In this case, the destructor is called when the main() function terminates execution.22

Program 16.16

```cpp
1 #include <iostream>
2 #include <iostream>
3 using namespace std;
4 // class declaration
5 class Date
6 {
7    private:
8        int month;
9        int day;
10        int year;
11    public:
```

22A destructor for a local object is called when the smallest block containing the object’s definition goes out of scope. The destructor for a global object is called when the program terminates.


```cpp
12     Date(); // constructor
13     ~Date(); // destructor
14     void showdate();
15   );
16   // implementation section
17     Date::Date() // user-defined default constructor
18     {
19         cout << *** A Date object is being initialized ***\n*;
20         month = 1;
21         day = 1;
22         year = 2006
23     }
24 
25     Date::~Date() // user-defined destructor
26     {
27         cout << *** A Date object is going out of existence ***\n*;
28     }
29 
30     void Date::showdate()
31     {
32         cout << "The date is "
33             << setfill('0')
34             << setw(2) << month << '/'
35             << setw(2) << day << '/'
36             << setw(2) << year % 100
37             << endl;
38     
39         return;
40     }
41 
42     int main()
43     {
44         Date thedate[5];
45         for(int i = 0; i < 5; i++)
46             thedate[i],showdate();
47         return 0;
48     }
```

The output produced by Program 16.16 is

*** A Date object is being initialized ***
*** A Date object is being initialized ***
*** A Date object is being initialized ***
*** A Date object is being initialized ***
*** A Date object is being initialized ***
16-58 Introduction to C++

*** A Date object is being initialized ***
The date is 01/01/06
The date is 01/01/06
The date is 01/01/06
The date is 01/01/06
The date is 01/01/06

*** A Date object is going out of existence ***
*** A Date object is going out of existence ***
*** A Date object is going out of existence ***
*** A Date object is going out of existence ***
*** A Date object is going out of existence ***

EXERCISES 16.5

1. Determine whether the following statements are true or false:
   a. A constructor function must have the same name as its class.
   b. A class can have only one constructor function.
   c. A class can have only one default constructor function.
   d. A default constructor can only be supplied by the compiler.
   e. A default constructor can have no arguments or all arguments must have default values.
   f. A constructor must be declared for each class.
   g. A constructor must be declared with a return type.
   h. A constructor is automatically called each time an object is created.
   i. A class can have only one destructor function.
   j. A destructor can have default arguments.
   l. A destructor must be declared for each class.
   m. A destructor must be declared with a return type.
   n. A destructor is automatically called each time an object goes out of existence.
   o. Destructors are not useful when the class contains a pointer data member.

2. For Program 16.15, what date is initialized for object c if the declaration Date c(2006); is used in place of the declaration Date c(20060515L);?

3. Modify Program 16.15 so that the only data member of the class is a long integer named yyyymmdd. Do this by substituting the declaration
   long yyyymmdd;
   for the existing declarations
   int month;
   int day;
   int year;

   Then, rewrite the same constructor function prototypes currently declared in the class declaration section so that the Date(long) function becomes the default constructor and the Date(int, int, int) function converts a month, day, and year into the proper form for the class data member.
4. a. Construct a `Time` class containing integer data members `seconds`, `minutes`, and `hours`. Have the class contain two constructors: the first should be a default constructor having the prototype `time(int, int, int)`, which uses default values of 0 for each data member. The second constructor should accept a long integer representing a total number of seconds and disassemble the long integer into hours, minutes, and seconds. The final function member should display the class data members.

b. Include the class written for Exercise 4a within the context of a complete program.

5. a. Construct a class named `Student` consisting of an integer student identification number, an array of five floating-point grades, and an integer representing the total number of grades entered. The constructor for this class should initialize all `Student` data members to 0. Included in the class should be member functions to
   1. Enter a student ID number.
   2. Enter a single test grade and update the total number of grades entered.
   3. Compute an average grade and display the student ID followed by the average grade.

b. Include the class constructed in Exercise 5a within the context of a complete program. Your program should declare two objects of type `Student` and accept and display data for the two objects to verify operation of the member functions.

16.6 An Application

Now that you have an understanding of how classes are constructed and the terminology used in describing them, let us apply this knowledge to a particular application. In this application we simulate the operation of an elevator. We assume that the elevator can travel between the first and fifteenth floors of a building and that the location of the elevator must be known at all times.

For this application the location of the elevator corresponds to its current floor position and is represented by an integer variable ranging between 1 and 15. The value of this variable, which we name `currentFloor`, for current floor, effectively represents the current state of the elevator. The services that we provide for changing the state of the elevator are an initialization function to set the initial floor position when a new elevator is put in service and a request function to change the elevator's position (state) to a new floor. Putting an elevator in service is accomplished by declaring a single class instance (declaring an object of type `Elevator`). Requesting a new floor position is equivalent to pushing an elevator button. To accomplish this, a suitable class declaration is

```cpp
// class declaration section
class Elevator
{
  private:
    int currentFloor;
  public:
    Elevator(int);  // constructor
    void request(int);
};
```

Notice that we have declared one data member, `currentFloor`, and two class functions. The data member, `currentFloor`, is used to store the current floor position of the elevator. As a private member it can be accessed only through member functions. The two public
member functions, Elevator() and request() define the external services provided by each Elevator object. The Elevator() function, which has the same name as its class, becomes a constructor function that is automatically called when an object of type Elevator is created. We use this function to initialize the starting floor position of the elevator. The request() function is used to alter its position. To accomplish these services, a suitable class implementation section is

```cpp
// class implementation section

Elevator::Elevator(int cfloor) // constructor
{
    currentFloor = cfloor;
}

void Elevator::request(int newfloor) // access function
{
    if (newfloor < 1 || newfloor > MAXFLOOR || newfloor == currentFloor)
        ;  // do nothing
    else if (newfloor > currentFloor) // move elevator up
    {
        cout << "Starting at floor " << currentFloor << endl;
        while (newfloor > currentFloor)
        {
            currentFloor++; // add one to current floor
            cout << " Going Up - now at floor " << currentFloor << endl;
        }
        cout << "Stopping at floor " << currentFloor << endl;
    }
    else // move elevator down
    {
        cout << "Starting at floor " << currentFloor << endl;
        while (newfloor < currentFloor)
        {
            currentFloor--; // subtract one from current floor
            cout << " Going down - now at floor " << currentFloor << endl;
        }
        cout << "Stopping at floor " << currentFloor << endl;
    }
}
```

The constructor function is straightforward. When an Elevator object is declared, it is initialized to the floor specified; if no floor is explicitly given, a default value of 1, specified in the prototype, is used. For example, the declaration

```
Elevator a(7);
```

initializes the variable a.currentFloor to 7, while the declaration

```
Elevator a;
```

uses the default argument value and initializes the variable a.currentFloor to 1.

The request() function defined in the implementation section is more complicated and provides the class's primary service. Essentially this function consists of an if-else statement having three parts: if an incorrect service is requested no action is taken; if a floor
above the current position is selected the elevator is moved up; and if a floor below the
current position is selected the elevator is moved down. For movement up or down the
function uses a while loop to increment the position one floor at a time and reports the
elevator’s movement using a cout stream. Program 16.17 includes this class in a working
program.

Program 16.17

```cpp
#include <iostream>
using namespace std;

const int MAXFLOOR = 15;

// class declaration section
class Elevator
{
private:
  int currentFloor;
public:
  Elevator(int = 1); // constructor
  void request(int);
};

// implementation section
Elevator::Elevator(int cfloor) // constructor
{
  currentFloor = cfloor;
}

void Elevator::request(int newfloor) // access function
{
  if (newfloor < 1 || newfloor > MAXFLOOR || newfloor == currentFloor)
    ; // do nothing
  else if (newfloor > currentFloor) // move elevator up
    {
    cout << "\nStarting at floor " << currentFloor << endl;
    while (newfloor > currentFloor)
    {
    currentFloor++;
    cout << " Going up - now at floor " << currentFloor << endl;
    }
    cout << "Stopping at floor " << currentFloor << endl;
  }
  else // move elevator down
    {
```
The lightly shaded portion of Program 16.17 contains the class construction that we have already described. To see how this class is used, concentrate on the darker-shaded section of the program. At the top of the program we have included the iostream header file and declared a named constant, MAXFLOOR, that corresponds to the highest floor that can be requested.

Within the main() function three statements are included. The first statement creates an object named a of type Elevator. Because no explicit floor has been given, this elevator will begin at floor 1, which is the default constructor argument. A request is then made to move the elevator to floor 6, which is followed by a request to move the elevator to floor 3. The output produced by Program 16.7 is

Starting at floor 1
Going Up - now at floor 2
Going Up - now at floor 3
Going Up - now at floor 4
Going Up - now at floor 5
Going Up - now at floor 6
Stopping at floor 6
Starting at floor 6
Going Down - now at floor 5
Going Down - now at floor 4
Going Down - now at floor 3
Stopping at floor 3

The basic requirements of object-oriented programming are evident in even so simple a program as Program 16.17. Before the main() function can be written a useful class must be constructed. This is typical of programs that use objects. For such programs the design process is front-loaded with the requirement that careful consideration of the class—its declaration and implementation—be given. Code contained in the implementation section effectively removes code that would otherwise be part of main()’s responsibility. Thus, any
program that uses the object does not have to repeat the implementation details within its main() function. Rather, the main() function and any function called by main() is concerned only with sending messages to its objects to activate them appropriately. How the object responds to the messages and how the state of the object is retained is not main()'s concern; these details are hidden within the class construction.

EXERCISES 16.6

1. Enter Program 16.17 into your computer and execute it.

2. Modify the main() function in Program 16.17 to put a second elevator in service starting at the fifth floor. Have this second elevator move to the first floor and then to the twelfth floor.

3. Verify that the constructor function is called by adding a message within the constructor that is displayed each time a new object is created. Run your program to ensure its operation.

4. Modify the main() function in Program 16.17 to use a while loop that calls the elevator's request() function with a random number between 1 and 15. If the random number is the same as the elevator's current floor, generate another request. The while loop should terminate after 5 valid requests have been made and satisfied by movement of the elevator. (Hint: Review Section 6.4 for the use of random numbers and add a movement return code to the modified request() function.)

5. a. Construct a class definition that can be used to represent an employee of a company. Each employee is defined by an integer ID number, a name consisting of no more than 30 characters, a floating-point pay rate, and the maximum number of hours the employee should work each week. The services provided by the class should be the ability to enter data for a new employee, the ability to change data for a new employee, and the ability to display the existing data for a new employee.
   b. Include the class definition created for Exercise 4a in a working C++ program that asks the user to enter data for three employees and displays the entered data.

6. a. Construct a class definition that can be used to represent types of food. A type of food is classified as basic or prepared. Basic foods are further classified as either dairy, meat, fruit, vegetable, or grain. The services provided by the class should be the ability to enter data for a new food, the ability to change data for a new food, and the ability to display the existing data for a new food.
   b. Include the class definition created for Exercise 5a in a working C++ program that asks the user to enter data for four food items and displays the entered data.
16-64 Introduction to C++

16.7 Common Programming Errors

The more common programming errors initially associated with the construction of classes are

1. Failing to terminate the class declaration section with a semicolon.
2. Including a return type with the constructor's prototype or failing to include a return type with the other functions' prototypes.
3. Using the same name for a data member as for a member function.
4. Defining more than one default constructor for a class.
5. Forgetting to include the class name and scope operator, ::, in the header line of all member functions defined in the class implementation section.

All of these errors result in a compiler error message.

16.8 Chapter Summary

1. The lack of an easy-to-use high-resolution graphics capability is a serious difficulty when creating a graphical user interface that utilizes the full potential of a graphical monitor. This was one of the reasons for the development of object-oriented languages such as C++. In addition to providing the procedural aspects of C, the object-oriented C++ language provides the capability for creating user-defined data-types.

2. A central concept in object-oriented languages is that of an abstract data type (ADT), which is a programmer-defined data type that includes both a data type and operations that can be applied to the data.

3. The data types provided in all programming languages, such as integers, are examples of built-in data types. They contain the two required elements of an abstract data type, data and operations applicable to the data, but they are provided as an intrinsic part of the language.

4. Once an abstract data type has been created, objects of that abstract data type may be defined. Objects have the same relationship to an abstract data type as variables do to a language’s built-in data types.

5. In C++ an abstract data type is referred to as a class, which is simply a programmer-defined abstract data type. Objects of a class may be declared and have the same relationship to their class as variables do to C++’s built-in data types.

6. A class definition consists of declaration and implementation sections. The most common form of a class definition is

```
// class declaration section
class Name
{
    private:
        a list of variable declarations;
    public:
        a list of function prototypes;
```
class function definitions

The variables and functions declared in the class declaration section are collectively referred to as class members. The variables are individually referred to as class data members and the functions as class member functions. The terms private and public are access specifiers. Once an access specifier is listed it remains in force until another access specifier is given. The private keyword specifies that the class members following it are private to the class and can be accessed only by member functions. The public keyword specifies that the class members following may be accessed from outside the class. Generally all data members should be specified as private and all member functions as public.

7. Class functions listed in the declaration section may either be written inline or their definitions included in the class implementation section. Except for constructor and destructor functions, all class functions defined in the class implementation section have the header line:
   returnType ClassName::functionName(parameter list);

   Except for the addition of the class name and scope operator, ::, which are required to associate the function name with the class, this header line is identical to the header line used for any user-written function.

8. A constructor function is a special function that is automatically called each time an object is declared. It must have the same name as its class and cannot have any return type. Its purpose is to initialize each declared object.

9. If no constructor is declared for a class, the compiler supplies a default constructor. This is a do-nothing function having the form ClassName(void){}.

10. The term default constructor refers to any constructor that does not require any arguments when it is called. This can be because no parameters are declared (as is the case for the compiler-supplied default constructor) or because all arguments have been given default values.

11. Each class may only have one default constructor. If a user-defined default constructor is defined the compiler will not create its default constructor.

12. are created using either a C++ or C style of declaration. The C++ style of declaration has the form:
   ClassName list-of-object-names(list of initializers);

   where the list of initializers is optional. An example of this style of declaration, including initializers, for a class named Date is
   Date a,b,c(12,25,2006);

   Here, the objects a and b are declared to be of type Date and are initialized using the default constructor values, while the object c is initialized with the values 12, 25, and 2006.

   The equivalent C style of declaration, including the optional list of initializers, has the form:
   ClassName objectName = ClassName(list of initializers);

   An example of this style of declaration for a class named Date is
Date c = Date(12, 25, 2006)

Here the object c is created and initialized with the values 12, 25, and 2006.

13. Constructors maybe overloaded in the same manner as any other user-written C++ function.

14. If a constructor is defined for a class, a user-defined default constructor also should be written, as the compiler does not supply it.

15. A destructor function is called each time an object goes out of scope. Destructors must have the same name as their class, but preceded with a tilde (~). There can only be one destructor per class.

16. A destructor function takes no arguments and returns no value. If a user-defined destructor is not included in a class, the compiler provides a do-nothing destructor.

17. Arrays of objects are declared in the same manner as arrays of C++’s built-in data types. For example, if Date is a class name, the declaration

Date thedate[5];

creates five Date objects named thedate[0] through thedate[4]. Member functions for each of these objects are called by listing the object name, such as thedate[3], followed by a dot (.) and the desired member function name.