Chapter 7

INTRODUCTION TO CLASSES

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

Let us briefly explain some key concepts and terminology of object orientation. Object-oriented programming (OOP) *encapsulates* data (attributes) and functions (behavior) into packages called *classes*; the data and functions of a class are intimately tied together. A class is like a blueprint. Out of a blueprint, a builder can build a house. Out of a class, a programmer can create an object. One blueprint can be reused many times to make many objects of the same class.

Classes have the property of *information hiding*. This means that although class objects may know how to communicate with one another across well-defined *interfaces*, classes normally are not allowed to know how other classes are implemented – implementation details are hidden within the classes themselves.

7.1 CLASSES

Classes

In C++ programming, classes are structures that contain variables along with functions for manipulating that data.

The functions and variables defined in a class are referred to as class members.

Class variables are referred to as **data members**, while class functions are referred to as **member functions**.

Classes are referred to as user-defined data types or programmer-defined data types because you can work with a class as a single unit, or objects, in the same way you work with variables.

When you declare an object from a class, you are said to be *instantiating* an object.

The most important feature of C++ programming is class definition with the **class** keyword. You define classes the same way you define structures, and you access a class's data members using the **member selection operator**.

Example:

```
class Time {
public:
  Time();
  void setTime( int, int, int );
  void printMilitary();
  void printStandard();
private:
  int hour;
  int minute;
  int second;
};
```

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

```
Time sunset,          // object of type Times
     ArOfTimes[5],    // array of Times objects
     *ptrTime;        // pointer to a Times objects
```

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

7.2 INFORMATION HIDING

The principle of *information hiding* states that any class members that other programmers, or clients, do not need to access or know about should be hidden.

Many programmers prefer to make all of their data member private in order to prevent clients from accidentally assigning the wrong value to a variable or from viewing the internal workings of their programs.

7.3.1 Access Specifiers

Access specifiers control a client's access to data members and member functions. There are four levels of access specifiers: public, private, protected, and friend.

The **public access specifier** allows anyone to call a class's function member or to modify a data member.

The **private access specifier** is one of the key elements in information hiding since it prevents clients from calling member functions or accessing data members.

Both public and private specifiers have what is called **class scope**: class members of both access types are accessible from any of a class's member functions.

Example:

```
class Time {
public:
   Time();
   void setTime( int, int, int );
   void printMilitary();
   void printStandard();
private:
   int hour;
   int minute;
   int second;
};
```

Note that the data members *hour*, *minute*, and *second* are preceded by the private member access specifier. A class' *private* data members are normally not accessible outside the class. The philosophy here is that the actual data representation used within the class is of no concern to the class' clients. In this sense, the implementation of a class is said to be hidden from its clients. Such information hiding promotes program modifiability and simplifies the client's perception of a class.

You can depict the classes graphically in a class diagram as in Figure 7.1.

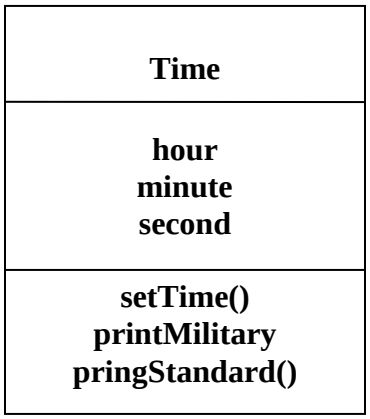

Figure 7.1 The class Time

The diagram shown above follows the format of the Unified Modeling Language (UML). Each class is represented by a box, with the class name in the top portion of the box, any data members that you care to describe in the middle portion of the box, and the member functions (the functions that belong to this object, which receive any messages you send to that object) in the bottom portion of the box.

7.3.2 Interface and Implementation Files

Although the first step in information hiding is to assign private access specifiers to class members, private access specifiers only designate which class members a client is not allowed to call or change. Private access specifiers do not prevent clients from seeing class code. To prevent clients from seeing the details of how your code is written, you place your class's interface code and implementation code in separate files.

The separation of classes into separate *interface* and *implementation files* is considered to be a fundamental software development technique since it allows you to hide the details of how your classes are written and makes it easier to modify programs.

The *interface* code refers to the data member and function member declarations inside a class's braces. Interface code does not usually contain definitions for function members, nor does it usually assigns values to the data members. You create interface code in a header file with an *.h* extension.

The *implementation* code refers to a class's function definitions and any code that assigns values to a class's data members. In other words, implementation code contains the actual member functions themselves and assigns values to data members. You add implementation code to standard C++ source files with an extension of *.cpp*.

As far as clients of a class are concerned, changes in the class' implementation do not affect the client as long as the class' interface originally provided to the client is unchanged. All that a client needs to know to use the class correctly should be provided by the interface.

Preventing Multiple Inclusion

Large class-based programs are sometimes composed of multiple interface and implementation files. With large program, you need to ensure that you do not include multiple instances of the same header file when you compile the program, since multiple inclusion will make your program unnecessary large.

C++ generates an error if you attempt to compile a program that includes multiple instances of the same header file. To prevent this kind of error, most C++ programmers use the **#define** preprocessor directive with the **#if** and **#endif** preprocessor directives in header files.

The **#if** and **#endif** preprocessor directives determine which portions of a file to compile depending on the result of a conditional expression.

The syntax for the **#if** and **#endif** preprocessor directives:

```
#if conditional expression
   statements to compile;
#endif
```

Example:

```
#if !defined(TIME1_H)
#define TIME1_H
class Time {
public:
   Time();
   void setTime( int, int, int );
   void printMilitary();
   void printStandard();
private:
   int hour;
   int minute;
   int second;
};
#endif
```

Note: Common practice when defining a header file's constant is to use the header file's name in uppercase letters appended with H. For example, the constant for the *time1.h* header file is usually defined as TIME1_H.

7.3 MEMBER FUNCTIONS

In this section, we learn how to write member functions for a class.

7.3.1 Inline functions

Although member functions are usually defined in an implementation file, they can also be defined in an interface file. Functions defined inside the class body in an interface file are called **inline functions**.

Example:

```
class Stocks {
public:
    double getTotalValue(int iShares, double dCurPrice){
          double dCurrentValue;
          iNumShares = iShares;
          dCurrentPricePerShare = dCurPrice;
          dCurrentValue = iNumShares*dCurrentPricePerShare;
          return dCurrentValue;
     }
private:
     int iNumShares;
    double dPurchasePricePerShare;
    double dCurrentPricePerShare;
```

};

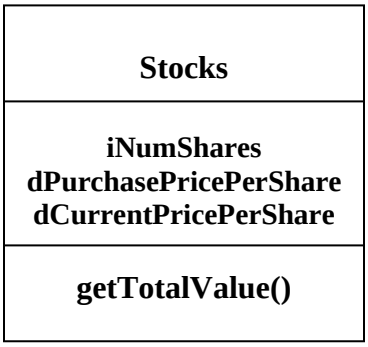

Figure 6.2 The class Stocks

7.3.2 Member functions in Implementation File

Member function definitions are always placed in the implementation file.

In the Example 7.3.1, for the class *Stocks*, the definition of the member function *getTotalValue* is placed in the source-code file *stocks.cpp* in which the main program is also included.

Example 7.3.1

```
//stocks.h     ---------- interface section
#if !defined(STOCKS_H)
#define STOCKS_H
class Stocks{
public:
      double getTotalValue(int iShares, double dCurPrice);
private:
    int iNumShares;
   double dPurchasePricePerShare;
   double dCurrentPricePerShare;
};
#endif
// stocks.cpp ---------- implementation section
#include “stocks.h”
#include<iostream.h>

double Stocks::getTotalValue(int iShares, double dCurPrice){
          double dCurrentValue;
          iNumShares = iShares;
          dCurrentPricePerShare = dCurPrice;
```

```
        dCurrentValue = iNumShares*dCurrentPricePerShare;
        return dCurrentValue;
}
int main(){
        Stocks stockPick;
        cout << stockPick.getTotalValue(200, 64.25) << endl;
        return 0;
}
```

Output of the above program:

12850

Note: The format of member functions included in the implementation section is as follows:

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

In order for your class to identify which functions in an implementation section belong to it, you precede the function name in the function definition header with the class name and the *scope resolution operator* (::).

7.3.3 Access Functions

Access to a class' private data should be carefully controlled by the use of member functions, called *access functions*. For example, to allow clients to read the value of private data, the class can provide a *get* function.

To enable clients to modify private data, the class can provide a *set* function. Such modification would seem to violate the notion of private data. But a *set* member function can provide data validation capabilities (such as range checking) to ensure that the value is set properly. A *set* function can also translate between the form of data used in the interface and the form used in the implementation.

A *get* function need not expose the data in "raw" format; rather, the *get* function can edit data and limit the view of the data the client will see.

Example 7.3.2

```
// time1.h
#if !defined(TIME1_H)
#define TIME1_H
class Time {
```

```
public:
   Time();                      // constructor
   void setTime( int, int, int ); // set hour, minute, second
   void printMilitary();          // print military time format
   void printStandard();          // print standard time format
private:
   int hour;
   int minute;
   int second;
};
#endif;
// time1.cpp
#include “time1.h”
#include <iostream.h>

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

void Time::setTime( int h, int m, int s )
{
   hour = ( h >= 0 && h < 24 ) ? h : 0;
   minute = ( m >= 0 && m < 60 ) ? m : 0;
   second = ( s >= 0 && s < 60 ) ? s : 0;
}

void Time::printMilitary()
{
   cout << ( hour < 10 ? "0" : "" ) << hour << ":"
        << ( minute < 10 ? "0" : "" ) << minute;
}

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

// Driver to test simple class Time
int main()
{
   Time t;  // instantiate object t of class Time
```

```
   cout << "The initial military time is ";
   t.printMilitary();
   cout << "\nThe initial standard time is ";
   t.printStandard();

   t.setTime( 13, 27, 6 );
   cout << "\n\nMilitary time after setTime is ";
   t.printMilitary();
   cout << "\nStandard time after setTime is ";
   t.printStandard();

   t.setTime( 99, 99, 99 );  // attempt invalid settings
   cout << "\n\nAfter attempting invalid settings:"
        << "\nMilitary time: ";
   t.printMilitary();
   cout << "\nStandard time: ";
   t.printStandard();
   cout << endl;
   return 0;
}
```

The output of the above program:

```
The initial military time is 00:00
The initial standard time is 12:00:00 AM

Military time after setTime is 13:37
Standard time after setTime is 1:27:06 PM

After attempting invalid settings:
Military time: 00:00
Standard time: 12:00:00 AM
```

In the above example, for the class *Time* we can see that member functions *printStandard* and *printMilitary* are two *get* function and member function *setTime* is a *set* function.

7.3.4 Constructor Functions

A **constructor function** is a special function with the same name as its class. This function is called automatically when an object from a class is instantiated.

You define and declare constructor functions the same way you define other functions, although you do not include a return type since constructor functions do not return values.

Example:

```
class Payroll{
public:
    Payroll( ){    // constructor function
      dFedTax = 0.28;
       dStateTax = 0.05;
   };
private:
    double dFedTax;
    double dStateTax;
}
```

You also include just a function prototype in the interface file for the constructor function and then create the function definition in the implementation file.

```
Payroll::Payroll( ){    // constructor function
      dFedTax = 0.28;
       dStateTax = 0.05;
   };
```

Example 7.3.3

```
#include <iostream.h>
#include <iomanip.h>

// class declaration section
class Date
{
  private:
    int month;
    int day;
    int year;
  public:
    Date(int = 7, int = 4, int = 2001); // constructor with default values
};

// implementation section
Date::Date(int mm, int dd, int yyyy)   // constructor
{
  month = mm;
  day = dd;
  year = yyyy;
  cout << "Created a new data object with data values "
       << month << ", " << day << ", " << year << endl;
}
```

```
int main()
{
  Date a;           // declare an object
  Date b;           // declare an object
  Date c(4,1,2002);  // declare an object
  return 0;
}
```

The output of the above program:

```
Created a new data object with data values 7, 4, 2001
Created a new data object with data values 7, 4, 2001
Created a new data object with data values 4,1, 2002
```

Default constructor refers to any constructor that does not require any parameters when it is called.

In Example 7.3.3, the prototype *Date(int = 7, int = 4, int = 2001)* is valid for a default constructor. Here, each argument has been given a default value. Then an object can be declared as type *Date* without supplying any further arguments.

Although any legitimate C++ statement can be used within a constructor function, such as the *cout* statement used in Example 7.3.3., it is best to keep constructors simple and use them only for initialization purposes.

7.4 DYNAMIC MEMORY ALLOCATION WITH OPERATORS new AND delete

The *new* and *delete* operators provides a nice means of performing dynamic memory allocation (for any built-in or user-defined type). Consider the following code

```
TypeName *typeNamPtr;
typeNamePtr = new TypeName;
```

The *new* operator automatically creates an object of the proper size, calls the constructor for the object and returns a pointer of the correct type.

To destroy the object and free the space for this object in C++ you must use the *delete* operator as follows:

```
delete typeNamePtr;
```

For built-in data types, we also can use the *new* and *delete* operators.

Example 1:

```
int *pPointer;
pPointer = new int;
```

Example 2:

```
delete pPointer;
```

Example 3: A 10-element integer array can be created and assigned to *arrayPtr* as follows:

```
int *arrayPtr = new int[10];
```

This array is deleted with the statement

```
delete [] arrayPtr;
```

Stack versus heap

A *stack* is a region of memory where applications can store data such as local variables, function calls, and parameter information.

The programmers have no control over the stack. C++ automatically handles placing and removing data to and from stack.

The *heap* or free store, is an area of memory that is available to application for storing data whose existence and size are not known until run-time.

Notice that when we use *new* operator, we can allocate a piece of memory on the heap and when we use *delete* operator, we can deallocate (free) a piece of memory on the heap. In other words, we can manage the memory allocation on the heap explicitly through *new* and *delete* operators.

The syntax for using the *new* operator is

```
pointer = new data_type;
```

For example, to declare an int pointer *iPointer* that points to a heap variable, you use the following statements:

```
int* iPointer;
iPointer = new int;
```

The syntax for using the *delete* operator is

```
delete pointer_name;
```

For example, to delete the heap memory pointed to by the *iPointer* pointer, you use the statement *delele iPointer;*.

Deleting the contents of an array stored on the heap also requires a slightly different syntax. You must append two brackets to the *delete* keyword using the syntax *delete[] array_name;* .

Notice that the **delete** operator does not delete the pointer itself. Rather, it deletes the contents of the heap memory address pointed to by a pointer variable. You can reuse the pointer itself after calling the **delete** operator. The pointer still exists and points to the same heap memory address that it did before calling the **delete** operator.

Example 7.4.1

```
#include<iostream.h>
int main( )
{
   double* pPrimeInterest = new double;
   *pPrimeInterest = 0.065;
   cout << "The value of pPrimeInterest is: "
        << *pPrimeInterest << endl;
   cout << "The memory address of pPimeInterest is:"
        << &pPrimeInterest << endl;

   delete pPrimeInterest;
   *pPimeInterest = 0.070;
   cout << "The value of pPrimeInterest is: "
        << *pPrimeInterest << endl;
   cout << "The memory address of pPrimeInterest is: "
        << &pPrimeInterest << endl;
   return 0;
}
```

The output of the above program:

```
The value of pPrimeInterest is: 0.065
The memory address of pPrimeInterest is: 0x0066FD74
The value of pPrimeInterest is: 0.070
The memory address of pPrimeInterest is: 0x0066FD74.
```

Note: The above program declares the *pPrimeInterest* pointer on the heap and assigns to it a value of 0.065. Then the **delete** operator deletes the heap address that stores the value of 0.065. Finally, a new value is added to the heap address. You can see that after the **delete** statement executes, the *pPimeInterest* pointer still point to the same memory address.

Example 7.4.2

In the following program, we can create some objects of the class *Stocks* on the stack or on the heap and then manipulate them.

```
#include<iostream.h>
class Stocks{
public:
    int iNumShares;
   double dPurchasePricePerShare;
   double dCurrentPricePerShare;
};
double totalValue(Stocks* pCurStock){
        double dTotalValue;
        dTotalValue = pCurStock->dCurrentPricePerShar*pCurStock->iNumShares;
        return dTotalValue;
}
int main( ){
        //allocated on the stack with a pointer to the stack object
        Stocks stockPick;
        Stocks* pStackStock = &stockPick;
        pStackStock->iNumShares = 500;
        pStackStock-> dPurchasePricePerShare = 10.785;
        pStackStock-> dCurrentPricePerShare = 6.5;
        cout << totalValue(pStackStock) << endl;
        //allocated on the heap
        Stocks* pHeapStock = new Stocks;
        pHeapStock->iNumShares = 200;
        pHeapStock-> dPurchasePricePerShare = 32.5;
        pHeapStock-> dCurrentPricePerShare = 48.25;
        cout << totalValue(pHeapStock) << endl;
        return 0;
}
```

The output of the above program:

```
3250
9650
```

In the above program, the *new* operator in the statement:

```
        Stocks* pHeapStock = new Stocks;
```

invokes the constructor of the *Stocks* class to create a *Stocks* object on the heap and returns a pointer which is assigned to the pointer variable *pHeapStock*.

Note:

1. The *totalValue()* function is *not* a function member of the *Stocks* class. Rather, it is a function that is available to the entire program.

2. When declaring and using pointers and references to class objects, follow the same rules as you would when declaring and using pointers and references to structures. You can use the **indirect member selection operator** (->) to access class members through a pointer to an object either on stack or on the heap.

As we will see, using *new* and *delete* offers other benefits as well. In particular, *new* invokes the constructor and *delete* invokes the class'destructor.

7.5 POINTERS AS CLASS MEMBERS

A class can contain any C++ data type. Thus, the inclusion of a pointer variable in a class should not seem surprising.

In some cases, pointers as class members are advantageous. For example, assume that in the class *Book* we need to store a book title. Rather than using a fixed length character array as a data member to hold each book title, we could include a pointer member to a character array and then allocate the correct size array for each book title as it is needed.

Example 7.5.1

```
#include <iostream.h>
#include <string.h>

// class declaration
class Book
{
  private:
    char *title;   // a pointer to a book title
  public:
    Book(char * = NULL);   // constructor with a default value
    void showtitle();      // display the title
};

// class implementation
Book::Book(char *strng)
{
  title = new char[strlen(strng)+1];  // allocate memory
  strcpy(title,strng);                // store the string
}
void Book::showtitle()
{
  cout << title << endl;
  return;
```

```
}

int main()
{
  Book  book1("DOS Primer");   // create 1st title
  Book  book2("A Brief History of Western Civilization");  // 2nd title

  book1.showtitle();   // display book1's title
  book2.showtitle();   // display book2's title

  return 0;
}
```

The output of the above program:

```
DOS Primer
A Brief History of Western Civilization
```

The body of the *Book()* constructor contains two statements. The first statement performs two taks: First, the statement allocates enough storage for the length of the name parameter plus one to accommodate the end-of-string null character, '\n'. Next, the address of the first allocated character position is assigned to the pointer variable *title*.