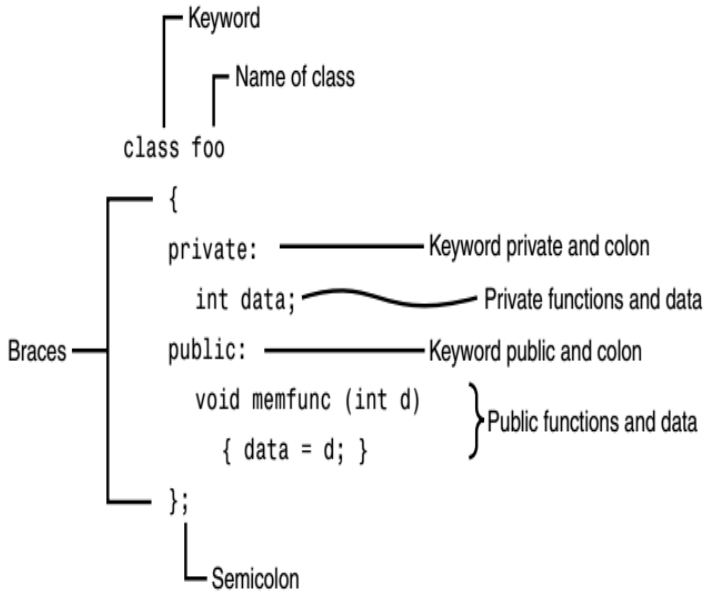
A Simple Class

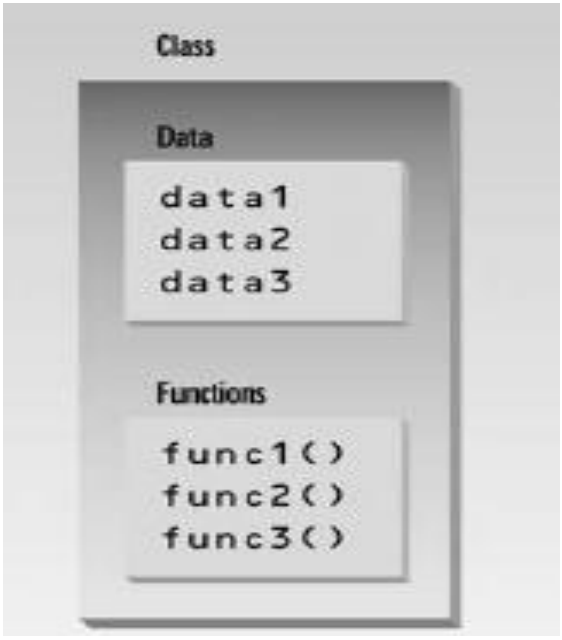
Our first program contains a class and two objects of that class. Although it's simple, the program demonstrates the syntax and general features of classes in C++. Here's the listing for the SMALLOBJ program:

```
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////
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. Thefirst m ember function sets the data item to a value, and the second displays the value.

Placing data and functions together into a single entity is a central idea in object-oriented programming. This is shown in Figure 6.1.





Classes and Objects

Recall from Chapter 1 that an object has the same relationship to a class that a variable has to a data type. An object is said to be an instance of a class, in the same way my 1954 Chevrolet is an instance of a vehicle. In SMALLOBJ, the class—whose name is smallobj—is defined in the first part of the program. Later, in main(), we define two objects—s1 and s2—that are instances of that class.

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

We'll begin by looking in detail at the first part of the program—the definition of the class smallobj. Later we'll focus on what main() does with objects of this class.

Defining the Class

Here's the definition (sometimes called a specifier) for the class smallobj, copied from the SMALLOBJ listing:

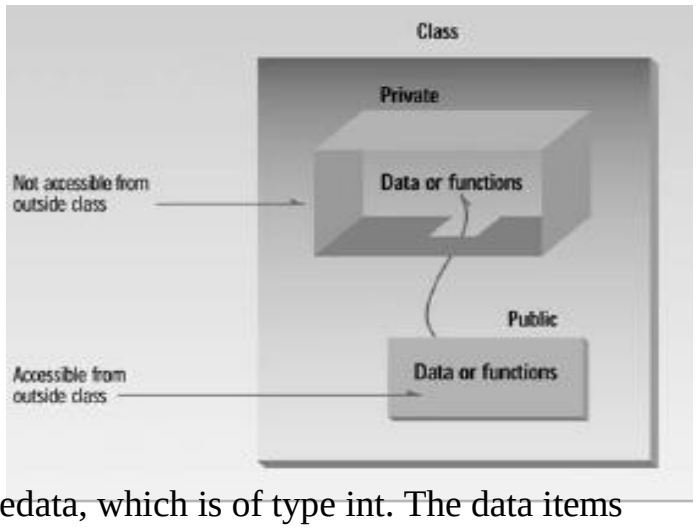
```
class smallobj //define a class
{
private:
int somedata; //class data
public:
void setdata(int d) //member function to set data
```

```
{ somedata = d; }
void showdata() //member function to display data
{ cout << "\nData is " << somedata; }
};
```

The definition starts with the keyword class, followed by the class name—smallobj in this example. Like a structure, the body of the class is delimited by braces and terminated by a semicolon.

Private and Public

The body of the class contains two unfamiliar keywords: private and public. 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 6.2.



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 (or sometimes member data). 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.

Member Functions

Member functions are functions that are included within a class. There are two member functions in smallobj: setdata() and showdata().
The function bodies of these functions have been written on the same line as the braces that delimit them. You could also use the more traditional format for these function definitions:

void setdata(int d)
{
somedata = d;
}
	void showdata()
	{
	cout << "\nData is " << somedata;
	}

Because setdata() and showdata() follow the keyword public, they can be accessed from outside the class. We'll see how this is done in a moment. Usually the data within a class is private and the functions are public. This is a result of the way classes are used. The data is hidden so it will be safe from accidental manipulation, while the functions that operate on the data are public so they can be accessed from outside the class. However, there is no rule that says data must be private and functions public; in some circumstances you may find you'll need to use private functions and public data.

The member functions in the smallobj class perform operations that are quite common in classes: setting and retrieving the data stored in the class. The setdata() function accepts a value as a parameter and sets the somedata variable to this value. The showdata() function displays the value stored in somedata.

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- 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, just as a structure definition describes how a structure will look but doesn't create any structure variables. It is objects that participate in program operations. *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. The term instantiating arises because an instance of the class is created. An object is an instance of a class. *Objects are sometimes called instance variables*.

2-Calling Member Functions

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

s1.setdata(1066);

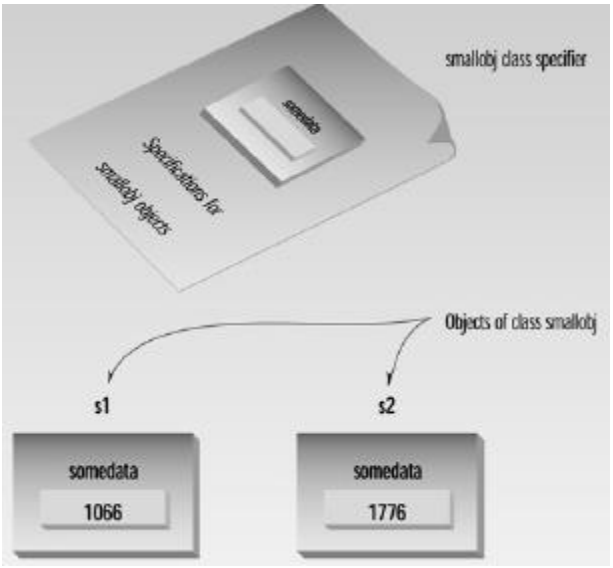
s2.setdata(1776);

These statements don't look like normal function calls because the object names s1 and s2 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);

Not only does this statement not make sense, but the compiler will issue an error message if you attempt it. Member functions of a class can be accessed only by an object of that class.

To use a member function, the dot operator (the period) connects the object name and the member function. The syntax is similar to the way we refer to structure members, but the parentheses signal that we're executing a member function rather than referring to a data item.



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 6.4.

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

their values:

s1.showdata();

s2.showdata();

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. it's like sending a message to the secretary in the sales department asking for a

list of products sold in the southwest distribution area

In many programming situations, objects in programs represent physical objects: things that can be felt or seen. These situations provide vivid examples of the correspondence

between the program and the real world. We'll look at two such situations: widget parts and graphics circles.

Widget Parts as Objects Example 1

The smallobj class in the last example had only one data item. Let's look at an example of a somewhat more ambitious class. We'll create a class based on the structure for the widget parts inventory, last seen in such examples as PARTS in Chapter 4, "Structures." Here's the listing for OBJPART:

```
#include <iostream>
////////////////////////////////////////////////////////////
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;
}
};
int main()
{
part part1; //define object
// of class part
part1.setpart(6244, 373, 217.55F); //call member function
part1.showpart(); //call member function
return 0;
}
```

This program features the class part. Instead of one data item, as SMALLOBJ had, this class has three: modelnumber, partnumber, and cost. A single member function, setpart(), supplies values to all three data items at once. Another function, showpart(), displays the values stored in all three items.
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
```

This is a somewhat more realistic example than SMALLOBJ. If you were designing an inventory program you might actually want to create a class something like part. It's an example of a C++ object representing a physical object in the real world—a widget part.

English measurements as Objects Example 2

Here's another kind of entity C++ objects can represent: variables of a user-defined data type. We'll use objects to represent distances measured in the English system, as discussed in Chapter 4. Here's the listing for ENGLOBJ:

```
#include <iostream>
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 << '\"'; }
};
int main()
{
Distance dist1, dist2; //define two lengths
```

```
dist1.setdist(11, 6.25); //set dist1
dist2.getdist(); //get dist2 from user
//display lengths
cout << "\ndist1 = "; dist1.showdist();
cout << "\ndist2 = "; dist2.showdist();
cout << endl;
return 0;
}
```

In this program, the class Distance contains two data items, feet and inches. This is similar to the Distance structure seen in examples in Chapter 4, but here the class Distance also has three member functions: setdist(), which uses arguments to set feet and inches; getdist(), which gets values for feet and inches from the user at the keyboard; and showdist(), which displays the distance in feet-and-inches format. The value of an object of class Distance can thus be set in either of two ways. In main(), we define two objects of class Distance: 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
```