Objects and Classes

1.1 A Simple Class

The first program contains a class and two objects of that class. Although it's simple, the program demonstrates the syntax and general features of classes in C++. Here's the listing for the SMALLOBJ program:

```
// smallobj
#include <iostream.h>
class smallobj                        //define a class
{
  private:
      int somedata;                   //class data
  public:
      void setdata(int d)             //member function to set data
      { somedata = d; }
     void showdata()              //member function to display data
     { cout << "Data is " << somedata << endl; }
};
int main()
{  smallobj s1, s2;             //define two objects of class smallobj
   s1.setdata(1066);            //call member function to set data
   s2.setdata(1776);
   s1.showdata();              //call member function to display data
   s2.showdata();
  return 0;
}
```

The class smallobj defined in this program contains one data item and two member functions. The two member functions provide the only access to the data item from outside the class. The first member function sets the data item to a value, and the second displays the value. Each of the two objects is given a value, and each displays its value.

Here's the output of the program:

```
Data is 1066 ← object s1 displayed this
Data is 1776 ← object s2 displayed this
```

Placing data and functions together into a single unit is a central idea in object-oriented programming. This is shown in Figure 1 below:

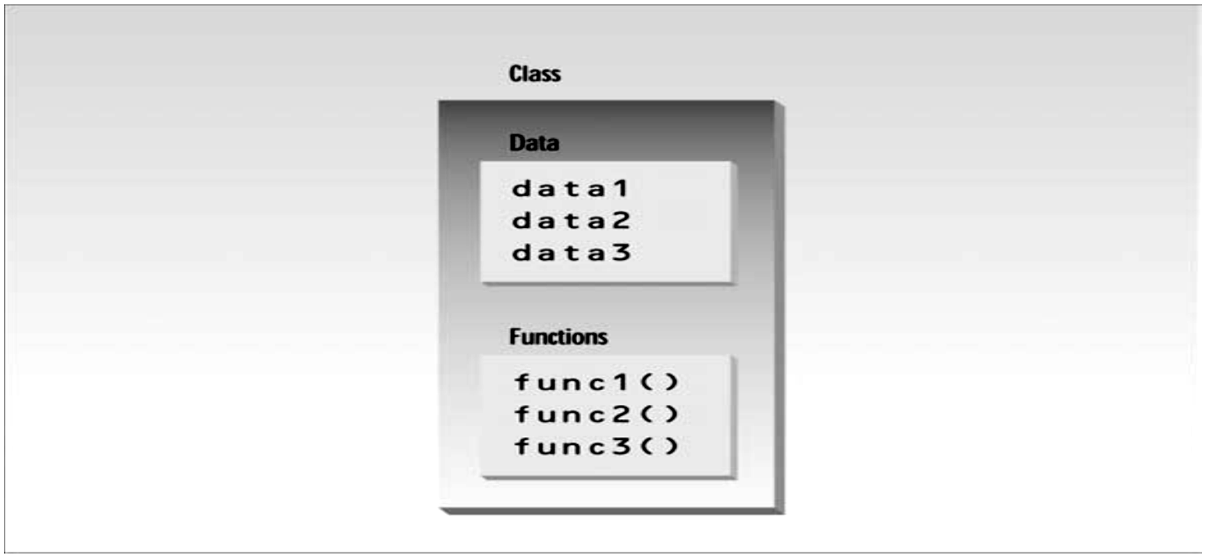
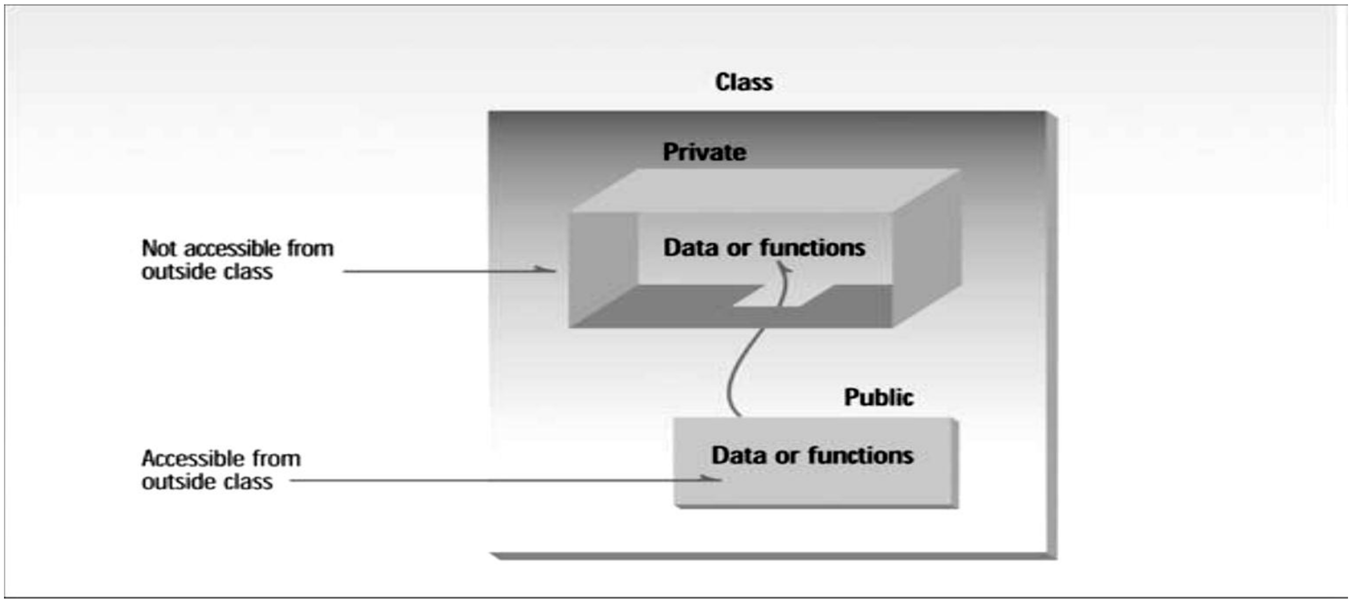


FIGURE 1 Classes contain data and functions.

1.1.1 private and public

The body of the class contains two keywords: private and public. What is their purpose?A key feature of object-oriented programming is data hiding. This term means that data is concealed within a class so that it cannot be accessed mistakenly by functions outside the class. The primary mechanism for hiding data is to put it in a class and make it private. Private data or functions can only be accessed from within the class. Public data or functions, on the other hand, are accessible from outside the class. This is shown in Figure 2.

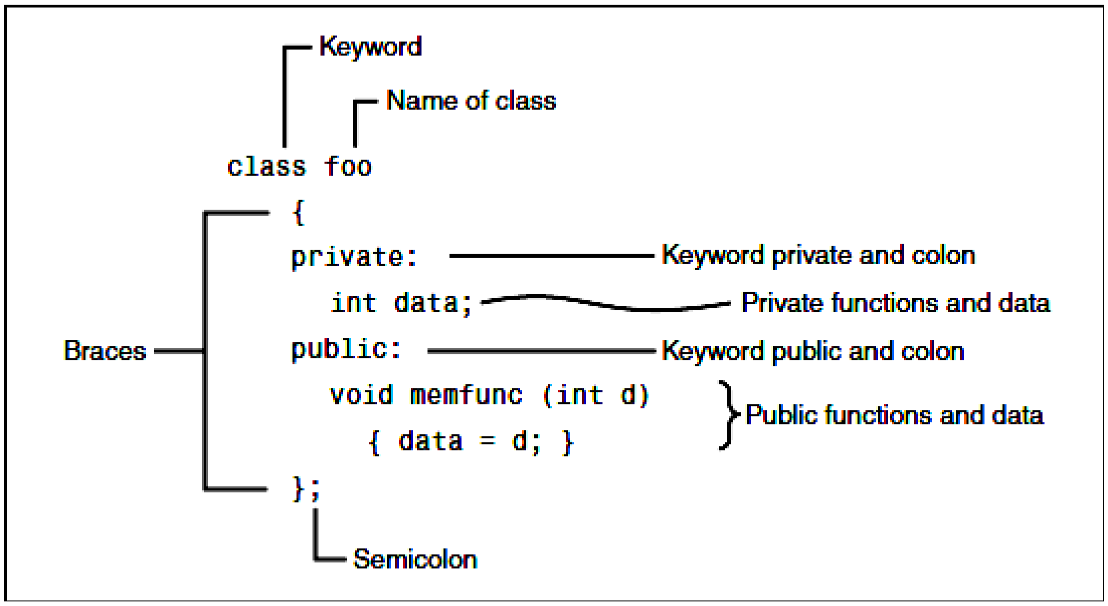

Class
Private
Data or functions
Not accessible from outside class
Public
Data or functions
Accessible from outside class

1.1.2 Class Data

The smallobj class contains one data item: somedata, which is of type int. The data items within a class are called data members. There can be any number of data members in a class, just as there can be any number of data items in a structure. The data member somedata follows the keyword private, so it can be accessed from within the class, but not from outside.

1.1.3 Member Functions

Member functions are functions that are included within a class. (In some object-oriented languages, are called methods). There are two member functions in smallobj: setdata() and showdata().Because setdata() and showdata() follow the keyword public, they can be accessed from outside the class. Figure 3 shows the syntax of a class definition.


Keyword
Name of class
Braces
Keyword private and colon
Private functions and data
Keyword public and colon
Public functions and data
Semicolon

```
class foo
{
private:
   int data;
public:
   void memfunc (int d)
     { data = d; }
};
```

FIGURE 3 Syntax of a class definition.

1.1.4 Using the Class

Now that the class is defined, let's see how main() makes use of it. We'll see how objects are defined, and, once defined, how their member functions are accessed.

1.1.5 Defining Objects

The first statement in main() smallobj s1, s2; defines two objects, s1 and s2, of class smallobj. The definition of the class smallobj does not create any objects. It only describes how they will look when they are created. Defining an object is similar to defining a variable of any data type: Space is set aside for it in memory. Defining objects in this way means creating them. This is also called instantiating them.

1.1.6 Calling Member Functions

The next two statements in main() call the member function setdata():

s1.setdata(1066);

s2.setdata(1776);

Why the object names s1 and s2 are connected to the function names with a period? This syntax is used to call a member function that is associated with a specific object. Because setdata() is a member function of the smallobj class, it must always be called in connection with an object of this class. It doesn't make sense to say
setdata(1066); by itself, because a member function is always called to act on a specific object, not on the class in general. To use a member function, the dot operator (the period) connects the object name and the member function.

The first call to setdata() s1.setdata(1066); executes the setdata() member function of the s1 object. This function sets the variable somedata in object s1 to the value 1066. The second call s2.setdata(1776); causes the variable somedata in s2 to be set to 1776. Now we have two objects whose somedata variables have different values, as shown in Figure 4.

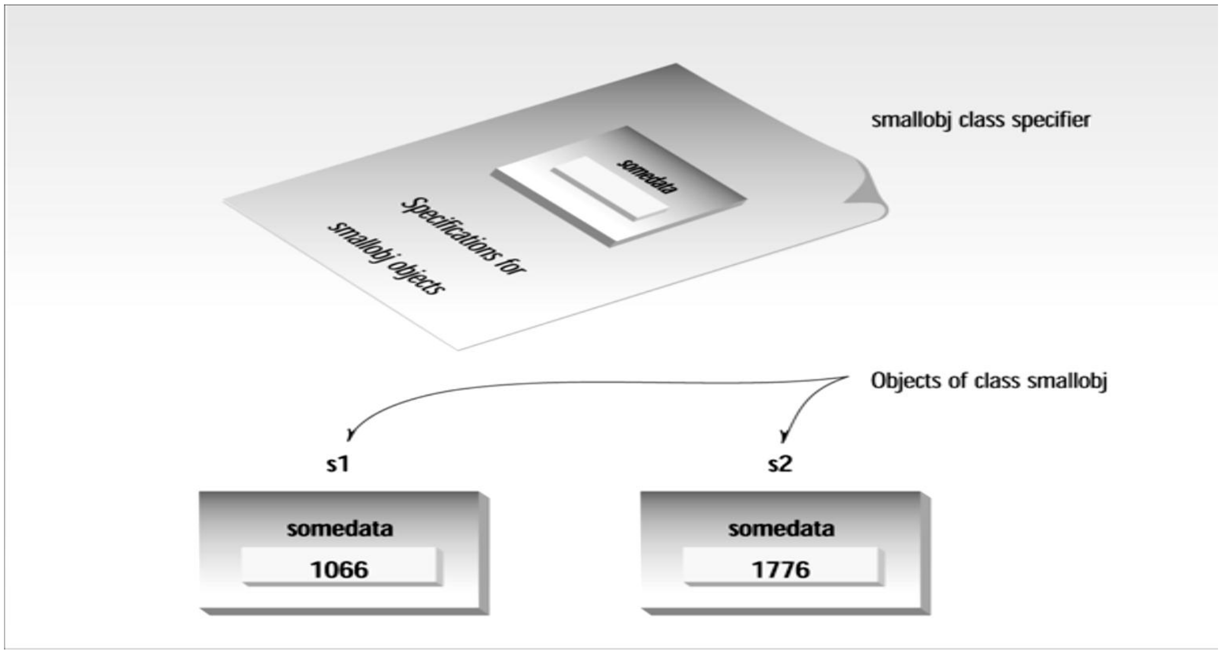

smallobj class specifier
somedata
Specifications for smallobj objects
Objects of class smallobj
s1
s2
somedata
1066
somedata
1776

FIGURE 4 Two objects of class smallobj.

Similarly, the following two calls to the showdata() function will cause the two objects to display their values:

s1.showdata();

s 2.showdata();

1.1.7 Messages

Some object-oriented languages refer to calls to member functions as messages. Thus the call s1.showdata(); can be thought of as sending a message to s1 telling it to show its data. Talking about messages emphasizes that objects are discrete entities and that we communicate with them by calling their member functions.

1.2 Widget Parts as Objects

```
// widget part as an object
#include <iostream.h>
class part //define class
{
   private:
         int modelnumber; //ID number of widget
         int partnumber; //ID number of widget part
         float cost; //cost of part
  public:
         void setpart(int mn, int pn, float c) //set data
                {
                        modelnumber = mn;
                        partnumber = pn;
                        cost = c;
                }
         void showpart() //display data
                {
                        cout << "Model " << modelnumber;
                        cout << ", part " << partnumber;
                        cout << ", costs $" << cost << endl;
                }
};
 main()
{  part part1;                                //define object  of class part
            part1.setpart(6244, 373, 217.55);    //call member function
            part1.showpart();                        //call member function
}
```

In this example only one object of type part is created: part1. The member function setpart() sets the three data items in this part to the values 6244, 373, and 217.55. The member function showpart() then displays these values. Here's the output:

```
Model 6244, part 373, costs $217.55
```

1.2.1 C++ Objects as Data Types

Here's another kind of entity C++ objects can represent: variables of a user-defined data type.

```
// englobj.cpp
// objects using English measurements
#include <iostream.h>
class Distance //English Distance class
{  private:
        int feet;
        float inches;
   public:
        void setdist(int ft, float in) //set Distance to args
                {       feet = ft; inches = in; }
        void getdist() //get length from user
                {
                        cout << "\nEnter feet: "; cin >> feet;
                        cout << "Enter inches: "; cin >> inches;
                }
        void showdist() //display distance
                {       cout << feet << "\'-" << inches << '\"'; }
};
main()
{
        Distance dist1, dist2;              //define two lengths
        dist1.setdist(11, 6.25);            //set dist1
        dist2.getdist();                    //get dist2 from user
        cout << "\ndist1 = "; dist1.showdist();
        cout << "\ndist2 = "; dist2.showdist();
        cout << endl;
}
```

In main(),two objects of class Distance are define: dist1 and dist2. The first is given a value using the setdist() member function with the arguments 11 and 6.25, and the second is given a value that is supplied by the user. Here's a sample interaction with the program:

```
Enter feet: 10
Enter inches: 4.75
dist1 = 11'-6.25" ← provided by arguments
dist2 = 10'-4.75" ← input by the user
```

1.3 Constructors

The ENGLOBJ example shows two ways that member functions can be used to give values to the data items in an object. Sometimes, however, it's convenient if an object can initialize itself when it's first created, without requiring a separate call to a member function. Automatic initialization is carried out using a special member function called a constructor. A constructor is a member function that is executed automatically whenever an object is created.

1.3.1 A Counter Example

This example, COUNTER, provides a counter variable that can be modified only through its member functions.

```
 // counter.cpp
// object represents a counter variable
#include <iostream.h>
class Counter
{      private:
            int count; //count
       public:
            Counter (): count (0) //constructor
            { /*empty body*/ }
      void inc_count() //increment count
     { count++; }
     int get_count() //return count
     { return count; }
};
 main()
{
   Counter c1, c2; //define and initialize
   cout << “\nc1=” << c1.get_count(); //display
   cout << “\nc2=” << c2.get_count();
   c1.inc_count(); //increment c1
   c2.inc_count(); //increment c2
   c2.inc_count(); //increment c2
   cout << “\nc1=” << c1.get_count(); //display again
   cout << “\nc2=” << c2.get_count();
   cout << endl;
}
```

The Counter class has one data member: count, of type unsigned int (since the count is always positive). It has three member functions: the constructor Counter(), which we’ll look at in a moment; in c_count(), which adds 1 to count; and get_count(), which returns the current value of count.

1.3.2 Automatic Initialization

When an object of type Counter is first created, we want its count to be initialized to 0. It’s more reliable and convenient, especially when there are a great many objects of a given class, to cause each object to initialize itself when it’s created. In the Counter class, the constructor Counter() does this. This function is called automatically whenever a new object of type Counter is created. Thus in main() the statement:

Counter c1, c2;

creates two objects of type Counter. As each is created, its constructor, Counter(), is executed. This function sets the count variable to 0. So the effect of this single statement is to not only create two objects, but also to initialize their count variables to 0.

1.3.3 Same Name as the Class

Aspects of constructor functions: First, it has exactly the same name (Counter in this example) as the class of which they are members. This is one way the compiler knows they are constructors. Second, no return type is used for constructors. Why not? Since the constructor is called automatically by the system, there's no program for it to return anything to; a return value wouldn't make sense. This is the second way the compiler knows they are constructors.

1.3.4 Initialize List

One of the most common tasks a constructor carries out is initializing data members. In the Counter class the constructor must initialize the count member to 0. If multiple members must be initialized, they're separated by commas. The result is the initializer list:

```
Some Class() : m1(7), m2(33), m2(4) ← initializer list
        {  }
```

1.3.5 Counter Output

The main() part of this program exercises the Counter class by creating two counters, c1 and c2. It causes the counters to display their initial values, which—as arranged by the constructor—are 0. It then increments c1 once and c2 twice, and again causes the counters to display themselves (non-criminal behavior in this context). Here's the output:

```
c1=0
c2=0
c1=1
c2=2
```

As you can see, the constructor is executed twice—once for c1 and once for c2—when the statement Counter c1, c2; is executed in main().

1.4 Destructors

We've seen that a special member function—the constructor—is called automatically when an object is first created. You might guess that another function is called automatically when an object is destroyed. This is indeed the case. Such a function is called a destructor. A destructor has the same name as the constructor (which is the same as the class name) but is preceded by a tilde:

```
class Foo
{
        private:
                int data;
        public:
                Foo() : data(0)                //constructor (same name as class)
             ~Foo()                            //destructor (same name with tilde)
                 { }
};
```

Like constructors, destructors do not have a return value. They also take no arguments (the assumption being that there's only one way to destroy an object). The most common use of destructors is to deallocate memory that was allocated for the object. If we allocate memory when we create an object, it's reasonable to deallocate the memory when the object is no longer needed.

<u>Exercises</u>

1. Create a class that imitates part of the functionality of the basic data type int. Call the class Int (note different capitalization). The only data in this class is an int variable. Include member functions to initialize an

Int to 0, to initialize it to an int value, to display it (it looks just like an int), and to add two Int values.
Write a program that exercises this class by creating one uninitialized and two initialized Int values, adding the two initialized values and placing the response in the uninitialized value, and then displaying this result.(<u>Instead of having z=x+y, and x,y and z are int , we could have z.add(x,y) and x,y and z are of type Int.)</u>

<u>Solutions to Exercises</u>

1.

```
// ex6_1.cpp
// uses a class to model an integer data type
#include <iostream.h>
class Int //(not the same as int)
{
     private:
            int i;
     public:
         Int() //create an Int
          { i = 0; }
        Int(int ii) //create and initialize an Int
          { i = ii; }
        void add(Int i2, Int i3) //add two Ints
           { i = i2.i + i3.i; }
        void display() //display an Int
           { cout << i; }
  };
 int main()
  {   Int Int1(7); //create and initialize an Int
       Int Int2(11); //create and initialize an Int
       Int Int3; //create an Int
       Int3.add(Int1, Int2); //add two Ints
       cout << "\nInt3 = "; Int3.display(); //display result
       cout << endl;
       return 0;
  }
```