Defining Classes 10

"The time has come," the Walrus said,
"To talk of many things:
Of shoes—and ships—and sealing wax—
Of cabbages—and kings—"

LEWIS CARROLL, *Through the Looking-Glass*

INTRODUCTION

In Chapter 6 you learned how to use classes and objects, but not how to define classes. In this chapter we will show you how to define your own classes. A class is a data type. You can use the classes you define in the same way you use the predefined data types, such as `int`, `char`, and `ifstream`. However, unless you define your classes the right way, they will not be as well behaved as the predefined data types. Thus, we spend a good deal of time explaining what makes for a good class definition and give you some techniques to help you define your classes in a way that is consistent with modern programming practices.

Before we introduce classes, we will first present *structures* (also known as *structs*). When used in the way we present them here, a structure is a kind of simplified class and structures will prove to be a stepping-stone to understanding classes.

PREREQUISITES

This chapter uses material from Chapters 2 through 6.

10.1 STRUCTURES

As we said in Chapter 6, an object is a variable that has member functions, and a class is a data type whose variables are objects. Thus, the definition of a class should be a data type definition that describes two things: (1) what kinds of values the variables can hold and (2) what the member functions are. We will approach class definitions in two steps. We will first tell you how to give a type definition for a *structure*. A structure (of the kind discussed here) can be thought of as an object without any member functions. After you learn about structures, it will be a natural extension to define classes.

Structures for Diverse Data

Sometimes it is useful to have a collection of values of different types and to treat the collection as a single item. For example, consider a bank certificate of deposit, which is often called a CD. A CD is a bank account that does not allow withdrawals for a specified number of months. A CD naturally has three pieces of data associated with it: the account balance, the interest rate for the account, and the term, which is the number of months until maturity. The first two items can be represented as values of type *double*, and the number of months can be represented as a value of type *int*. Display 10.1 shows the definition of a structure called CDAccount that can be used for this kind of account. The definition is embedded in a complete program that demonstrates this structure type definition. As you can see from the sample dialogue, this particular bank specializes in short-term CDs, so the term will always be 12 or fewer months. Let's look at how this sample structure is defined and used.

The structure definition is as follows:

```
struct CDAccount
{
    double balance;
    double interest_rate;
    int term; //months until maturity
};
```

The keyword *struct* announces that this is a structure type definition. The identifier CDAccount is the name of the structure type. The name of a structure type is called the **structure tag.** The tag can be any legal identifier (but not a keyword). Although this convention is not required by the C++ language, structure tags are usually spelled with a mix of uppercase and lowercase letters, beginning with an uppercase letter. The identifiers declared inside the braces, {}, are called **member names.** As illustrated in this example, a structure type definition ends with both a brace, }, and a semicolon.

Where to place a structure definition

A structure definition is usually placed outside of any function definition (in the same way that globally defined constant declarations are placed outside of all function definitions). The structure type is then available to all the code that follows the structure definition.

Once a structure type definition has been given, the structure type can be used just like the predefined types *int*, *char*, and so forth. For example, the following will declare two variables, named my_account and your_account, both of type CDAccount:

```
CDAccount my_account, your_account;
```

A structure variable can hold values just like any other variable can hold values. A **structure value** is a collection of smaller values called **member values.** There is one member value for each member name declared in the structure definition. For example, a value of the type CDAccount is a collection of three member values: two of type *double* and one of type *int*. The member values that together make up the structure value are stored in *member variables*, which we discuss next.

Each structure type specifies a list of member names. In Display 10.1 the structure CDAccount has the three member names balance, interest_rate,

DISPLAY 10.1 A Structure Definition *(part 1 of 2)*

```
//Program to demonstrate the CDAccount structure type.
#include <iostream>
using namespace std;
//Structure for a bank certificate of deposit:
struct CDAccount
{
    double balance;
    double interest_rate;
    int term; //months until maturity
};


void get_data(CDAccount& the_account);
//Postcondition: the_account.balance and the_account.interest_rate
//have been given values that the user entered at the keyboard.


int main( )
{
    CDAccount account;
    get_data(account);

    double rate_fraction, interest;
    rate_fraction = account.interest_rate / 100.0;
    interest = account.balance * rate_fraction * (account.term / 12.0);
    account.balance = account.balance + interest;

    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    cout << "When your CD matures in "
         << account.term << " months,\n"
         << "it will have a balance of $"
         << account.balance << endl;
    return 0;
}

//Uses iostream:
void get_data(CDAccount& the_account)
{
    cout << "Enter account balance: $";
    cin >> the_account.balance;
    cout << "Enter account interest rate: ";
    cin >> the_account.interest_rate;
    cout << "Enter the number of months until maturity\n"
         << "(must be 12 or fewer months): ";
    cin >> the_account.term;
}
```

(continued)

DISPLAY 10.1 A Structure Definition *(part 2 of 2)*

Sample Dialogue

```
Enter account balance: $100.00
Enter account interest rate: 10.0
Enter the number of months until maturity
(must be 12 or fewer months): 6
When your CD matures in 6 months,
it will have a balance of $105.00
```

and `term`. Each of these member names can be used to pick out one smaller variable that is a part of the larger structure variable. These smaller variables are called **member variables.** Member variables are specified by giving the name of the structure variable followed by a dot (that is, followed by a period) and then the member name. For example, if `account` is a structure variable of type `CDAccount` (as declared in Display 10.1), then the structure variable `account` has the following three member variables:

```
account.balance
account.interest_rate
account.term
```

The first two member variables are of type *`double`*, and the last is of type *`int`*. These member variables can be used just like any other variables of those types. For example, the member variables above can be given values with the following three assignment statements:

```
account.balance = 1000.00;
account.interest_rate = 4.7;
account.term = 11;
```

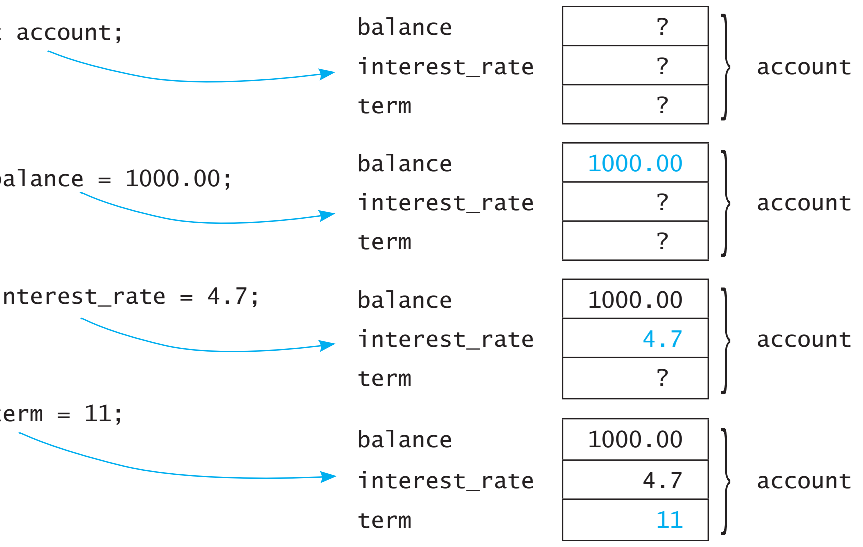
The result of these three statements is diagrammed in Display 10.2. Member variables can be used in all the ways that ordinary variables can be used. For example, the following line from the program in Display 10.1 will add the value contained in the member variable `account.balance` and the value contained in the ordinary variable `interest` and will then place the result in the member variable `account.balance`:

```
account.balance = account.balance + interest;
```

Notice that you specify a member variable for a structure variable by using the dot operator in the same way you used it in Chapter 6, where the dot operator was used to specify a member function of a class. The only difference is that in the case of structures, the members are variables rather than functions.

DISPLAY 10.2 Member Values

```
struct CDAccount
{
    double balance;
    double interest_rate;
    int term; //months until maturity
};
int main( )
{
   CDAccount account;
        . . .



   account.balance = 1000.00;



   account.interest_rate = 4.7;



   account.term = 11;
```



Reusing member names

Two or more structure types may use the same member names. For example, it is perfectly legal to have the following two type definitions in the same program:

```
struct FertilizerStock
{
    double quantity;
    double nitrogen_content;
};
```

and

```
struct CropYield
{
    int quantity;
    double size;
};
```

This coincidence of names will produce no problems. For example, if you declare the following two structure variables:

```
FertilizerStock super_grow;
CropYield apples;
```

then the quantity of `super_grow` fertilizer is stored in the member variable `super_grow.quantity` and the quantity of apples produced is stored in the member variable `apples.quantity`. The dot operator and the structure variable specify which `quantity` is meant in each instance.

Structure variables in assignment statements

A structure value can be viewed as a collection of member values. Viewed this way, a structure value is many different values. A structure value can also be viewed as a single (complex) value (which just happens to be made up of member values). Since a structure value can be viewed as a single value, structure values and structure variables can be used in the same ways that you use simple values and simple variables of the predefined types such as *`int`*. In particular, you can assign structure values using the equal sign. For example, if `apples` and `oranges` are structure variables of the type `CropYield` defined earlier, then the following is perfectly legal:

```
apples = oranges;
```

This assignment statement is equivalent to:

```
apples.quantity = oranges.quantity;
apples.size = oranges.size;
```

PITFALL Forgetting a Semicolon in a Structure Definition

When you add the final brace, `}`, to a structure definition, it feels like the structure definition is finished, but it is not. You must also place a semicolon after that final brace. There is a reason for this, even though the reason is a feature that we will have no occasion to use. A structure definition is more than a definition. It can also be used to declare structure variables. You are allowed to list structure variable names between that final brace and that final semicolon. For example, the following defines a structure called `WeatherData` and declares two structure variables, `data_point1` and `data_point2`, both of type `WeatherData`:

```
struct WeatherData
{
    double temperature;
    double wind_velocity;
} data_point1, data_point2;
```

However, as we said, we will always separate a structure definition and the declaration of variables of that structure type, so our structure definitions will always have a semicolon immediately after the final brace. ■

The Dot Operator

The **dot operator** is used to specify a member variable of a structure variable.

SYNTAX

Dot operator

```
Structure_Variable_Name.Member_Variable_Name
```

```
struct StudentRecord
{
    int student_number;
    char grade;
};

int main()
{
    StudentRecord your_record;
    your_record.student_number = 2001;
    your_record.grade = 'A';
```

Some writers call the dot operator the *structure member access operator* although we will not use that term.

Structures as Function Arguments

A function can have call-by-value parameters of a structure type and/or call-by-reference parameters of a structure type. The program in Display 10.1, for example, includes a function named `get_data` that has a call-by-reference parameter with the structure type `CDAccount`.

Functions can return structures

A structure type can also be the type for the value returned by a function. For example, the following defines a function that takes three appropriate arguments and returns a value of type `CDAccount`:

```
CDAccount shrink_wrap(double the_balance,
                      double the_rate, int the_term)
{
    CDAccount temp;
    temp.balance = the_balance;
    temp.interest_rate = the_rate;
    temp.term = the_term;
    return temp;
}
```

Notice the local variable `temp` of type `CDAccount`; `temp` is used to build up a complete structure value, which is then returned by the function. Once you have defined the function `shrink_wrap`, you can give a value to a variable of type `CDAccount` as illustrated by the following:

```
CDAccount new_account;
new_account = shrink_wrap(10000.00, 5.1, 11);
```

PROGRAMMING TIP Use Hierarchical Structures

Structures within structures

Sometimes it makes sense to have structures whose members are themselves smaller structures. For example, a structure type called `PersonInfo`, which can be used to store a person's height, weight, and birth date, can be defined as follows:

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

struct PersonInfo
{
    double height; //in inches
    int weight; //in pounds
    Date birthday;
};
```

A structure variable of type `PersonInfo` is declared in the usual way:

```
PersonInfo person1;
```

If the structure variable `person1` has had its value set to record a person's birth date, then the year the person was born can be output to the screen as follows:

```
cout << person1.birthday.year;
```

The way to read such expressions is left to right, and very carefully. Starting at the left end, `person1` is a structure variable of type `PersonInfo`. To obtain the member variable with the name `birthday`, use the dot operator as follows:

```
person1.birthday
```

This member variable is itself a structure variable of type `Date`. Thus, this member variable has member variables itself. A member variable of the structure variable `person1.birthday` is obtained by adding a dot and the member variable name, such as `year`, which produces the expression `person1.birthday.year` shown previously.

Simple Structure Types

You define a **structure type** as shown below. The *Structure_Tag* is the name of the structure type.

SYNTAX

```
struct Structure_Tag
{
    Type_1 Member_Variable_Name_1;
    Type_2 Member_Variable_Name_2;
        .
        .
        .
    Type_Last Member_Variable_Name_Last;
}; ←———— Do not forget this semicolon.
```

EXAMPLE

```
struct Automobile
{
    int year;
    int doors;
    double horse_power;
    char model;
};
```

Although we will not use this feature, you can combine member names of the same type into a single list separated by commas. For example, the following is equivalent to the previous structure definition:

```
struct Automobile
{
    int year, doors;
    double horse_power;
    char model;
};
```

Variables of a structure type can be declared in the same way as variables of other types. For example:

```
Automobile my_car, your_car;
```

The member variables are specified using the **dot operator.** For example,

```
my_car.year, my_car.doors, my_car.horse_power, and
my_car.model.
```

Initializing Structures

You can initialize a structure at the time that it is declared. To give a structure variable a value, you follow it by an equal sign and a list of the member values enclosed in braces. For example, the following definition of a structure type for a date was given in the previous subsection:

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

Once the type `Date` is defined, you can declare and initialize a structure variable called `due_date` as follows:

```
Date due_date = {12, 31, 2004};
```

Be sure to notice that the initializing values must be given in the order that corresponds to the order of member variables in the structure type definition. In this example, `due_date.month` receives the first initializing value of `12`, `due_date.day` receives the second value of `31`, and `due_date.year` receives the third value of `2004`.

It is an error if there are more initializers than *`struct`* members. If there are fewer initializer values than *`struct`* members, the provided values are used to initialize data members, in order. Each data member without an initializer is initialized to a zero value of an appropriate type for the variable.

SELF-TEST EXERCISES

1. Given the following structure and structure variable declaration:

```
struct TermAccount
{
    double balance;
    double interest_rate;
    int term;
    char initial1;
    char initial2;
};
TermAccount account;
```

what is the type of each of the following? Mark any that are not correct.

a. `account.balance`

b. `account.interest_rate`

   c. `TermAccount.term`

   d. `savings_account.initial1`

   e. `account.initial2`

   f. `account`

2. Consider the following type definition:

   ```
   struct ShoeType
   {
       char style;
       double price;
   };
   ```

   Given this structure type definition, what will be the output produced by the following code?

   ```
   ShoeType shoe1, shoe2;
   shoe1.style ='A';
   shoe1.price = 9.99;
   cout << shoe1.style << " $" << shoe1.price << endl;
   shoe2 = shoe1;

   shoe2.price = shoe2.price/9;
   cout << shoe2.style << " $" << shoe2.price << endl;
   ```

3. What is the error in the following structure definition? What is the message your compiler gives for this error? State what the error is, in your own words.

   ```
   struct Stuff
   {
       int b;
       int c;
   }
   int  main( )
   {
       Stuff x;
       //other code
   }
   ```

4. Given the following *struct* definition:

   ```
   struct A
   {
       int member_b;
       int member_c;
   };
   ```

declare x to have this structure type. Initialize the members of x, member_b and member_c, to the values 1 and 2, respectively.

(*Note:* This requests an initialization, not an assignment of the members. This distinction is important and will be made in a later chapter.)

5. Here is an initialization of a structure type. Tell what happens with each initialization. Note any problems with these initializations.

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

a. `Date due_date = {12, 21};`

b. `Date due_date = {12, 21, 20, 22};`

c. `Date due_date = {12, 21, 20, 22};`

d. `Date due_date = {12, 21, 22};`

6. Write a definition for a structure type for records consisting of a person's wage rate, accrued vacation (which is some whole number of days), and status (which is either hourly or salaried). Represent the status as one of the two *char* values 'H' and 'S'. Call the type EmployeeRecord.

7. Give a function definition corresponding to the following function declaration. (The type ShoeType is given in Self-Test Exercise 2.)

```
void read_shoe_record(ShoeType& new_shoe);
//Fills new_shoe with values read from the keyboard.
```

8. Give a function definition corresponding to the following function declaration. (The type ShoeType is given in Self-Test Exercise 2.)

```
ShoeType discount(ShoeType old_record);
//Returns a structure that is the same as its argument,
//but with the price reduced by 10%.
```

9. Give the structure definition for a type named StockRecord that has two member variables, one named shoe_info of the type ShoeType given in Self-Test Exercise 2 and one named arrival_date of type Date given in Self-Test Exercise 5.

10. Declare a variable of type StockRecord (given in the previous exercise) and write a statement that will set the year of the arrival date to 2006.

10.2 CLASSES

I don't care to belong to any club that will accept me as a member.

GROUCHO MARX, *The Groucho Letters*

Defining Classes and Member Functions

A **class** is a data type whose variables are objects. In Chapter 6 we described an **object** as a variable that has member functions as well as the ability to hold data values.[1] Thus, within a C++ program, the definition of a class should be a data type definition that describes what kinds of values the variables can hold and also what the member functions are. A structure definition describes some of these things. A structure is a defined type that allows you to define values of the structure type by defining member variables. To obtain a class from a structure, all you need to do is add some member functions.

A sample class definition is given in the program shown in Display 10.3. The type `DayOfYear` defined there is a class definition for objects whose values are dates, such as January 1 or July 4. These values can be used to record holidays, birthdays, and other special dates. In this definition of `DayOfYear`, the month is recorded as an *`int`* value, with `1` standing for January, `2` standing for February, and so forth. The day of the month is recorded in a second *`int`* member variable. The class `DayOfYear` has one member function called `output`, which has no arguments and outputs the month and day values to the screen. Let's look at the definition for the class `DayOfYear` in detail.

A member function

The definition of the class `DayOfYear` is shown near the top of Display 10.3. For the moment, ignore the line that contains the keyword *`public`*. This line simply says that the member variables and functions have no restriction on them. We will explain this line later in this chapter. The rest of the definition of the class `DayOfYear` is very much like a structure definition, except that it uses the keyword *`class`* instead of *`struct`* and it lists the member function `output` (as well as the member variables `month` and `day`). Notice that the member function `output` is listed by giving only its function declaration. The definitions for the member functions are given elsewhere. (In a C++ class definition, you can intermix the ordering of the member variables and member functions in any way you wish, but the style we will follow has a tendency to list the member functions before the member variables.) Objects (that is, variables) of a class type are declared in the same way as variables of the predefined types and in the same way as structure variables.

[1]The object is actually the value of the variable rather than the variable itself, but since we use the variable to name the value it holds, we can simplify our discussion by ignoring this nicety and talking as if the variable and its value were the same thing.

DISPLAY 10.3 Class with a Member Function *(part 1 of 2)*

```
//Program to demonstrate a very simple example of a class.
//A better version of the class DayOfYear will be given in Display 10.4.
#include <iostream>
using namespace std;

class DayOfYear
{
public:
    void output( );
    int month;
    int day;
};

int main( )
{
    DayOfYear today, birthday;

    cout << "Enter today's date:\n";
    cout << "Enter month as a number: ";
    cin >> today.month;
    cout << "Enter the day of the month: ";
    cin >> today.day;
    cout << "Enter your birthday:\n";
    cout << "Enter month as a number: ";
    cin >> birthday.month;
    cout << "Enter the day of the month: ";
    cin >> birthday.day;

    cout << "Today's date is ";
    today.output( );
    cout << "Your birthday is ";
    birthday.output( );

    if (today.month == birthday.month
        && today.day == birthday.day)
        cout << "Happy Birthday!\n";
    else
        cout << "Happy Unbirthday!\n";

    return 0;
}

//Uses iostream:
void DayOfYear::output( )
{
    cout << "month = " << month
         << ", day = " << day << endl;
}
```

Member function declaration

Calls to the member function `output`

Member function definition

(continued)

DISPLAY 10.3 Class with a Member Function *(part 2 of 2)*

Sample Dialogue

```
Enter today's date:
Enter month as a number: 10
Enter the day of the month: 15
Enter your birthday:
Enter month as a number: 2
Enter the day of the month: 21
Today's date is month = 10, day = 15
Your birthday is month = 2, day = 21
Happy Unbirthday!
```

Calling member functions

Member functions for classes that you define are called in the same way as we described in Chapter 6 for predefined classes. For example, the program in Display 10.3 declares two objects of type `DayOfYear` in the following way:

```
DayOfYear today, birthday;
```

The member function `output` is called with the object `today` as follows:

```
today.output();
```

and the member function `output` is called with the object `birthday` as follows:

```
birthday.output();
```

Encapsulation

Combining a number of items, such as variables and functions, into a single package, such as an object of some class, is called **encapsulation.**

Defining member functions

When a member function is defined, the definition must include the class name because there may be two or more classes that have member functions with the same name. In Display 10.3 there is only one class definition, but in other situations you may have many class definitions, and each class may have a member function called `output`. The definition for the member function `output` of the class `DayOfYear` is shown in Display 10.3. The definition is similar to an ordinary function definition, but there are some differences.

The heading of the function definition for the member function `output` is as follows:

```
void DayOfYear::output()
```

The operator `::` is called the **scope resolution operator,** and it serves a purpose similar to that of the dot operator. Both the dot operator and the scope resolution operator are used to tell what a member function is a member of. However, the scope resolution operator `::` is used with a class name, whereas the dot operator is used with objects (that is, with class variables). The scope resolution operator consists of two colons with no space between them. The class name that precedes the scope resolution operator is often called a **type qualifier,** because it specializes ("qualifies") the function name to one particular type.

Member variables in function definitions

Look at the definition of the member function `DayOfYear::output` given in Display 10.3. Notice that in the function definition of `DayOfYear::output`, we used the member names `month` and `day` by themselves without first giving the object and dot operator. That is not as strange as it may at first appear. At this point we are simply defining the member function `output`. This definition of `output` will apply to all objects of type `DayOfYear`, but at this point we do not know the names of the objects of type `DayOfYear` that we will use, so we cannot give their names. When the member function is called, as in

```
today.output();
```

all the member names in the function definition are specialized to the name of the calling object. So the function call above is equivalent to the following:

```
{
    cout << "month = " << today.month
         << ", day = " << today.day << endl;
}
```

In the function definition for a member function, you can use the names of all members of that class (both the data members and the function members) without using the dot operator.

Member Function Definition

A member function is defined the same way as any other function except that the *Class_Name* and the scope resolution operator `::` are given in the function heading.

SYNTAX

```
Returned_Type Class_Name::Function_Name(Parameter_List)
{
    Function_Body_Statements
}
```

(continued)

EXAMPLE

```
//Uses iostream:
void DayOfYear::output()
{
     cout << "month = " << month
          << ", day = " << day << endl;
}
```

The class definition for the example class DayOfYear above is given in Display 10.3, where month and day are defined as the names of member variables for the class DayOfYear. Note that month and day are not preceded by an object name and dot.

The Dot Operator and the Scope Resolution Operator

Both the dot operator and the scope resolution operator are used with member names to specify what thing they are members of. For example, suppose you have declared a class called DayOfYear and you declare an object called today as follows:

```
DayOfYear today;
```

You use the **dot operator** to specify a member of the object today. For example, output is a member function for the class DayOfYear (defined in Display 10.3), and the following function call will output the data values stored in the object today:

```
today.output();
```

You use the **scope resolution operator ::** to specify the class name when giving the function definition for a member function. For example, the heading of the function definition for the member function output would be as follows:

```
void DayOfYear::output()
```

Remember, the scope resolution operator :: is used with a class name, whereas the dot operator is used with an object of that class.

SELF-TEST EXERCISES

11. Below we have redefined the class `DayOfYear` from Display 10.3 so that it now has one additional member function called `input`. Write an appropriate definition for the member function `input`.

```
class DayOfYear
{
public:
    void input();
    void output();
    int month;
    int day;
};
```

12. Given the following class definition, write an appropriate definition for the member function `set`:

```
class Temperature
{
public:
    void set(double new_degrees, char new_scale);
    //Sets the member variables to the values given as
    //arguments.

    double degrees;
    char scale; //'F' for Fahrenheit or 'C' for Celsius.
};
```

13. Carefully distinguish between the meaning and use of the dot operator and the scope resolution operator `::`.

Public and Private Members

The predefined types such as *double* are not implemented as C++ classes, but the people who wrote your C++ compiler did design some way to represent values of type *double* in your computer. It is possible to implement the type *double* in many different ways. In fact, different versions of C++ do implement the type *double* in slightly different ways, but if you move your C++ program from one computer to another with a different implementation of the type *double*, your program should still work correctly.[2] Classes are types that you define, and the types that you define should behave as well as the predefined types. You can build a library of your own class type definitions and use your types as if they were predefined types. For example, you could place each class definition in a separate file and copy it into any program that uses the type.

[2] Sometimes this ideal is not quite realized, but in the ideal world it should be realized, and at least for simple programs, it is realized even in the imperfect world that we live in.

Your class definitions should separate the rules for using the class and the details of the class implementation in as strong a way as was done for the predefined types. If you change the implementation of a class (for example, by changing some details in the definition of a member function in order to make function calls run faster), then you should not need to change any of the other parts of your programs. In order to realize this ideal, we need to describe one more feature of class definitions.

Look back at the definition of the type `DayOfYear` given in Display 10.3. The type `DayOfYear` is designed to hold values that represent dates such as birthdays and holidays. We chose to represent these dates as two integers, one for the month and one for the day of the month. We might later decide to change the representation of the month from one variable of type *`int`* to three variables of type *`char`*. In this changed version, the three characters would be an abbreviation of the month's name. For example, the three *`char`* values `'J'`, `'a'`, and `'n'` would represent the month January. However, whether you use a single member variable of type *`int`* to record the month or three member variables of type *`char`* is an implementation detail that need not concern a programmer who uses the type `DayOfYear`. Of course, if you change the way the class `DayOfYear` represents the month, then you must change the implementation of the member function `output`—but that is all you should need to change. You should not need to change any other part of a program that uses your class definition for `DayOfYear`. Unfortunately, the program in Display 10.3 does not meet this ideal. For example, if you replace the one member variable named `month` with three member variables of type *`char`*, then there will be no member variable named `month`, so you must change those parts of the program that perform input and also change the *`if-else`* statement.

With an ideal class definition, you should be able to change the details of how the class is implemented and the only things you should need to change in any program that uses the class are the definitions of the member functions. In order to realize this ideal, you must have enough member functions so that you never need to access the member variables directly, but access them only through the member functions. Then, if you change the member variables, you need change only the definitions of the member functions to match your changes to the member variables, and nothing else in your programs need change. In Display 10.4 we have redefined the class `DayOfYear` so that it has enough member functions to do everything we want our programs to do, and so the program does not need to directly reference any member variables. If you look carefully at the program in Display 10.4, you will see that the only place the member variable names `month` and `day` are used is in the definitions of the member functions. There is no reference to `today.month`, `today.day`, `bach_birthday.month`, nor `bach_birthday.day` anywhere outside of the definitions of member functions.

The program in Display 10.4 has one new feature that is designed to ensure that no programmer who uses the class `DayOfYear` will ever

DISPLAY 10.4 Class with Private Members *(part 1 of 2)*

```
//Program to demonstrate the class DayOfYear.
#include <iostream>
using namespace std;
class DayOfYear
{
public:
    void input( );
    void output( );
    void set(int new_month, int new_day);
    //Precondition: new_month and new_day form a possible date.
    //Postcondition: The date is reset according to the arguments.
    int get_month( );
    //Returns the month, 1 for January, 2 for February, etc.
    int get_day( );
    //Returns the day of the month.
private:
    void check_date( );
    int month;
    int day;
};
int main( )
{
    DayOfYear today, bach_birthday;
    cout << "Enter today's date:\n";
    today.input( );
    cout << "Today's date is ";
    today.output( );
    bach_birthday.set(3, 21);
    cout << "J. S. Bach's birthday is ";
    bach_birthday.output( );
    if (today.get_month( ) == bach_birthday.get_month( ) &&
        today.get_day( ) == bach_birthday.get_day( ) )
        cout << "Happy Birthday Johann Sebastian!\n";
    else
        cout << "Happy Unbirthday Johann Sebastian!\n";
    return 0;
}
//Uses iostream:
void DayOfYear::input( )
{
    cout << "Enter the month as a number: ";
```

This is an improved version of the class `DayOfYear` *that we gave in Display 10.3.*

Private member function (`check_date`)

Private member variables (`month`, `day`)

(continued)

DISPLAY 10.4 Class with Private Members *(part 2 of 2)*

```
        cin >> month;
        cout << "Enter the day of the month: ";
        cin >> day;
        check_date( );
    }

    void DayOfYear::output( )
     <The rest of the definition of DayOfYear::output is
      given in Display 10.3.>

    void DayOfYear::set(int new_month, int new_day)
    {
        month = new_month;
        day = new_day;
        check_date();
    }

    void DayOfYear::check_date( )
    {
        if ((month < 1) || (month > 12) || (day < 1) || (day > 31))
        {
            cout << "Illegal date. Aborting program.\n";
            exit(1);
        }
    }

    int DayOfYear::get_month( )
    {
        return month;
    }

    int DayOfYear::get_day( )
    {
        return day;
    }
```

Private members may be used in member function definitions (but not elsewhere).

A better definition of the member function `input` *would ask the user to reenter the date if the user enters an incorrect date.*

The member function `check_date` *does not check for all illegal dates, but it would be easy to make the check complete by making it longer. See Self-Test Exercise 14.*

The function `exit` *is discussed in Chapter 6. It ends the program.*

Sample Dialogue

```
Enter today's date:
Enter the month as a number: 3
Enter the day of the month: 21
Today's date is month = 3, day = 21
J. S. Bach's birthday is month = 3, day = 21
Happy Birthday Johann Sebastian!
```

directly reference any of its member variables. Notice the line in the definition of the class `DayOfYear` that contains the keyword *`private`*. All the member variable names that are listed after this line are **private members,** which means that they cannot be directly accessed in the program except within the definition of a member function. If you try to access one of these member variables in the `main` part of your program or in the definition of some function that is not a member function of this particular class, the compiler will give you an error message. If you insert the keyword *`private`* and a colon in the list of member variables and member functions, all the members that follow the label *`private`*`:` will be private members. The variables that follow the label *`private`*`:` will be **private member variables,** and the functions that follow it will be **private member functions.**

All the member variables for the class `DayOfYear` defined in Display 10.4 are private members. A private member variable may be used in the definition of any of the member functions, but nowhere else. For example, with this changed definition of the class `DayOfYear`, the following two assignments are no longer permitted in the `main` part of the program:

```
DayOfYear today; //This line is OK.
today.month = 12; //ILLEGAL
today.day = 25; //ILLEGAL
```

Any reference to these private variables is illegal (except in the definition of member functions). Since this new definition makes `month` and `day` private member variables, the following are also illegal in the `main` part of any program that declares `today` to be of type `DayOfYear`:

```
cout << today.month; //ILLEGAL
cout << today.day; //ILLEGAL
if (today.month == 1) //ILLEGAL
    cout << "January";
```

Once you make a member variable a private member variable, there is then no way to change its value (or to reference the member variable in any other way) except by using one of the member functions. This is a severe restriction, but it is usually a wise restriction to impose. Programmers find that it usually makes their code easier to understand and easier to update if they make all member variables private.

It may seem that the program in Display 10.4 does not really disallow direct access to the private member variables, since they can be changed using the member function `DayOfYear::set`, and their values can be discovered using the member functions `DayOfYear::get_month` and `DayOfYear::get_day`. While that is almost true for the program in Display 10.4, it might not be so true if we changed the implementation of how we represented the month and/or day in our dates. For example, suppose we change the type definition of `DayOfYear` to the following:

```
class DayOfYear
{
public:
    void input();
    void output();

    void set(int new_month, int new_day);
    //Precondition: new_month and new_day form a possible date.
    //Postcondition: The date is reset according to the
    //arguments.

    int get_month();
    //Returns the month, 1 for January, 2 for February, etc.

    int get_day();
    //Returns the day of the month.
private:
    void DayOfYear::check_date( );
    char first_letter; //of month
    char second_letter; //of month
    char third_letter; //of month
    int day;
};
```

It would then be slightly more difficult to define the member functions, but they could be redefined so that they would behave *exactly* as they did before. For example, the definition of the function `get_month` might start as follows:

```
int DayOfYear::get_month()
{
    if (first_letter == 'J' && second_letter == 'a'
            && third_letter == 'n')
        return 1;
    if (first_letter == 'F' && second_letter == 'e'
            && third_letter == 'b')
        return 2;
        . . .
```

This approach would be rather tedious, but not difficult.

Also notice that the member functions `DayOfYear::set` and `DayOfYear::input` check to make sure the member variables `month` and `day` are set to legal values. This is done with a call to the member function `DayOfYear::check_date`. If the member variables `month` and `day` were public instead of private, then these member variables could be set to any values, including illegal values. By making the member variables private and manipulating them only via member functions, we can ensure that the member variables are never set to illegal or meaningless values. (In Self-Test Exercise 14 you are asked to redefine the member function `DayOfYear::check_date` so that it does a complete check for illegal dates.)

It is also possible to make a member function private. Like a private member variable, a private member function can be used in the definition of any other member function, but nowhere else, such as in the `main` part of a program that uses the class type. For example, the member function `DayOfYear::check_date` in Display 10.4 is a private member function. The normal practice is to make a member function private if you only expect to use that member function as a helping function in the definitions of the member functions.

The keyword *`public`* is used to indicate **public members** the same way that the keyword *`private`* is used to indicate private members. For example, for the class `DayOfYear` defined in Display 10.4, all the member functions except `DayOfYear::check_date` are public members (and all the member variables are private members). A public member can be used in the `main` body of your program or in the definition of any function, even a nonmember function.

VideoNote
Class Scope, Public and Private Members

You can have any number of occurrences of *`public`* and *`private`* in a class definition. Every time you insert the label

```
public:
```

the list of members changes from private to public. Every time you insert the label

```
private:
```

the list of members changes back to being private members. For example, the member function `do_something_else` and the member variable `more_stuff` in the following structure definition are private members, while the other four members are all public:

```
class SampleClass
{
public:
    void do_something();
    int stuff;
private:
    void do_something_else();
    char more_stuff;
public:
    double do_yet_another_thing();
    double even_more_stuff;
};
```

If you list members at the start of your class definition and do not insert either *`public:`* or *`private:`* before these first members, then they will be private members. However, it is a good idea to always explicitly label each group of members as either *`public`* or *`private`*.

Classes and Objects

A **class** is a type whose variables are **objects.** These objects can have both member variables and member functions. The syntax for a class definition is as follows.

SYNTAX

```
class Class_Name
{
public:
    Member_Specification_1
    Member_Specification_2
            .
            .
            .
    Member_Specification_n
private:
    Member_Specification_n+1
    Member_Specification_n+2
        .
        .
        .
};
```

Each *Member_Specification_i* is either a member variable declaration or a member function declaration. (Additional *public* and *private* sections are permitted.)

EXAMPLE

```
class Bicycle
{
public:
    char get_color();
    int number_of_speeds();
    void set(int the_speeds, char the_color);
private:
    int speeds;
    char color;
};
```

Once a class is defined, an object (which is just a variable of the class type) can be declared in the same way as variables of any other type. For example, the following declares two objects of type Bicycle:

```
Bicycle my_bike, your_bike;
```

PROGRAMMING TIP Make All Member Variables Private

When defining a class, the normal practice is to make all member variables private. This means that the member variables can only be accessed or changed using the member functions. Much of this chapter is dedicated to explaining how and why you should define classes in this way.

PROGRAMMING TIP Define Accessor and Mutator Functions

The operator == can be used to test two values of a simple type to see if they are equal. Unfortunately, the predefined operator == does not automatically apply to objects. In Chapter 11 we will show you how you can make the operator == apply to the objects of the classes you define. Until then, you will not be able to use the equality operator == with objects (nor can you use it with structures). This can produce some complications. When defining a class, the preferred style is to make all member variables private. Thus, in order to test two objects to see if they represent the same value, you need some way to access the values of the member variables (or something equivalent to the values of the member variables). This allows you to test for equality by testing the values of each pair of corresponding member variables. To do this in Display 10.4, we used the member functions `get_month` and `get_day` in the *`if-else`* statement.

Member functions, such as `get_month` and `get_day`, that allow you to find out the values of the private member variables are called **accessor functions.** Given the techniques you have learned to date, it is important to always include a complete set of accessor functions with each class definition so that you can test objects for equality. The accessor functions need not literally return the values of each member variable, but they must return something equivalent to those values. In Chapter 11 we will develop a more elegant method to test two objects for equality, but even after you learn that technique, it will still be handy to have accessor functions.

Member functions, such as `set` in Display 10.4, that allow you to change the values of the private member variables are called **mutator functions.** It is important to always include mutator functions with each class definition so that you can change the data stored in an object.

Accessor and Mutator Functions

Member functions that allow you to find out the values of the private member variables of a class are called **accessor functions.** The accessor functions need not literally return the values of each member variable, but they must return something equivalent to those values. Although this is not required by the C++ language, the names of accessor functions normally include the word `get`.

(continued)

Member functions that allow you to change the values of the private member variables of a class are called **mutator functions.** Although this is not required by the C++ language, the names of mutator functions normally include the word set.

It is important to always include accessor and mutator functions with each class definition so that you can change the data stored in an object.

SELF-TEST EXERCISES

14. The private member function DayOfYear::check_date in Display 10.4 allows some illegal dates to get through, such as February 30. Redefine the member function DayOfYear::check_date so that it ends the program whenever it finds any illegal date. Allow February to contain 29 days, so you account for leap years. (*Hint:* This is a bit tedious and the function definition is a bit long, but it is not very difficult.)

15. Suppose your program contains the following class definition:

```
class Automobile
{
public:
    void set_price(double new_price);
    void set_profit(double new_profit);
    double get_price();
private:
    double price;
    double profit;
    double get_profit();
};
```

and suppose the main part of your program contains the following declaration and that the program somehow sets the values of all the member variables to some values:

```
Automobile hyundai, jaguar;
```

Which of the following statements are then allowed in the main part of your program?

```
hyundai.price = 4999.99;
jaguar.set_price(30000.97);
double a_price, a_profit;
a_price = jaguar.get_price();
a_profit = jaguar.get_profit();
a_profit = hyundai.get_profit();
```

```
if (hyundai == jaguar)
    cout << "Want to swap cars?";
hyundai = jaguar;
```

16. Suppose you change Self-Test Exercise 15 so that the definition of the class `Automobile` omits the line that contains the keyword `private`. How would this change your answer to the question in Self-Test Exercise 15?

17. Explain what `public:` and `private:` do in a class definition. In particular, explain why we do not just make everything `public:` and save difficulty in access.

18. a. How many `public:` sections are required in a class for the class to be useful?

    b. How many `private:` sections are required in a class?

    c. What kind of section do you have between the opening { and the first `public:` or `private:` section label of a class?

    d. What kind of section do you have between the opening { and the first `public:` or `private:` section label of a structure?

PROGRAMMING TIP Use the Assignment Operator with Objects

It is perfectly legal to use the assignment operator = with objects or with structures. For example, suppose the class `DayOfYear` is defined as shown in Display 10.4 so that it has two private member variables named `month` and `day`, and suppose that the objects `due_date` and `tomorrow` are declared as follows:

```
DayOfYear due_date, tomorrow;
```

The following is then perfectly legal (provided the member variables of the object `tomorrow` have already been given values):

```
due_date = tomorrow;
```

The previous assignment is equivalent to the following:

```
due_date.month = tomorrow.month;
due_date.day = tomorrow.day;
```

Moreover, this is true even though the member variables named `month` and `day` are private members of the class `DayOfYear`.[3] ■

[3] In Chapter 11 we see situations in which the assignment operator = should be redefined (overloaded) for a class.

PROGRAMMING EXAMPLE BankAccount Class—Version 1

Display 10.5 contains a class definition for a bank account that illustrates all of the points about class definitions you have seen thus far. This type of bank account allows you to withdraw your money at any time, so it has no term as did the type `CDAccount` that you saw earlier. A more important difference is that the class `BankAccount` has member functions for all the operations you would expect to use in a program. Objects of the class `BankAccount` have two private member variables: one to record the account balance and one to record the interest rate. Let's discuss some of features of the class `BankAccount`.

First, notice that the class `BankAccount` has a private member function called `fraction`. Since `fraction` is a private member function, it cannot be called in the body of `main` or in the body of any function that is not a member function of the class `BankAccount`. The function `fraction` can only be called in the definitions of other member functions of the class `BankAccount`. The only reason we have this (or any) private member function is to aid us in defining other member functions for the same class. In our definition of the class `BankAccount`, we included the member function `fraction` so that we could use it in the definition of the function `update`. The function `fraction` takes one argument that is a percentage figure, like `10.0` for 10.0%, and converts it to a fraction, like `0.10`. That allows us to compute the amount of interest on the account at the given percentage. If the account contains $100.00 and the interest rate is 10%, then the interest is equal to $100 times 0.10, which is $10.00.

When you call a public member function, such as `update`, in the `main` body of your program, you must include an object name and a dot, as in the following line from Display 10.5:

```
account1.update();
```

One member function calling another

However, when you call a private member function (or any other member function) within the definition of another member function, you use only the member function name without any calling object or dot operator. For example, the following definition of the member function `BankAccount::update` includes a call to `BankAccount::fraction` (as shown in Display 10.5):

```
void BankAccount::update()
{
    balance = balance + fraction(interest_rate) * balance;
}
```

The calling object for the member function `fraction` and for the member variables `balance` and `interest_rate` are determined when the function `update` is called. For example, the meaning of

```
account1.update();
```

is the following:

DISPLAY 10.5 The BankAccount Class *(part 1 of 3)*

```
//Program to demonstrate the class BankAccount.
#include <iostream>
using namespace std;

//Class for a bank account:
class BankAccount
{
public:
    void set(int dollars, int cents, double rate);
    //Postcondition: The account balance has been set to $dollars.cents;
    //The interest rate has been set to rate percent.

    void set(int dollars, double rate);
    //Postcondition: The account balance has been set to $dollars.00.
    //The interest rate has been set to rate percent.

    void update( );
    //Postcondition: One year of simple interest has been
    //added to the account balance.

    double get_balance( );
    //Returns the current account balance.

    double get_rate( );
    //Returns the current account interest rate as a percentage.

    void output(ostream& outs);
    //Precondition: If outs is a file output stream, then
    //outs has already been connected to a file.
    //Postcondition: Account balance and interest rate have
    //been written to the stream outs.
private:
    double balance;
    double interest_rate;

    double fraction(double percent);
    //Converts a percentage to a fraction. For example, fraction(50.3)
    //returns 0.503.
};

int main( )
{
    BankAccount account1, account2;
    cout << "Start of Test:\n";
    account1.set(123, 99, 3.0);
    cout << "account1 initial statement:\n";
    account1.output(cout);
    account1.set(100, 5.0);
```

The member function set is overloaded.

Calls to the overloaded member function set

(continued)

DISPLAY 10.5 The BankAccount Class *(part 2 of 3)*

```
    cout << "account1 with new setup:\n";
    account1.output(cout);

    account1.update( );
    cout << "account1 after update:\n";
    account1.output(cout);

    account2 = account1;
    cout << "account2:\n";
    account2.output(cout);
    return 0;
}

void BankAccount::set(int dollars, int cents, double rate)
{
    if ((dollars < 0) || (cents < 0) || (rate < 0))
    {
        cout << "Illegal values for money or interest rate.\n";
        exit(1);
    }

    balance = dollars + 0.01*cents;
    interest_rate = rate;
}

void BankAccount::set(int dollars, double rate)
{
    if ((dollars < 0) || (rate < 0))
    {
        cout << "Illegal values for money or interest rate.\n";
        exit(1);
    }

    balance = dollars;
    interest_rate = rate;
}

void BankAccount::update( )
{
    balance = balance + fraction(interest_rate)*balance;
}

double BankAccount::fraction(double percent_value)
{
    return (percent_value / 100.0);
}

```

Definitions of overloaded member function `set`

In the definition of a member function, you call another member function like this.

(continued)

DISPLAY 10.5 The `BankAccount` Class *(part 3 of 3)*

```
double BankAccount::get_balance( )
{
    return balance;
}

double BankAccount::get_rate( )
{
    return interest_rate;
}

//Uses iostream:
void BankAccount::output(ostream& outs)
{
    outs.setf(ios::fixed);
    outs.setf(ios::showpoint);
    outs.precision(2);
    outs << "Account balance $" << balance << endl;
    outs << "Interest rate " << interest_rate << "%" << endl;
}
```

Stream parameter that can be replaced either with cout or with a file output stream

Sample Dialogue

```
Start of Test:
account1 initial statement:
Account balance $123.99
Interest rate 3.00%
account1 with new setup:
Account balance $100.00
Interest rate 5.00%
account1 after update:
Account balance $105.00
Interest rate 5.00%
account2:
Account balance $105.00
Interest rate 5.00%
```

```
{
    account1.balance = account1.balance +
    account1.fraction(account1.interest_rate) * account1.balance;
}
```

Notice that the call to the member function `fraction` is handled in the same way in this regard as the references to the member variables.

Input/output stream arguments

Like the classes we discussed earlier, the class `BankAccount` has a member function that outputs the data information stored in the object. In this program we are sending output to the screen. However, we want to write this class definition so that it can be copied into other programs and used unchanged in those other programs. Since some other program may want to send output to a file, we have given the member function `output` a formal parameter of type `ostream` so that the function `output` can be called with an argument that is either the stream `cout` or a file output stream. In the sample program we want the output to go to the screen, so the first function call to the member function `output` has the form

```
account1.output(cout);
```

Other calls to `output` also use `cout` as the argument, so all output is sent to the screen. If you want the output to go to a file instead, then you must first connect the file to an output stream, as we discussed in Chapter 6. If the file output stream is called `fout` and is connected to a file, then the following would write the data information for the object `account1` to this file rather than to the screen:

```
account1.output(fout);
```

Overloading member functions

The value of an object of type `BankAccount` represents a bank account that has some balance and pays some interest rate. The balance and interest rate can be set with the member function `set`. Notice that we have overloaded the member function named `set` so that there are two versions of `set`. One version has three formal parameters, and the other has only two formal parameters. Both versions have a formal parameter of type *`double`* for the interest rate, but the two versions of `set` use different formal parameters to set the account balance. One version has two formal parameters to set the balance, one for the dollars and one for the cents in the account balance. The other version has only a single formal parameter, which gives the number of dollars in the account and assumes that the number of cents is zero. This second version of `set` is handy, since most people open an account with some "even" amount of money, such as $1,000 and no cents. Notice that this overloading is nothing new. A member function is overloaded in the same way as an ordinary function is overloaded.

Summary of Some Properties of Classes

Classes have all of the properties that we described for structures plus all the properties associated with member functions. The following is a list of some points to keep in mind when using classes.

- Classes have both member variables and member functions.
- A member (either a member variable or a member function) may be either public or private.

- Normally, all the member variables of a class are labeled as private members.
- A private member of a class cannot be used except within the definition of another member function of the same class.
- The name of a member function for a class may be overloaded just like the name of an ordinary function.
- A class may use another class as the type for a member variable.
- A function may have formal parameters whose types are classes. (See Self-Test Exercises 19 and 20.)
- A function may return an object; that is, a class may be the type for the value returned by a function. (See Self-Test Exercise 21.)

Structures Versus Classes

Structures are normally used with all member variables being public and having no member functions. However, in C++ a structure can have private member variables and both public and private member functions. Aside from some notational differences, a C++ structure can do anything a class can do. Having said this and satisfied the "truth in advertising" requirement, we advocate that you forget this technical detail about structures. If you take this technical detail seriously and use structures in the same way that you use classes, then you have two names (with different syntax rules) for the same concept. On the other hand, if you use structures as we described them, then you will have a meaningful difference between structures (as you use them) and classes, and your usage will be the same as that of most other programmers.

SELF-TEST EXERCISES

19. Give a definition for the function with the following function declaration. The class `BankAccount` is defined in Display 10.5.

```
double difference(BankAccount account1, BankAccount account2);
//Precondition: account1 and account2 have been given values
//(that is, their member variables have been given values).
//Returns the balance in account1 minus the balance in
account2.
```

20. Give a definition for the function with the following function declaration. The class `BankAccount` is defined in Display 10.5. (*Hint:* It's easy if you use a member function.)

```
void double_update(BankAccount& the_account);
//Precondition: the_account has previously been given a value
//(that is, its member variables have been given values).
//Postcondition: The account balance has been changed so that
//two years' interest has been posted to the account.
```

21. Give a definition for the function with the following function declaration. The class `BankAccount` is defined in Display 10.5.

    ```
    BankAccount new_account(BankAccount old_account);
    //Precondition: old_account has previously been given a value
    //(that is, its member variables have been given values).
    //Returns the value for a new account that has a balance of zero
    //and the same interest rate as the old_account.
    ```

    For example, after this function is defined, a program could contain the following:

    ```
    BankAccount account3, account4;
    account3.set(999, 99, 5.5);
    account4 = new_account(account3);
    account4.output(cout);
    ```

    This would produce the following output:

    ```
    Account balance $0.00
    Interest rate 5.50%
    ```

Constructors for Initialization

You often want to initialize some or all the member variables for an object when you declare the object. As we will see later in this book, there are other initializing actions you might also want to take, but initializing member variables is the most common sort of initialization. C++ includes special provisions for such initializations. When you define a class, you can define a special kind of member function known as a **constructor.** A constructor is a member function that is automatically called when an object of that class is declared. A constructor is used to initialize the values of member variables and to do any other sort of initialization that may be needed. You can define a constructor the same way that you define any other member function, except for two points:

1. A constructor must have the same name as the class. For example, if the class is named `BankAccount`, then any constructor for this class must be named `BankAccount`.

2. A constructor definition cannot return a value. Moreover, no return type, not even *void*, can be given at the start of the function declaration or in the function header.

For example, suppose we wanted to add a constructor for initializing the balance and interest rate for objects of type `BankAccount` shown in Display 10.5. The class definition could be as follows. (We have omitted some of the comments to save space, but they should be included.)

```
class BankAccount
{
public:
    BankAccount(int dollars, int cents, double rate);
    //Initializes the account balance to $dollars.cents and
    //initializes the interest rate to rate percent.

    void set(int dollars, int cents, double rate);
    void set(int dollars, double rate);
    void update();

    double get_balance();
    double get_rate();
    void output(ostream& outs);
private:
    double balance;
    double interest_rate;
    double fraction(double percent);
};
```

Notice that the constructor is named `BankAccount`, which is the name of the class. Also notice that the function declaration for the constructor `BankAccount` does not start with *void* or with any other type name. Finally, notice that the constructor is placed in the public section of the class definition. Normally, you should make your constructors public member functions. If you were to make all your constructors private members, then you would not be able to declare any objects of that class type, which would make the class completely useless.

With the redefined class `BankAccount`, two objects of type `BankAccount` can be declared and initialized as follows:

```
BankAccount account1(10, 50, 2.0), account2(500, 0, 4.5);
```

Assuming that the definition of the constructor performs the initializing action that we promised, the previous declaration will declare the object `account1`, set the value of `account1.balance` to `10.50`, and set the value of `account1.interest_rate` to `2.0`. Thus, the object `account1` is initialized so that it represents a bank account with a balance of $10.50 and an interest rate of 2.0%. Similarly, `account2` is initialized so that it represents a bank account with a balance of $500.00 and an interest rate of 4.5%. What happens is that the object `account1` is declared and then the constructor `BankAccount` is called with the three arguments `10`, `50`, and `2.0`. Similarly, `account2` is declared and then the constructor `BankAccount` is called with the arguments `500`, `0`, and `4.5`. The result is conceptually equivalent to the following (although you cannot write it this way in C++):

```
BankAccount account1, account2; //PROBLEMS--BUT FIXABLE
account1.BankAccount(10, 50, 2.0); //VERY ILLEGAL
account2.BankAccount(500, 0, 4.5); //VERY ILLEGAL
```

As the comments indicate, you cannot place those three lines in your program. The first line can be made to be acceptable, but the two calls to the constructor BankAccount are illegal. A constructor cannot be called in the same way as an ordinary member function is called. Still, it is clear what we want to happen when we write those three lines, and that happens automatically when you declare the objects account1 and account2 as follows:

```
BankAccount account1(10, 50, 2.0), account2(500, 0, 4.5);
```

The definition of a constructor is given in the same way as any other member function. For example, if you revise the definition of the class BankAccount by adding the constructor just described, you need to also add the following definition of the constructor:

```
BankAccount::BankAccount(int dollars, int cents, double rate)
{
    if ((dollars < 0) || (cents < 0) || (rate < 0))
    {
        cout << "Illegal values for money or interest rate.\n";
        exit(1);
    }
    balance = dollars + 0.01*cents;
    interest_rate = rate;
}
```

Since the class and the constructor function have the same name, the name BankAccount occurs twice in the function heading: The BankAccount before the scope resolution operator :: is the name of the class, and the BankAccount after the scope resolution operator is the name of the constructor function. Also notice that no return type is specified in the heading of the constructor definition, not even the type *void*. Aside from these points, a constructor can be defined in the same way as an ordinary member function.

You can overload a constructor name like BankAccount::BankAccount, just as you can overload any other member function name, such as we did with BankAccount::set in Display 10.5. In fact, constructors usually are overloaded so that objects can be initialized in more than one way. For example, in Display 10.6 we have redefined the class BankAccount so that it has three versions of its constructor. This redefinition overloads the constructor name BankAccount so that it may have three arguments (as we just discussed), two arguments, or no arguments.

For example, suppose you give only two arguments when you declare an object of type BankAccount, as in the following example:

```
BankAccount account1(100, 2.3);
```

Then the object `account1` is initialized so that it represents an account with a balance of $100.00 and an interest rate of 2.3%.

On the other hand, if no arguments are given, as in the following example,

```
BankAccount account2;
```

then the object is initialized to represent an account with a balance of $0.00 and an interest rate of 0.0%. Notice that when the constructor has no arguments, you do not include any parentheses in the object declaration. The following is incorrect:

```
BankAccount account2(); //WRONG! DO NOT DO THIS!
```

In some cases, you can omit mutator member functions such as `set` once you have a good set of constructor definitions. You can use the overloaded constructor `BankAccount` in Display 10.6 to create a new `BankAccount` object with the values of your choice. However, invoking the constructor will create a new object, so if you want to change the existing member variables in the object, then you should use a mutator function.

DISPLAY 10.6 Class with Constructors *(part 1 of 3)*

```
//Program to demonstrate the class BankAccount.
#include <iostream>
using namespace std;

//Class for a bank account:
class BankAccount
{
public:
    BankAccount(int dollars, int cents, double rate);
    //Initializes the account balance to $dollars.cents and
    //initializes the interest rate to rate percent.

    BankAccount(int dollars, double rate);
    //Initializes the account balance to $dollars.00 and
    //initializes the interest rate to rate percent.

    BankAccount( );
    //Initializes the account balance to $0.00
    //and the interest rate to 0.0%.

    void set(int dollars, int cents, double rate);
    //Postcondition: The account balance has been set to $dollars.cents;
    //The interest rate has been set to rate percent.

    void set(int dollars, double rate);
    //Postcondition: The account balance has been set to $dollars.00.
    //The interest rate has been set to rate percent.

    void update( );
```

This definition of `BankAccount` is an improved version of the class `BankAccount` given in Display 10.5.

(continued)

DISPLAY 10.6 Class with Constructors *(part 2 of 3)*

```
        //Postcondition: One year of simple interest has been added
        //to the account balance.

        double get_balance( );
        //Returns the current account balance.

        double get_rate( );
        //Returns the current account interest rate as a percentage.

        void output(ostream& outs);
        //Precondition: If outs is a file output stream, then
        //outs has already been connected to a file.
        //Postcondition: Account balance and interest rate
        //have been written to the stream outs.
    private:
        double balance;
        double interest_rate;

        double fraction(double percent);
        //Converts a percentage to a fraction. For example, fraction(50.3)
        //returns 0.503.
    };

    int main( )
    {
        BankAccount account1(100, 2.3), account2;

        cout << "account1 initialized as follows:\n";
        account1.output(cout);
        cout << "account2 initialized as follows:\n";
        account2.output(cout);

        account1 = BankAccount(999, 99, 5.5);
        cout << "account1 reset to the following:\n";
        account1.output(cout);
        return 0;
    }

    BankAccount::BankAccount(int dollars, int cents, double rate)
    {
        if ((dollars < 0) || (cents < 0) || (rate < 0))
        {
            cout << "Illegal values for money or interest rate.\n";
            exit(1);
        }
        balance = dollars + 0.01 * cents;
        interest_rate = rate;
    }

    BankAccount::BankAccount(int dollars, double rate)
    {
```

This declaration causes a call to the default constructor. Notice that there are no parentheses.

An explicit call to the constructor BankAccount::BankAccount

(continued)

DISPLAY 10.6 Class with Constructors *(part 3 of 3)*

```
        if ((dollars < 0) || (rate < 0))
        {
            cout << "Illegal values for money or interest rate.\n";
            exit(1);
        }
        balance = dollars;
        interest_rate = rate;
    }

    BankAccount::BankAccount( ) : balance(0), interest_rate(0.0)
    {
        //Body intentionally empty
    }
```

<Definitions of the other member functions are the same as in Display 10.5.

Screen Output

```
account1 initialized as follows:
Account balance $100.00
Interest rate 2.30%
account2 initialized as follows:
Account balance $0.00
Interest rate 0.00%
account1 reset to the following:
Account balance $999.99
Interest rate 5.50%
```

Constructor

A **constructor** is a member function of a class that has the same name as the class. A constructor is called automatically when an object of the class is declared. Constructors are used to initialize objects. A constructor must have the same name as the class of which it is a member.

The constructor with no parameters in Display 10.6 deserves some extra discussion since it contains something we have not seen before. For reference, we reproduce the defining of the constructor with no parameters:

```
BankAccount::BankAccount() : balance(0), interest_rate(0.0)
{
    //Body intentionally empty
}
```

The new element, which is shown on the first line, is the part that starts with a single colon. This part of the constructor definition is called the **initialization section.** As this example shows, the initialization section goes after the parentheses that ends the parameter list and before the opening brace of the function body. The initialization section consists of a colon followed by a list of some or all the member variables separated by commas. Each member variable is followed by its initializing value in parentheses. This constructor definition is completely equivalent to the following way of writing the definition:

```
BankAccount::BankAccount( )
{
    balance = 0;
    interest_rate = 0.0;
}
```

The function body in a constructor definition with an initialization section need not be empty. For example, the following definition of the two-parameter constructor is equivalent to the one given in Display 10.6:

```
BankAccount::BankAccount(int dollars, double rate)
           : balance(dollars), interest_rate(rate)
{
    if ((dollars < 0) || (rate < 0))
    {
        cout << "Illegal values for money or interest rate.\n";
        exit(1);
    }
}
```

Notice that the initializing values can be given in terms of the constructor parameters.

Constructor Initialization Section

Some or all of the member variables in a class can (optionally) be initialized in the **constructor initialization section** of a constructor definition. The constructor initialization section goes after the parentheses that end the parameter list and before the opening brace of the function body. The initialization section consists of a colon followed by a list of some or all the member variables separated by commas. Each member variable is followed by its initializing value in parentheses. The example given below uses a constructor initialization section and is equivalent to the three-parameter constructor given in Display 10.6.

(continued)

EXAMPLE

```
BankAccount::BankAccount(int dollars, int cents,
                         double rate)
    : balance(dollars + 0.01*cents), interest_rate(rate)
{
    if ((dollars < 0) || (cents < 0) || (rate < 0))
    {
        cout <<
           "Illegal values for money or interest rate.\n";
        exit(1);
    }
}
```

Notice that the initializing values can be given in terms of the constructor parameters.

Calling a Constructor

A constructor is called automatically when an object is declared, but you must give the arguments for the constructor when you declare the object. A constructor can also be called explicitly in order to create a new object for a class variable.

SYNTAX (for an object declaration when you have constructors)

```
Class_Name Object_Name(Arguments_for_Constructor);
```

EXAMPLE

```
BankAccount account1(100, 2.3);
```

SYNTAX (for an explicit constructor call)

```
Object = Constructor_Name(Arguments_For_Constructor);
```

EXAMPLE

```
account1 = BankAccount(200, 3.5);
```

A constructor must have the same name as the class of which it is a member. Thus, in the syntax descriptions above, *Class_Name* and *Constructor_Name* are the same identifier.

Initializers can also be specified if the object is created as a dynamic variable.

```
BankAccount *myAcct; myAcct = new BankAccount (300, 4.2);
```

A constructor is called automatically whenever you declare an object of the class type, but it can also be called again after the object has been declared. This allows you to conveniently set all the members of an object. The technical details are as follows. Calling the constructor creates an anonymous object with new values. An anonymous object is an object that is not named (as yet) by any variable. The anonymous object can be assigned to the named object (that is, to the class variable). For example, the following line of code is a call to the constructor `BankAccount` that creates an anonymous object with a balance of `$999.99` and interest rate of `5.5%`. This anonymous object is assigned to object `account1` so that it too represents an account with a balance of $999.99 and an interest rate of 5.5%:

```
account1 = BankAccount(999, 99, 5.5);
```

As you might guess from the notation, a constructor behaves like a function that returns an object of its class type. However, since a call to a constructor always creates a new object and a call to a set member function merely changes the values of existing member variables, a call to set may be a more efficient way to change the values of member variables than a call to a constructor. Thus, for efficiency reasons or if you need to change the values of member variables without creating a new object, you may wish to have both the set member functions and the constructors in your class definition.

PROGRAMMING TIP Always Include a Default Constructor

C++ does not always generate a default constructor for the classes you define. If you give no constructor, the compiler will generate a default constructor that does nothing. This constructor will be called if class objects are declared. On the other hand, if you give at least one constructor definition for a class, then the C++ compiler will generate no other constructors. Every time you declare an object of that type, C++ will look for an appropriate constructor definition to use. If you declare an object without using arguments for the constructor, C++ will look for a default constructor, and if you have not defined a default constructor, none will be there for it to find.

For example, suppose you define a class as follows:

```
class SampleClass
{                                  Constructor that requires two arguments
public:
    SampleClass(int parameter1, double parameter2);
    void do_stuff();
private:
    int data1;
    double data2;
};
```

You should recognize the following as a legal way to declare an object of type SampleClass and call the constructor for that class:

```
SampleClass my_object(7, 7.77);
```

However, you may be surprised to learn that the following is illegal:

```
SampleClass your_object;
```

The compiler interprets this declaration as including a call to a constructor with no arguments, but there is no definition for a constructor with zero arguments. You must either add two arguments to the declaration of your_object or add a constructor definition for a constructor with no arguments.

A constructor that can be called with no arguments is called a **default constructor**, since it applies in the default case where you declare an object without specifying any arguments. Since it is likely that you will sometimes want to declare an object without giving any constructor arguments, you should always include a default constructor. The following redefined version of SampleClass includes a default constructor:

```
class SampleClass
{
public:
    SampleClass(int parameter1, double parameter2);
    SampleClass();          Default constructor
    void do_stuff();
private:          <------
    int data1;
    double data2;
};
```

If you redefine the class SampleClass in this manner, then the previous declaration of your_object would be legal.

If you do not want the default constructor to initialize any member variables, you can simply give it an empty body when you implement it. The following constructor definition is perfectly legal. It does nothing when called except make the compiler happy:

```
SampleClass::SampleClass()
{
    //Do nothing.
}
```

Note that if a class is created as a dynamic variable using the new operator then the default constructor is invoked. ■

PITFALL Constructors with No Arguments

If a constructor for a class called BankAccount has two formal parameters, you declare an object and give the arguments to the constructor as follows:

```
BankAccount account1(100, 2.3);
```

To call the constructor with no arguments, you would naturally think that you would declare the object as follows:

```
BankAccount account2(); //THIS WILL CAUSE PROBLEMS.
```

After all, when you call a function that has no arguments, you include a pair of empty parentheses. However, this is wrong for a constructor. Moreover, it may not produce an error message, since it does have an unintended meaning. The compiler will think that this code is the function declaration for a function called `account2` that takes no arguments and returns a value of type `BankAccount`.

Do not include parentheses when you declare an object and want C++ to use the constructor with no arguments. The correct way to declare `account2` using the constructor with no arguments is as follows:

```
BankAccount account2;
```

However, if you explicitly call a constructor in an assignment statement, you do use the parentheses. If the definitions and declarations are as in Display 10.6, then the following will set the account balance for `account1` to $0.00 and set the interest rate to 0.0%:

```
account1 = BankAccount();
```

Constructors with No Arguments

When you declare an object and want the constructor with zero arguments to be called, you do not include any parentheses. For example, to declare an object and pass two arguments to the constructor, you might do the following:

```
BankAccount account1(100, 2.3);
```

However, if you want the constructor with zero arguments to be used, declare the object as follows:

```
BankAccount account1;
```

You do *not* declare the object as follows:

```
BankAccount account1(); //INCORRECT DECLARATION
```

(The problem is that this syntax declares a function named `account1` that returns a `BankAccount` object and has no parameters.)

■

Member Initializers and Constructor Delegation in C++11

VideoNote
Default Initialization of Member Variables

C++11 supports a feature called *member initialization* that is present in most object-oriented programming languages. This feature allows you to set default values for member variables. When an object is created the member variables are automatically initialized to the specified values. Consider the following definition and implementation of the Coordinate class:

```
class Coordinate
{
  public:
        Coordinate();
        Coordinate(int x);
        Coordinate(int x, int y);
        int getX();
        int getY();
  private:
        int x=1;
        int y=2;
};
Coordinate::Coordinate()
{ }
Coordinate::Coordinate(int xval) : x(xval)
{ }
Coordinate::Coordinate(int xval, int yval) : x(xval), y(yval)
{ }
int Coordinate::getX()
{
  return x;
}
int Coordinate::getY()
{
  return y;
}
```

If we create a Coordinate object, then member variable x will be set to 1 and member variable y will be set to 2 by default. These values can be overridden if we invoke a constructor that explicitly sets the variable. In the snippet below, the default values for x and y are set for c1, but for c2 the default value is only set for y because x is explicitly set to the input argument:

```
Coordinate c1, c2(10);
cout << c1.getX() << " " << c1.getY() << endl; // Outputs 1 2
cout << c2.getX() << " " << c2.getY() << endl; // Outputs 10 2
```

A related feature supported by C++11 is *constructor delegation*. Simply put, this allows one constructor to call another constructor. For example, we could modify the implementation of the default constructor so it invokes the constructor with two parameters:

```
Coordinate::Coordinate() : Coordinate(99,99)
{ }
```

The object defined by `Coordinate c1;` will invoke the default constructor which will in turn invoke the constructor to set x to 99 and y to 99.

SELF-TEST EXERCISES

22. Suppose your program contains the following class definition (along with definitions of the member functions):

```
class YourClass
{
public:
    YourClass(int new_info, char more_new_info);
    YourClass();
    void do_stuff();
private:
    int information;
    char more_information;
};
```

Which of the following are legal?

```
YourClass an_object(42, 'A');
YourClass another_object;
YourClass yet_another_object();
an_object = YourClass(99, 'B');
an_object = YourClass();
an_object = YourClass;
```

23. How would you change the definition of the class `DayOfYear` in Display 10.4 so that it has two versions of an (overloaded) constructor? One version should have two *int* formal parameters (one for the month and one for the day) and should set the private member variables to represent that month and day. The other should have no formal parameters and should set the date represented to January 1. Do this without using a constructor initialization section in either constructor.

24. Redo the previous exercise, but this time use a constructor initialization section to initialize all member functions in each constructor.

10.3 ABSTRACT DATA TYPES

We all know — the Times knows — but we pretend we don't.

VIRGINIA WOOLF, *Monday or Tuesday*

A data type, such as the type *int*, has certain specified values, such as 0, 1, −1, 2, and so forth. You tend to think of the data type as being these values, but the operations on these values are just as important as the values. Without the operations, you could do nothing of interest with the values. The operations for the type *int* consist of +, −, *, /, %, and a few other operators and predefined library functions. You should not think of a data type as being simply a collection of values. A **data type** consists of a collection of values together with a set of basic operations defined on those values.

A data type is called an **abstract data type** (abbreviated ADT) if the programmers who use the type do not have access to the details of how the values and operations are implemented. The predefined types, such as *int*, are abstract data types (ADTs). You do not know how the operations, such as + and *, are implemented for the type *int*. Even if you did know, you would not use this information in any C++ program.

Programmer-defined types, such as the structure types and class types, are not automatically ADTs. Unless they are defined and used with care, programmer-defined types can be used in unintuitive ways that make a program difficult to understand and difficult to modify. The best way to avoid these problems is to make sure all the data types that you define are ADTs. The way that you do this in C++ is to use classes, but not every class is an ADT. To make it an ADT you must define the class in a certain way, and that is the topic of the next subsection.

Classes to Produce Abstract Data Types

A class is a type that you define, as opposed to the types, such as *int* and *char*, that are already defined for you. A value for a class type is the set of values of the member variables. For example, a value for the type BankAccount in Display 10.6 consists of two numbers of type *double*. For easy reference, we repeat the class definition (omitting only the comments):

```
class BankAccount
{
public:
    BankAccount(int dollars, int cents, double rate);
    BankAccount(int dollars, double rate);
    BankAccount();
    void set(int dollars, int cents, double rate);
    void set(int dollars, double rate);
    void update();
    double get_balance();
    double get_rate();
    void output(ostream& outs);
private:
    double balance;
    double interest_rate;
    double fraction(double percent);
};
```

The programmer who uses the type `BankAccount` need not know how you implemented the definition of `BankAccount::update` or any of the other member functions. The function definition for the member function `BankAccount::update` that we used is as follows:

```
void BankAccount::update()
{
    balance = balance + fraction(interest_rate) * balance;
}
```

However, we could have dispensed with the private function `fraction` and implemented the member function `update` with the following slightly more complicated formula:

```
void BankAccount::update()
{
    balance = balance + (interest_rate / 100.0) * balance;
}
```

The programmer who uses the class `BankAccount` need not be concerned with which implementation of `update` we used, since both implementations have the same effect.

Similarly, the programmer who uses the class `BankAccount` need not be concerned about how the values of the class are implemented. We chose to implement the values as two values of type *`double`*. If `vacation_savings` is an object of type `BankAccount`, the value of `vacation_savings` consists of the two values of type *`double`* stored in the following two member variables:

```
vacation_savings.balance
vacation_savings.interest_rate
```

However, you do not want to think of the value of the object `vacation_savings` as two numbers of type *`double`*, such as `1.3546e + 2` and `4.5`. You want to think of the value of `vacation_savings` as the single entry

Account balance $135.46
Interest rate 4.50%

That is why our implementation of `BankAccount::output` writes the class value in this format.

The fact that we chose to implement this `BankAccount` value as the two *`double`* values `1.3546e + 2` and `4.5` is an implementation detail. We could instead have implemented this `BankAccount` value as the two *`int`* values `135` and `46` (for the dollars and cents part of the balance) and the single value `0.045` of type *`double`*. The value `0.045` is simply 4.5% converted to a fraction, which might be a more useful way to implement a percentage figure. After all, in order to compute interest on the account we convert a percentage to just such a fraction. With this alternative implementation of the class `BankAccount`, the public members would remain unchanged but the private members would change to the following:

```
class BankAccount
{
public:
    <This part is exactly the same as before>
private:
    int dollars_part;
    int cents_part;
    double interest_rate;
    double fraction(double percent);
};
```

We would need to change the member function definitions to match this change, but that is easy to do. For example, the function definitions for `get_balance` and one version of the constructor could be changed to the following:

```
double BankAccount::get_balance()
{
    return (dollars_part + 0.01 * cents_part);
}

BankAccount::BankAccount(int dollars, int cents, double rate)
{
    if ((dollars < 0) || (cents < 0) || (rate < 0))
    {
        cout << "Illegal values for money or interest rate.\n";
        exit(1);
    }
    dollars_part = dollars;
    cents_part = cents;
    interest_rate = rate;
}
```

Similarly, each of the other member functions could be redefined to accommodate this new way of storing the account balance and the interest rate.

Notice that even though the user may think of the account balance as a single number, that does not mean the implementation has to be a single number of type *`double`*. You have just seen that it could, for example, be two numbers of type *`int`*. The programmer who uses the type `BankAccount` need not know any of this detail about how the values of the type `BankAccount` are implemented.

How to write an ADT

These comments about the type `BankAccount` illustrate the basic technique for defining a class so that it will be an abstract data type. In order to define a class so that it is an abstract data type, you need to separate the specification of how the type is used by a programmer from the details of how the type is implemented. The separation should be so complete that you can change the implementation of the class without needing to make any changes in any program that uses the class ADT. One way to ensure this separation is to follow these rules:

1. Make all the member variables private members of the class.

2. Make each of the basic operations that the programmer needs a public member function of the class, and fully specify how to use each such public member function.

3. Make any helping functions private member functions.

In Chapters 11 and 12 you will learn some alternative approaches to defining ADTs, but these three rules are one common way to ensure that a class is an abstract data type.

The **interface** of an ADT tells you how to use the ADT in your program. When you define an ADT as a C++ class, the interface consists of the public member functions of the class along with the comments that tell you how to use these public member functions. The interface of the ADT should be all you need to know in order to use the ADT in your program.

The **implementation** of the ADT tells how this interface is realized as C++ code. The implementation of the ADT consists of the private members of the class and the definitions of both the public and private member functions. Although you need the implementation in order to run a program that uses the ADT, you should not need to know anything about the implementation in order to write the rest of a program that uses the ADT; that is, you should not need to know anything about the implementation in order to write the `main` part of the program and to write any nonmember functions used by the `main` part of the program. The situation is similar to what we advocated for ordinary function definitions in Chapters 4 and 5. The implementation of an ADT, like the implementation of an ordinary function, should be thought of as being in a black box that you cannot see inside.

Separate interface and implementation

VideoNote
Separate Interface and Implementation

In Chapter 12 you will learn how to place the interface and implementation of an ADT in files separate from each other and separate from the programs that use the ADT. That way a programmer who uses the ADT literally does not see the implementation. Until then, we will place all of the details about our ADT classes in the same file as the `main` part of our program, but we still think of the interface (given in the public section of the class definitions) and the implementation (the private section of the class definition and the member function definitions) as separate parts of the ADT. We will strive to write our ADTs so that the user of the ADT need only know about the interface of the ADT and need not know anything about the implementation. To be sure you are defining your ADTs this way, simply make sure that if you change the implementation of your ADT, your program will still work without your needing to change any other part of the program. This is illustrated in the next Programming Example.

The most obvious benefit you derive from making your classes ADTs is that you can change the implementation without needing to change the other parts of your program. But ADTs provide more benefits than that. If you make your classes ADTs, you can divide work among different programmers, with one programmer designing and writing the ADT and other programmers using the ADT. Even if you are the only programmer working on a project, you have divided one larger task into two smaller tasks, which makes your program easier to design and easier to debug.

PROGRAMMING EXAMPLE Alternative Implementation of a Class

Display 10.7 contains the alternative implementation of the ADT class `BankAccount` discussed in the previous subsection. In this version, the data for a bank account is implemented as three member values: one for the dollars part of the account balance, one for the cents part of the account balance, and one for the interest rate.

Notice that, although both the implementation in Display 10.6 and the implementation in Display 10.7 each have a member variable called `interest_rate`, the value stored is slightly different in the two implementations. If the account pays interest at a rate of 4.7%, then in the implementation in Display 10.6 (which is basically the same as the one in Display 10.5), the value of `interest_rate` is `4.7`. However, in the implementation in Display 10.7, the value of `interest_rate` would be `0.047`. This alternative implementation, shown in Display 10.7, stores the interest rate as a fraction rather than as a percentage figure. The basic difference in this new implementation is that when an interest rate is set, the function `fraction` is used to immediately convert the interest rate to a fraction. Hence, in this new implementation the private member function `fraction` is used in the definitions of constructors, but it is not needed in the definition of the member function `update` because the value in the member variable `interest_rate` has already been converted to a fraction. In the old implementation (shown in Display 10.5 and Display 10.6), the situation was just the reverse. In the old implementation, the private member function `fraction` was not used in the definition of constructors, but was used in the definition of `update`.

The public interface is not changed

Although we have changed the private members of the class `BankAccount`, we have not changed anything in the public section of the class definition. The public member functions have the same function declarations and they behave exactly as they did in the old version of the ADT class given in Display 10.6. For example, although this new implementation stores a percentage such as 4.7% as the fraction `0.047`, the member function `get_rate` still returns the value `4.7`, just as it would for the old implementation in Display 10.5. Similarly, the member function `get_balance` returns a single value of type *`double`*, which gives the balance as a number with a decimal point, just as it did in the old implementation in Display 10.5. This is true even though the balance is now stored in two member variables of type *`int`*, rather than in a single member variable of type *`double`* (as in the old versions).

DISPLAY 10.7 Alternative BankAccount Class Implementation *(part 1 of 4)*

```
//Demonstrates an alternative implementation of the class BankAccount.
#include <iostream>
#include <cmath>
using namespace std;
//Class for a bank account:
class BankAccount
{
public:
    BankAccount(int dollars, int cents, double rate);
    //Initializes the account balance to $dollars.cents and
    //initializes the interest rate to rate percent.

    BankAccount(int dollars, double rate);
    //Initializes the account balance to $dollars.00 and
    //initializes the interest rate to rate percent.

    BankAccount( );
    //Initializes the account balance to $0.00 and the
    //interest rate to 0.0%.

    void set(int dollars, int cents, double rate);
    //Postcondition: The account balance has been set to $dollars.cents;
    //The interest rate has been set to rate percent.

    void set(int dollars, double rate);
    //Postcondition: The account balance has been set to $dollars.00.
    //The interest rate has been set to rate percent.

    void update( );
    //Postcondition: One year of simple interest has been
    //added to the account balance.

    double get_balance( );
    //Returns the current account balance.

    double get_rate( );
    //Returns the current account interest rate as a percentage.

    void output(ostream& outs);
    //Precondition: If outs is a file output stream, then
    //outs has already been connected to a file.
    //Postcondition: Account balance and interest rate
    //have been written to the stream outs.
private:
    int dollars_part;
    int cents_part;
    double interest_rate;
    //Expressed as a fraction, for example, 0.057 for 5.7%
```

Notice that the public members of BankAccount *look and behave exactly the same as in Display 10.6*

(continued)

DISPLAY 10.7 Alternative BankAccount Class Implementation *(part 2 of 4)*

```
    double fraction(double percent);
    //Converts a percentage to a fraction. For example, fraction(50.3)
    //returns 0.503.
    double percent(double fraction_value);
    //Converts a fraction to a percentage. For example, percent(0.503)
    //returns 50.3.
};
int main( )
{
    BankAccount account1(100, 2.3), account2;

    cout << "account1 initialized as follows:\n";
    account1.output(cout);
    cout << "account2 initialized as follows:\n";
    account2.output(cout);

    account1 = BankAccount(999, 99, 5.5);
    cout << "account1 reset to the following:\n";
    account1.output(cout);
    return 0;
}
BankAccount::BankAccount(int dollars, int cents, double rate)
{
    if ((dollars < 0) || (cents < 0) || (rate < 0))
    {
        cout << "Illegal values for money or interest rate.\n";
        exit(1);
    }
    dollars_part = dollars;
    cents_part = cents;
    interest_rate = fraction(rate);
}
BankAccount::BankAccount(int dollars, double rate)
{
    if ((dollars < 0) || (rate < 0))
    {
        cout << "Illegal values for money or interest rate.\n";
        exit(1);
    }
    dollars_part = dollars;
    cents_part = 0;
    interest_rate = fraction(rate);
}
BankAccount::BankAccount( ) : dollars_part(0), cents_part(0), interest_rate(0.0)

```

New

Since the body of `main` *is identical to that in Display 10.6, the screen output is also identical to that in Display 10.6*

In the old implementation of this ADT, the private member function `fraction` *was used in the definition of* `update`*. In this implementation,* `fraction` *is instead used in the definition of constructors and in the set function.*

(continued)

DISPLAY 10.7 Alternative BankAccount Class Implementation *(part 3 of 4)*

```
{
    //Body intentionally empty.
}
double BankAccount::fraction(double percent_value)
{
    return (percent_value/100.0);
}
//Uses cmath:
void BankAccount::update( )
{
    double balance = get_balance( );
    balance = balance + interest_rate * balance;
    dollars_part = static_cast<int>(floor(balance));
    cents_part = static_cast<int>(floor((balance - dollars_part)*100));
}
double BankAccount::get_balance( )
{
    return (dollars_part + 0.01 * cents_part);
}
double BankAccount::percent(double fraction_value)
{
    return (fraction_value * 100);
}
double BankAccount::get_rate( )
{
    return percent(interest_rate);
}
//Uses iostream:
void BankAccount::output(ostream& outs)
{
    outs.setf(ios::fixed);
    outs.setf(ios::showpoint);
    outs.precision(2);
    outs << "Account balance $" << get_balance( ) << endl;
    outs << "Interest rate "<< get_rate( ) << "%" << endl;
}
void BankAccount::set(int dollars, int cents, double rate)
{
    if ((dollars < 0) || (cents < 0) || (rate < 0))
    {
        cout << "Illegal values for money or interest rate.\n";
        exit(1);
    }
```

The new definitions of `get_balance` *and* `get_rate` *ensure that the output will still be in the correct units.*

(continued)

DISPLAY 10.7 Alternative `BankAccount` Class Implementation *(part 4 of 4)*

```
        dollars_part = dollars;
        cents_part = cents;
        interest_rate = fraction(rate);
    }
    void BankAccount::set(int dollars, double rate)
    {
        if ((dollars < 0) || (rate < 0))
        {
            cout << "Illegal values for money or interest rate.\n";
            exit(1);
        }
        dollars_part = dollars;
        interest_rate = fraction(rate);
    }
```

Changing private member functions

Notice that there is an important difference between how you treat the public member functions and how you treat the private member functions. If you want to preserve the interface of an ADT class so that any programs that use it need not change (other than changing the definitions of the class and its member functions), then you must leave the public member function declarations unchanged. However, you are free to add, delete, or change any of the private member functions. In this example, we have added one additional private function called `percent`, which is the inverse of the function `fraction`. The function `fraction` converts a percentage to a fraction, and the function `percent` converts a fraction back to a percentage. For example, `fraction(4.7)` returns `0.047`, and `percent(0.047)` returns `4.7`.

Information Hiding

We discussed information hiding when we introduced functions in Chapter 3. We said that **`information hiding,`** as applied to functions, means that you should write your functions so that they could be used as black boxes, that is, so that the programmer who uses the function need not know any details about how the function is implemented. This principle means that all the programmer who uses a function needs to know is the function declaration and the accompanying comment that explains how to use the function. The use of private member variables and private member functions in the definition of an abstract data type is another way to implement information hiding, but now we apply the principle to data values as well as to functions.

SELF-TEST EXERCISES

25. When you define an ADT as a C++ class, should you make the member variables public or private? Should you make the member functions public or private?

26. When you define an ADT as a C++ class, what items are considered part of the interface for the ADT? What items are considered part of the implementation for the ADT?

27. Suppose your friend defines an ADT as a C++ class in the way we described in Section 10.3. You are given the task of writing a program that uses this ADT. That is, you must write the `main` part of the program as well as any nonmember functions that are used in the `main` part of the program. The ADT is very long and you do not have a lot of time to write this program. What parts of the ADT do you need to read and what parts can you safely ignore?

28. Redo the three- and two-parameter constructors in Display 10.7 so that all member variables are set using a constructor initialization section.

10.4 INTRODUCTION TO INHERITANCE

One of the most powerful features of C++ is the use of *derived classes*. The word *inheritance* is just another name for the topic of derived classes. When we say that one class was derived from another class, we mean that the derived class was obtained from the other class by adding features. For example, suppose we define a class for vehicles that has member variables to record the vehicle's number of wheels and maximum number of occupants. The class also has accessor and mutator functions. Imagine that we then define a class for automobiles that has member variables and functions just like the ones in the class of vehicles. In addition, our automobile class would have added member variables for such things as the amount of fuel in the fuel tank and the license plate number and would also have some added member functions. Instead of repeating the definitions of the member variables and functions of the class of vehicles within the class of automobiles, we could use C++'s inheritance mechanism and let the automobile class inherit all the member variables and functions of the class for vehicles.

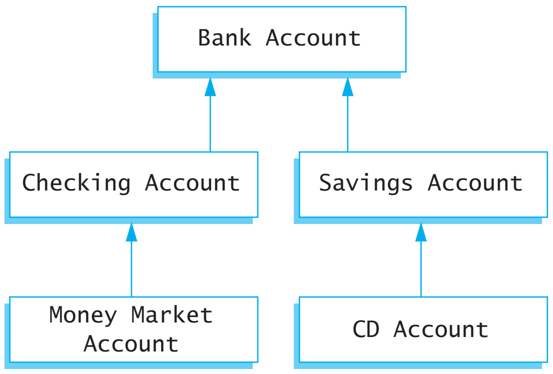
Inheritance allows you to define a general class and then later define more specialized classes that add some new details to the existing general class. This saves work because the more specialized, or derived, class inherits all the properties of the general class and you, the programmer, need only program the new features. This section will first introduce the notion of inheritance and a derived class and then we briefly describe how to create your own derived classes. Details of inheritance are left to Chapter 15. It may take a while before you are completely comfortable with the idea of a derived class, but you easily can learn enough about derived classes to start using them in some simple, and very useful, ways.

Derived Classes

Consider the `BankAccount` class defined in Display 10.7. This class keeps track of an amount and interest rate for a bank account—fairly generic features that apply to any interest-bearing account. If we would like to implement more specific types of bank accounts, then there is a natural hierarchy for grouping the account types. Display 10.8 depicts a part of this hierarchical arrangement for bank accounts, checking accounts, money market accounts, savings accounts, and Certificate of Deposit (CD) accounts. In the hierarchy, `BankAccount` is the most general type of account; more specific types of accounts are shown underneath. An arrow points from a specific account type to a more general account type. In addition to representing different types of bank accounts, each box also corresponds to a class that we can implement in C++.

For example, a checking account does everything a bank account can do (store an amount and interest rate) but in addition allows customers to make deposits and write checks. Similarly, a savings account does everything a bank account can do but in addition allows customers to make deposits and withdrawals. Unlike a checking account, a savings account may not allow customers to write checks. Since both checking accounts and savings accounts are types of bank accounts they are shown in Display 10.8 directly underneath the `BankAccount` class. When we say that some class A is a **derived class** of some other class B, it means that class A has all the features of class B but it also has *added features*. The convention for indicating this relationship in a diagram is to draw an an unfilled arrow from the specific to the more general class. For example, in Display 10.8 the `CheckingAccount` and `SavingsAccount` classes are derived classes of the `BankAccount` class.

DISPLAY 10.8 A Class Hierarchy



In C++, some class **A** can be a derived class of some other class **B,** which in turn can be a derived class of some other class **C,** and so on. For example, a CD account is similar to a savings account except the funds and any accrued interest must not be withdrawn until after a "maturity" date. If the funds are withdrawn prior to the maturity date, then there is a penalty. Due to these restrictions, a CD account normally accrues interest at a higher rate than a savings account. In the hierarchy, this is shown by deriving `CDAccount` from `SavingsAccount`. Similarly, a money market account is a special type of checking account in which the customer normally has a limit on the number of checks that can be written, along with higher minimum balances, but pays a higher interest rate. In the hierarchy, this is shown by deriving `MoneyMarketAccount` from `CheckingAccount`.

Derived classes are often discussed using the metaphor of inheritance and family relationships. If class B is a derived class of class A, then class B is called a **child** of class A and class A is called a **parent** of class B. The parent class is also referred to as the **base** class. The derived class is said to **inherit** the member functions of its parent class. For example, every convertible inherits the fact that it has four wheels from the class of all automobiles. This is why the topic of derived classes is often called *inheritance*.

Defining Derived Classes

If we want to create a class to represent a savings account, we could start by making a copy of the `BankAccount` class and renaming it to `SavingsAccount`. We would need to add new public member functions to deposit and withdraw funds. While this approach would work, it would be very inefficient, because the `SavingsAccount` class would duplicate most of the functionality in the `BankAccount` class. Not only does this waste memory space, it also becomes more difficult to make modifications. For example, if we later decide to change the `update( )` function to accrue interest daily instead of annually, then we would have two places to make the change: in the `SavingsAccount` class and also in the `BankAccount` class. These problems can be solved by defining the `SavingsAccount` class as a derived class of the `BankAccount` class. The `SavingsAccount` class then can share member variables and functions defined in the `BankAccount` class. We specify this relationship when defining the derived class by adding a colon followed by the keyword `public` and the name of the parent or base class:

The colon separates the derived class, `SavingsAccount`*, from the parent class,* `BankAccount`

```
class SavingsAccount : public BankAccount
{
public:
    SavingsAccount(int dollars, int cents, double rate);
    <Other constructors would normally go here>
    void deposit(int dollars, int cents);
    void withdraw(int dollars, int cents);
private:
};
```

Notice that we only defined functions and data that specifically relate to savings accounts, in this case, functions to deposit and withdraw money. We don't need to redefine all of the variables and functions relating to bank accounts—such as storing the interest rate, dollars, cents, or defining the `update( )` function—because those members will be inherited from the `BankAccount` class and are automatically created when we construct a `SavingsAccount` object. For example, if we create a `SavingsAccount` object, we could invoke the following functions:

```
SavingsAccount account(100, 50, 5.5);
account.deposit(10,25);
account.output(cout);
```

Invoking a function in the derived class,

Invoking a function in the parent class, `BankAccount`

In this example, inheritance allowed us to reuse code defined in the parent class from the context of the derived class. Moreover, if we later change one of `BankAccount's` functions—such as `update( )`—then the new code automatically will be used from the context of its derived classes when the program is recompiled and linked. An implementation of the `SavingsAccount` class along with a `main` function to test the `deposit` and `withdraw` functions is given in Display 10.9. For simplicity, we have left verification out of the `deposit` and `withdraw` functions, for example, checking for negative amounts, but you should be able to add them easily with some *`if`* statements.

Once the `SavingsAccount` class is defined we can go one step further and derive more specialized classes from the `SavingsAccount`. For example, to define the CD account class we need a new private member variable to store the days until maturity and define functions to access this variable:

```
class CDAccount : public SavingsAccount
{
public:
    CDAccount(int dollars, int cents, double rate,
              int days_to_maturity);
    <Other constructors would normally go here>
    int get_days_to_maturity( );
    //Returns the number of days until the CD matures
    void decrement_days_to_maturity( );
    //Subtracts one from the days_to_maturity variable
private:
    int days_to_maturity; //Days until the CD matures
};
```

Once again, we only defined functions and data that specifically relate to CD accounts, in this case, storing and manipulating the number of days to maturity. We don't need to redefine all of the variables and functions relating to bank accounts or savings accounts because those members will be inherited from the parent classes. For example, once the functions in the `CDAccount` class are implemented, we could invoke the following functions from the `CDAccount`, `SavingsAccount`, or `BankAccount` classes given a `CDAccount` object:

DISPLAY 10.9 A `SavingsAccount` Derived Class *(part 1 of 2)*

<Everything from Display 10.6 should be inserted here except for the `main` *function.>*

```
class SavingsAccount : public BankAccount
{
public:

    SavingsAccount(int dollars, int cents, double rate);
    //Other constructors would go here
    void deposit(int dollars, int cents);
    //Adds $dollars.cents to the account balance

    void withdraw(int dollars, int cents);
    //Subtracts $dollars.cents from the account balance
private:
};

int main( )
{
    SavingsAccount account(100, 50, 5.5);
    account.output(cout);
    cout << endl;
    cout << "Depositing $10.25." << endl;
    account.deposit(10,25);
    account.output(cout);
    cout << endl;
    cout << "Withdrawing $11.80." << endl;
    account.withdraw(11,80);
    account.output(cout);
    cout << endl;
    return 0;
}

SavingsAccount::SavingsAccount(int dollars, int cents, double rate):
    BankAccount(dollars, cents, rate)
{
    //deliberately empty

}

void SavingsAccount::deposit(int dollars, int cents)
{

    double balance = get_balance();
    balance += dollars;
    balance += (static_cast<double>(cents) / 100);
    int new_dollars = static_cast<int>(balance);
    int new_cents = static_cast<int>((balance - new_dollars) * 100);
```

The colon indicates that the class `SavingsAccount` *is derived from the class* `BankAccount`

Only new member functions or variables need to be defined

The `SavingsAccount` *constructor invokes the* `BankAccount` *constructor. Note the preceding colon.*

The `deposit` *function adds the new amount to the balance and changes the member variables via the* `set` *function*

(continued)

DISPLAY 10.9 A `SavingsAccount` Derived Class *(part 2 of 2)*

```
        set(new_dollars, new_cents, get_rate());
    }

    void SavingsAccount::withdraw(int dollars, int cents)
    {
        double balance = get_balance();
        balance -= dollars;
        balance -= (static_cast<double>(cents) / 100);
        int new_dollars = static_cast<int>(balance);
        int new_cents = static_cast<int>((balance - new_dollars) * 100);
        set(new_dollars, new_cents, get_rate());
    }
```

The `withdraw` *function subtracts the amount from the balance and changes the member variables via the* `set` *function*

Screen Output

```
Account balance $100.50
Interest rate 5.50%
Depositing $10.25.
Account balance $110.75
Interest rate 5.50%
Withdrawing $11.80.
Account balance $98.95
Interest rate 5.50%
```

```
//Create a new CD with $1000, 6% interest, 180 days to maturity
CDAccount newCD(1000, 0, 6.0, 180);

newCD.deposit(100,50);
days_to_maturity = newCD.get_days_to_maturity( );
//Returns 180
balance = newCD.get_balance( );
//Returns 1100.50
```

Invoking a function in `SavingsAccount`

Invoking a function in `CDAccount`

Invoking a function in `BankAccount`

This short example has only scratched the surface of what is possible using inheritance. Additional details are described in Chapter 15. While it does take some effort to learn how to effectively design classes using inheritance, the effort will pay off in the long run. You will end up writing less code that is easier to understand and maintain than code that does not use inheritance.

SELF-TEST EXERCISES

29. How does inheritance support code reuse and make code easier to maintain?

30. Can a derived class directly access by name a private member variable of the parent class?

31. Suppose the class `SportsCar` is a derived class of a class `Automobile`. Suppose also that the class `Automobile` has public member functions named `accelerate` and `addGas`. Will an object of the class `SportsCar` have member functions named `accelerate` and `addGas`?

CHAPTER SUMMARY

- A structure can be used to combine data of different types into a single (compound) data value.
- A class can be used to combine data and functions into a single (compound) object.
- A member variable or a member function for a class may be either public or private. If it is public, it can be used outside of the class. If it is private, it can be used only in the definition of another member function in the class.
- A function may have formal parameters of a class or structure type. A function may return values of a class or structure type.
- A member function for a class can be overloaded in the same way as ordinary functions are overloaded.
- A **constructor** is a member function of a class that is called automatically when an object of the class is declared. A constructor must have the same name as the class of which it is a member.
- A data type consists of a collection of values together with a set of basic operations defined on these values.
- A data type is called an **abstract data type** (abbreviated **ADT**) if a programmer who uses the type does not need to know any of the details about how the values and operations for that type are implemented.
- One way to implement an abstract data type in C++ is to define a class with all member variables being private and with the operations implemented as public member functions.
- Inheritance refers to a parent/child relationship between classes. The child or derived class inherits members from the parent class.

Answers to Self-Test Exercises

1. a. *double*

   b. *double*

   c. illegal—cannot use `struct` tag instead of a structure variable

   d. illegal—`savings_account` undeclared

   e. *char*

   f. *TermAccount*

2.
```
A $9.99
A $1.11
```

3. Many compilers give poor error messages. Surprisingly, the error message from g++ is quite informative.

```
g++ -fsyntax-only c10testq3.cc
c10testq3.cc:8: semicolon missing after declaration of
'Stuff'
c10testq3.cc:8: extraneous 'int' ignored
c10testq3.cc:8: semicolon missing after declaration of
'struct Stuff'
```

4. `A x = {1,2};`

5. a. Too few initializers, not a syntax error. After initialization, `due_date.month == 12`, `due_date.day == 21`, `due_date.year == 0`. Member variables not provided an initializer are initialized to a zero of appropriate type.

   b. Correct after initialization: `12 == due_date.month`, `21 == due_date.day`, `2022 == due_date.year`.

   c. Error: too many initializers.

   d. May be a design error, that is, an error in intent. The author of the code provides only two digits for the date initializer. There should be four digits used for the year because a program using two-digit dates could fail in ways that vary from amusing to disastrous at the turn of the century.

6.
```
struct EmployeeRecord
{
    double wage_rate;
    int vacation;
    char status;
};
```

7.
```
void read_shoe_record(ShoeType& new_shoe)
{
    cout << "Enter shoe style (one letter): ";
    cin >> new_shoe.style;
    cout << "Enter shoe price $";
    cin >> new_shoe.price;
}
```

8. `ShoeType discount(ShoeType old_record)`

```
{
    ShoeType temp;
    temp.style = old_record.style;
    temp.price = 0.90 * old_record.price;
    return temp;
}
```

9. `struct StockRecord`

```
{
    ShoeType shoe_info;
    Date arrival_date;
};
```

10. `StockRecord aRecord;`

```
aRecord.arrival_date.year = 2006;
```

11. `void DayOfYear::input()`

```
{
    cout << "Enter month as a number: ";
    cin >> month;
    cout << "Enter the day of the month: ";
    cin >> day;
}
```

12. `void Temperature::set(double new_degrees, char new_scale)`

```
{
    degrees = new_degrees;
    scale = new_scale;
}
```

13. Both the dot operator and the scope resolution operator are used with member names to specify the `class` or `struct` of which the member name is a member. If `class DayOfYear` is as defined in Display 10.3 and `today` is an object of the class `DayOfYear`, then the member `month` may be accessed with the dot operator: `today.month`. When we give the definition of a member function, the scope resolution operator is used to tell the compiler that this function is the one declared in the class whose name is given before the scope resolution operator.

14.
```
void DayOfYear::check_date( )
{
    if ((month < 1) || (month > 12)
        || (day < 1) || (day > 31))
    {
        cout << "Illegal date. Aborting program.\n";
        exit(1);
    }
    if (((month == 4) || (month == 6) || (month == 9)
                                      || (month == 11))
                                      && (day == 31))
    {
        cout << "Illegal date. Aborting program.\n";
        exit(1);
    }
    if ((month == 2) && (day > 29))
    {
        cout << "Illegal date. Aborting program.\n";
        exit(1);
    }
}
```

15.
```
hyundai.price = 4999.99; //ILLEGAL. price is private.

jaguar.set_price(30000.97); //LEGAL
double a_price, a_profit; //LEGAL
a_price = jaguar.get_price(); //LEGAL
a_profit = jaguar.get_profit(); //ILLEGAL. get_profit is private.
a_profit = hyundai.get_profit(); //ILLEGAL. get_profit is private.
if (hyundai == jaguar) //ILLEGAL. Cannot use == with classes.
    cout << "Want to swap cars?";
hyundai = jaguar; //LEGAL
```

16. After the change, they would all be legal except for the following, which is still illegal:

```
if (hyundai == jaguar) //ILLEGAL. Cannot use == with classes.
    cout << "Want to swap cars?";
```

17. *private* restricts access to function definitions to member functions of the same class. This restricts any change of *private* variables to functions provided by the class author. The class author is then in control of these changes to the *private* data, preventing inadvertent corruption of the class data.

18. a. Only one. The compiler warns if you have no *public*: members in a class (or *struct* for that matter).

    b. None; we normally expect to find at least one *private*: section in a class.

c. In a class, such a section is by default a `private`: section.

d. In a *`struct`*, such a section is by default a *`public`*: section.

19. A possible correct answer is as follows:

```
double difference(BankAccount account1, BankAccount account2)
{
    return (account1.get_balance() - account2.get_balance());
}
```

Note that the following is not correct, because `balance` is a private member.

```
double difference(BankAccount account1, BankAccount account2)
{
    return (account1.balance - account2.balance); //ILLEGAL
}
```

20.
```
void double_update(BankAccount& the_account)
{
    the_account.update();
    the_account.update();
}
```

Note that since this is not a member function, you must give the object name and dot operator when you call `update`.

21.
```
BankAccount new_account(BankAccount old_account)
{
    BankAccount temp;
    temp.set(0, old_account.get_rate( ));
    return temp;
}
```

22.
```
YourClass an_object(42, 'A'); //LEGAL
YourClass another_object; //LEGAL
YourClass yet_another_object(); //PROBLEM
an_object = YourClass(99, 'B'); //LEGAL
an_object = YourClass(); //LEGAL
an_object = YourClass; //ILLEGAL
```

The statement marked *//PROBLEM* is not, strictly speaking, illegal, but it does not mean what you might think it means. If you mean this to be a declaration of an object called `yet_another_object`, then it is wrong. It is a correct function declaration for a function called `yet_another_object` that takes zero arguments and that returns a value of type `YourClass`, but that is not the intended meaning. As a practical matter, you can probably consider it illegal. The correct way to declare an object called yet_another_object so that it will be initialized with the default constructor is as follows:

```
YourClass yet_another_object;
```

23. The modified class definition is as follows:

```
class DayOfYear
{
public:
    DayOfYear(int the_month, int the_day);
    //Precondition: the_month and the_day form a
    //possible date. Initializes the date according to
    //the arguments.
    DayOfYear();
    //Initializes the date to January first.
    void input();
    void output();
    int get_month();
    //Returns the month, 1 for January, 2 for February, etc.
    int get_day();
    //Returns the day of the month.
private:
    void check_date( );
    int month;
    int day;
};
```

Notice that we have omitted the member function set, since the constructors make set unnecessary. You must also add the following function definitions (and delete the function definition for DayOfYear::set):

```
DayOfYear::DayOfYear(int the_month, int the_day)
{
    month = the_month;
    day = the_day;
    check_date();
}
DayOfYear::DayOfYear()
{
    month = 1;
    day = 1;
}
```

24. The class definition is the same as in the previous exercise. The constructor definitions would change to the following:

```
DayOfYear::DayOfYear(int the_month, int the_day)
    : month(the_month), day(the_day)
```

```
{
    check_date();
}
DayOfYear::DayOfYear() : month(1), day(1)
{
    //Body intentionally empty.
}
```

25. The member variables should all be private. The member functions that are part of the interface for the ADT (that is, the member functions that are operations for the ADT) should be public. You may also have auxiliary helping functions that are used only in the definitions of other member functions. These auxiliary functions should be private.

26. All the declarations of private member variables are part of the implementation. (There should be no public member variables.) All the function declarations for public member functions of the class (which are listed in the class definitions) as well as the explanatory comments for these function declarations are part of the interface. All the function declarations for private member functions are part of the implementation. All member function definitions (whether the function is public or private) are part of the implementation.

27. You need to read only the interface parts. That is, you need to read only the function declarations for public members of the class (which are listed in the class definitions) as well as the explanatory comments for these function declarations. You need not read any of the function declarations of the private member functions, the declarations of the private member variables, the definitions of the public member functions, or the definitions of the private member functions.

28.
```
BankAccount::BankAccount(int dollars, int cents,
                         double rate) : dollars_part(dollars),
                         cents_part(cents), interest_
                         rate(fraction(rate))
{
    if ((dollars < 0) || (cents < 0) || (rate < 0))
    {
        cout << "Illegal values for money or interest rate.\n";
        exit(1);
    }
}

BankAccount::BankAccount(int dollars, double rate)
    : dollars_part(dollars), cents_part(0),
                   interest_rate(fraction(rate))
{
    if ((dollars < 0) || (rate < 0))
    {
        cout << "Illegal values for money or interest rate.\n";
        exit(1);
    }
```

29. Functions and data defined for the parent class can be made available in the derived class, eliminating the need to redefine the functions and data again in the derived class. This enhances maintainability because there is now no duplication of code among multiple classes and hence only a single location in the code that may be subject to change. Additionally, inheritance provides a clean way to isolate code that is only applicable to a derived class. Since such code only appears in the definition of the derived class, it is usually easier to read.

30. No, but a derived class can indirectly access a private member variable of the parent class through a public function.

31. Yes, the derived class will have access to the same functions. In Chapter 15 we will discuss how we can make the functions do different things for an object of class `SportsCar` versus an object of class `Automobile`.

PRACTICE PROGRAMS

Practice Programs can generally be solved with a short program that directly applies the programming principles presented in this chapter.

VideoNote
Solution to Practice Program 10.1

1. Redefine `CDAccount` from Display 10.1 so that it is a class rather than a structure. Use the same member variables as in Display 10.1 but make them private. Include member functions for each of the following: one to return the initial balance, one to return the balance at maturity, one to return the interest rate, and one to return the term. Include a constructor that sets all of the member variables to any specified values, as well as a default constructor. Embed your class definition in a test program.

2. Redo your definition of the class `CDAccount` from Practice Program 1 so that it has the same interface but a different implementation. The new implementation is in many ways similar to the second implementation for the class `BankAccount` given in Display 10.7. Your new implementation for the class `CDAccount` will record the balance as two values of type *`int`*: one for the dollars and one for the cents. The member variable for the interest rate will store the interest rate as a fraction rather than as a percentage. For example, an interest rate of 4.3% will be stored as the value `0.043` of type *`double`*. Store the term in the same way as in Display 10.1.

3. Define a class for a type called `CounterType`. An object of this type is used to count things, so it records a count that is a nonnegative whole number. Include a default constructor that sets the counter to zero and a constructor

with one argument that sets the counter to the value specified by its argument. Include member functions to increase the count by 1 and to decrease the count by 1. Be sure that no member function allows the value of the counter to become negative. Also, include a member function that returns the current count value and one that outputs the count to a stream. The member function for doing output will have one formal parameter of type `ostream` for the output stream that receives the output. Embed your class definition in a test program.

PROGRAMMING PROJECTS

Programming Projects require more problem-solving than Practice Programs and can usually be solved many different ways.Visit www.myprogramminglab.com to complete many of these Programming Projects online and get instant feedback.

1. Write a grading program for a class with the following grading policies:

   a. There are two quizzes, each graded on the basis of 10 points.

   b. There is one midterm exam and one final exam, each graded on the basis of 100 points.

   c. The final exam counts for 50 percent of the grade, the midterm counts for 25 percent, and the two quizzes together count for a total of 25 percent. (Do not forget to normalize the quiz scores. They should be converted to a percent before they are averaged in.)

   Any grade of 90 or more is an A, any grade of 80 or more (but less than 90) is a B, any grade of 70 or more (but less than 80) is a C, any grade of 60 or more (but less than 70) is a D, and any grade below 60 is an F.

   The program will read in the student's scores and output the student's record, which consists of two quiz and two exam scores as well as the student's average numeric score for the entire course and the final letter grade. Define and use a structure for the student record. If this is a class assignment, ask your instructor if input/output should be done with the keyboard and screen or if it should be done with files. If it is to be done with files, ask your instructor for instructions on file names.

2. Redo Programming Project 1 (or do it for the first time), but this time make the student record type a class type rather than a structure type. The student record class should have member variables for all the input data described in Programing Project 1 and a member variable for the student's weighted average numeric score for the entire course as well as a member variable for the student's final letter grade. Make all member variables private. Include member functions for each of the following: member functions to set each of the member variables to values given as an argument(s) to the function, member functions to retrieve the data from each of the

member variables, a *void* function that calculates the student's weighted average numeric score for the entire course and sets the corresponding member variable, and a *void* function that calculates the student's final letter grade and sets the corresponding member variable.

3. Define a class called `Month` that is an abstract data type for a month. Your class will have one member variable of type *int* to represent a month (1 for January, 2 for February, and so forth). Include all the following member functions: a constructor to set the month using the first three letters in the name of the month as three arguments, a constructor to set the month using an integer as an argument (1 for January, 2 for February, and so forth), a default constructor, an input function that reads the month as an integer, an input function that reads the month as the first three letters in the name of the month, an output function that outputs the month as an integer, an output function that outputs the month as the first three letters in the name of the month, and a member function that returns the next month as a value of type `Month`. The input and output functions will each have one formal parameter for the stream. Embed your class definition in a test program.

4. Redefine the implementation of the class `Month` described in Programming Project 3 (or do the definition for the first time, but do the implementation as described here). This time the month is implemented as three member variables of type *char* that store the first three letters of the name of the month. Embed your definition in a test program.

5. (In order to do this project you must have first done either Programming Project 3 or Project 4.) Rewrite the program in Display 10.4, but use the class `Month` that you defined in Project 3 or Project 4 as the type for the member variable to record the month. (You may define the class `Month` either as described in Project 3 or as described in Project 4.) Redefine the member function `output` so that it has one formal parameter of type `ostream` for the output stream. Modify the program so that everything that is output to the screen is *also* output to a file. This means that all output statements will occur twice: once with the argument `cout` and once with an output-stream argument. If you are in a class, obtain the file name from your instructor. The input will still come from the keyboard. Only the output will be sent to a file.

6. My mother always took a little red counter to the grocery store. The counter was used to keep a tally of the amount of money she would have spent so far on that visit to the store, if she bought all the items in her basket. There was a four-digit display, increment buttons for each digit, and a reset button. There was an overflow indicator that came up red if more money was entered than the `$99.99` it would register. (This was a long time ago.)

   Write and implement the member functions of a class `Counter` that simulates and slightly generalizes the behavior of this grocery store counter. The constructor should create a `Counter` object that can count up to the constructor's argument. That is, `Counter(9999)` should provide a counter that can count up to `9999`. A newly constructed counter displays a reading of `0`. The member function `void reset();` sets the counter's number to `0`. The member functions `void incr1();` increments the units digit by 1, `void incr10();` increments the tens digit by 1, and `void incr100();` and `void incr1000( );` increment the next two digits, respectively. Accounting for any carry when you increment should require no further action than adding an appropriate number to the private data member. A member function `bool overflow();` detects overflow. (Overflow is the result of incrementing the counter's private data member beyond the maximum entered at counter construction.)

Use this class to provide a simulation of my mother's little red clicker. Even though the display is an integer, in the simulation, the rightmost (lower-order) two digits are always thought of as cents and tens of cents, the next digit is dollars, and the fourth digit is tens of dollars.

Provide keys for cents, dimes, dollars, and tens of dollars. Unfortunately, no choice of keys seems particularly mnemonic. One choice is to use the keys `asdfo`: `a` for cents, followed by a digit `1` to `9`; `s` for dimes, followed by digits `1` to `9`; `d` for dollars, followed by a digit `1` to `9`; and `f` for tens of dollars, again followed by a digit `1` to `9`. Each entry (one of `asdf` followed by `1` to `9`) is followed by pressing the Return key. Any overflow is reported after each operation. Overflow can be requested by pressing the `o` key.

7. Write a `rational number` class. This problem will be revisited in Chapter 11, where operator overloading will make the problem much easier. For now we will use member functions `add`, `sub`, `mul`, `div`, and `less` that each carry out the operations `+`, `-`, `*`, `/`, and `<`. For example, `a + b` will be written `a.add(b)`, and `a < b` will be written `a.less(b)`.

   Define a class for rational numbers. A rational number is a "ratio-nal" number, composed of two integers with division indicated. The division is not carried out, it is only indicated, as in `1/2`, `2/3`, `15/32`, `65/4`, `16/5`. You should represent rational numbers by two `int` values, `numerator` and `denominator`.

   A principle of abstract data type construction is that constructors must be present to create objects with any legal values. You should provide constructors to make objects out of pairs of `int` values; this is a constructor with two `int` parameters. Since every `int` is also a rational number, as in `2/1` or `17/1`, you should provide a constructor with a single `int` parameter.

   Provide member functions `input` and `output` that take an `istream` and `ostream` argument, respectively, and fetch or write rational numbers in the form `2/3` or `37/51` to or from the keyboard (and to or from a file).

Provide member functions `add`, `sub`, `mul`, and `div` that return a rational value. Provide a function `less` that returns a *`bool`* value. These functions should do the operation suggested by the name. Provide a member function `neg` that has no parameters and returns the negative of the calling object.

Provide a `main` function that thoroughly tests your class implementation. The following formulas will be useful in defining functions.

```
a/b + c/d = (a * d + b * c) / (b * d)
a/b - c/d = (a * d - b * c) / (b * d)
(a/b) * (c/d) = (a * c) / (b * d)
(a/b) / (c/d) = (a * d) / (c * b)
-(a/b) = (-a/b)
(a/b) < (c/d) means (a * d) < (c * b)
(a/b) == (c/d) means (a * d) == (c * b)
```

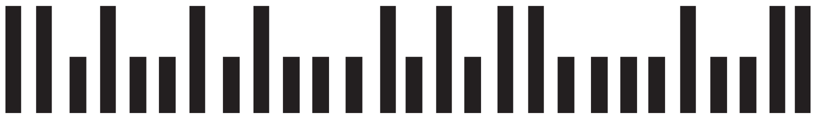
Let any sign be carried by the numerator; keep the denominator positive.

8. Define a class called `Odometer` that will be used to track fuel and mileage for an automotive vehicle. Include private member variables to track the miles driven and the fuel efficiency of the vehicle in miles per gallon. The class should have a constructor that initializes these values to zero. Include a member function to reset the odometer to zero miles, a member function to set the fuel efficiency, a member function that accepts miles driven for a trip and adds it to the odometer's total, and a member function that returns the number of gallons of gasoline that the vehicle has consumed since the odometer was last reset.

   Use your class with a test program that creates several trips with different fuel efficiencies.

9. Redo Programming Project 7 from Chapter 5 (or do it for the first time), but this time use a class to encapsulate the date. Use private member variables to store the day, month, and year along with an appropriate constructor and member functions to get and set the data. Create a public function that returns the day of the week. All helper functions should be declared private. Embed your class definition in a suitable test program.

10. The U.S. Postal Service printed a bar code on every envelope that represented a five- (or more) digit zip code using a format called POSTNET (this format was deprecated in favor of a new system, OneCode, in 2009). The bar code consists of long and short bars as shown:



For this program, we will represent the bar code as a string of digits. The digit 1 represents a long bar, and the digit 0 represents a short bar. Therefore, the bar code would be represented in our program as

```
110100101000101011000010011
```

The first and last digits of the bar code are always 1. Removing these leaves 25 digits. If these 25 digits are split into groups of 5 digits each, we have

```
10100     10100     01010     11000     01001
```

Next, consider each group of 5 digits. There will always be exactly two 1s in each group of digits. Each digit stands for a number. From left to right, the digits encode the values 7, 4, 2, 1, and 0. Multiply the corresponding value with the digit and compute the sum to get the final encoded digit for the zip code. The table below shows the encoding for 10100.

Bar Code Digits	1	0	1	0	0
Value	7	4	2	1	0
Product of Digit * Value	7	0	2	0	0

```
Zip Code Digit = 7 + 0 + 2 + 0 + 0 = 9
```

Repeat this for each group of 5 digits and concatenate to get the complete zip code. There is one special value. If the sum of a group of 5 digits is 11, then this represents the digit 0 (this is necessary because with two digits per group it is not possible to represent zero). The zip code for the sample bar code decodes to 99504. Although the POSTNET scheme may seem unnecessarily complex, its design allows machines to detect if errors have been made in scanning the zip code.

Write a zip code class that encodes and decodes 5-digit bar codes used by the U.S. Postal Service on envelopes. The class should have two constructors. The first constructor should input the zip code as an integer, and the second constructor should input the zip code as a bar code string consisting of 0s and 1s, as described above. Although you have two ways to input the zip code, internally, the class should store the zip code using only one format (you may choose to store it as a bar code string or as a zip code number). The class should also have at least two public member functions, one to return the zip code as an integer, and the other to return the zip code in bar code format as a string. All helper functions should be declared private. Embed your class definition in a suitable test program. Your program should print an error message if an invalid bar code is passed to the constructor.

11. Consider a class `Movie` that contains information about a movie. The class has the following attributes:

    - The movie name
    - The MPAA rating (for example, G, PG, PG-13, R)
    - The number of people that have rated this movie as a 1 (Terrible)
    - The number of people that have rated this movie as a 2 (Bad)
    - The number of people that have rated this movie as a 3 (OK)
    - The number of people that have rated this movie as a 4 (Good)
    - The number of people that have rated this movie as a 5 (Great)

    Implement the class with accessor and mutator functions for the movie name and MPAA rating. Write a function `addRating` that takes an integer as an input parameter. The function should verify that the parameter is a number between 1 and 5, and if so, increment the number of people rating the movie that match the input parameter. For example, if 3 is the input parameter, then the number of people that rated the movie as a 3 should be incremented by 1. Write another function, `getAverage`, that returns the average value for all of the movie ratings. Finally, add a constructor that allows the programmer to create the object with a specified name and MPAA rating. The number of people rating the movie should be set to 0 in the constructor.

    Test the class by writing a `main` function that creates at least two movie objects, adds at least five ratings for each movie, and outputs the movie name, MPAA rating, and average rating for each movie object.