Structures and Classes

In fact, you can use structures in almost exactly the same way that you use classes. The only formal difference between class and struct is that in a class the members are private by default, while in a structure they are public by default.

Here's the format we've been using for classes:

```
class foo
{
private:
int data1;
public:
void func();
};
```

Because private is the default in classes, this keyword is unnecessary. You can just as well write and the data1 will still be private. Many programmers prefer this style.

```
class foo
{
int data1;
public:
void func();
};
```

We like to include the private keyword because it offers an increase in clarity. If you want to use a structure to accomplish the same thing as this

class, you can dispense with the keyword public, provided you put the public members before the private ones

```
struct foo
{
void func();
private:
int data1;
};
```

Since public is the default. However, in most situations programmers don't use a struct this way. They use structures to group only data, and classes to group both data and functions.

Classes, Objects, and Memory

we've probably given you the impression that each object created from a class contains separate copies of that class's data and member functions. This is a good first approximation, since it Objects and Classes emphasizes that objects are complete, self-contained entities, designed using the class definition.

The mental image here is of cars (objects) rolling off an assembly line, each one made according to a blueprint (the class definitions). Actually, things are not quite so simple. It's true that each object has its own separate data items. On the other hand, contrary to what you may have been led to believe, all the objects in a given class use the same member functions. The member functions are created and placed in memory only once—when they are defined in the class definition. This makes sense; there's really no point in duplicating all the member functions in a class every time you create another object of that class,

since the functions for each object are identical. The data items, however, will hold different values, so there must be a separate instance of each data item for each object.

Data is therefore placed in memory when each object is defined, so there is a separate set of data for each object. Figure 6.8 shows how this looks.

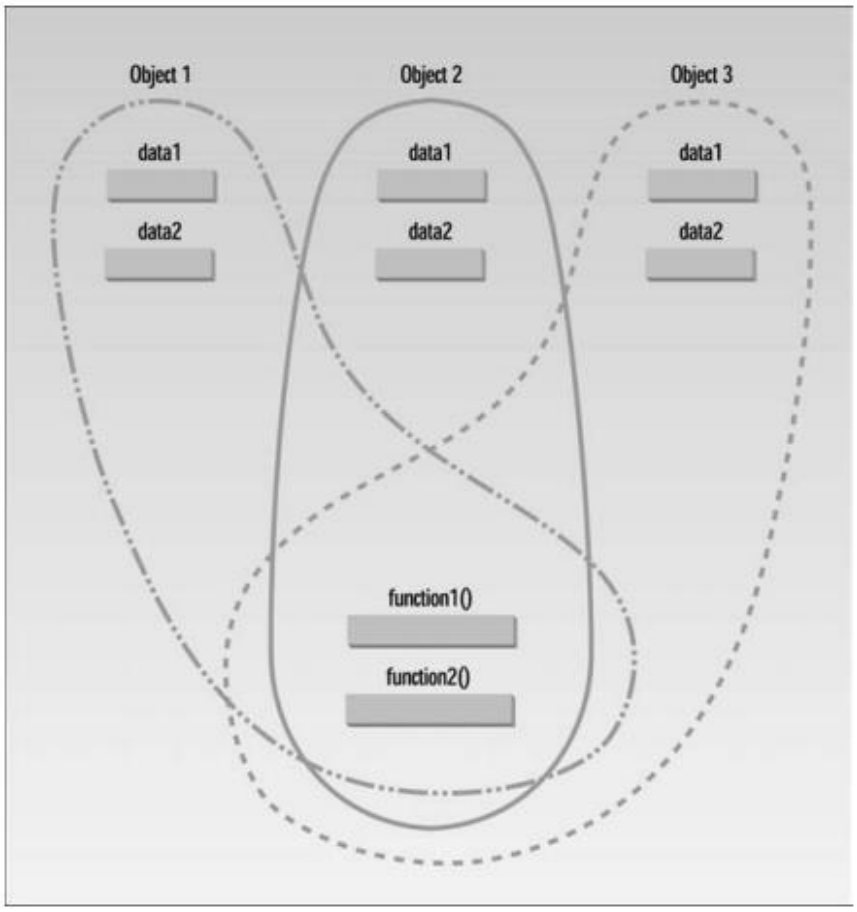

FIGURE 6.8
Objects, data, functions, and memory.

Static Class Data

In the SMALLOBJ example at the beginning of this chapter there are two objects of type smallobj, so there are two instances of somedata in memory. However, there is only one instance of the functions setdata() and showdata(). These functions are shared by all the objects of the class. There is no conflict because (at least in a single-threaded system) only one function is executed at a time.

In most situations you don't need to know that there is only one member function for an entire class. It's simpler to visualize each object as containing both its own data and its own member functions. But in some situations, such as in estimating the size of an executing program, it's helpful to know what's happening behind the scenes. Having said that each object contains its own separate data, if a data item in a class is declared as static, only one such item is created for the entire class, no matter how many objects there are. A static data item is useful when all objects of the same class must share a common item of information. A member variable defined as static has characteristics similar to a normal static variable: It is visible only within the class, but its lifetime is the entire program. It continues to exist even if there are no objects of the class. However, while a normal static variable is used to retain information between calls to a function, static class member data is used to share information among the objects of a class.

Uses of Static Class Data

why would you want to use static member data? As an example, suppose an object needed to know how many other objects of its class were in the program. In a road-racing game, for example, a race car might want to

know how many other cars are still in the race. In this case a static variable count could be included as a member of the class. All the objects would have access to this variable. It would be the same variable for all of them; they would all see the same count.

An Example of Static Class Data

Here's an example, STATDATA that demonstrates a simple static data member:

```
// statdata.cpp
// static class data
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////
class foo
{
private:
static int count; //only one data item for all objects
//note: "declaration" only!
public:
foo() //increments count when object created
{ count++; }
int getcount() //returns count
{ return count; }
};
//----------------------------------------------------------
int foo::count = 0; //*definition* of count
////////////////////////////////////////////////////////////
```

```
int main()
{
foo f1, f2, f3; //create three objects
cout << "count is " << f1.getcount() << endl; //each object
cout << "count is " << f2.getcount() << endl; //sees the
cout << "count is " << f3.getcount() << endl; //same value
return 0;
}
```

The class **foo** in this example has one data item, **count**, which is type **static int**.

The constructor for this class causes **count** to be incremented. In **main()** we define three objects of class **foo**. Since the constructor is called three times, **count** is incremented three times. Another member function, **getcount()**, returns the value in **count**. We call this function from all three objects, and—as we expected—each prints the same value. Here's the output:

```
count is 3 ←□□□ static data
count is 3
count is 3
```

If we had used an ordinary automatic variable—as opposed to a static variable—for count, each constructor would have incremented its own private copy of **count** once, and the output would have been

```
count is 1 ←□□□ automatic data
count is 1
count is 1
```

Static class variables are not used as often as ordinary non-static variables, but they are important in many situations. Figure 6.9 shows how static variables compare with automatic variables.

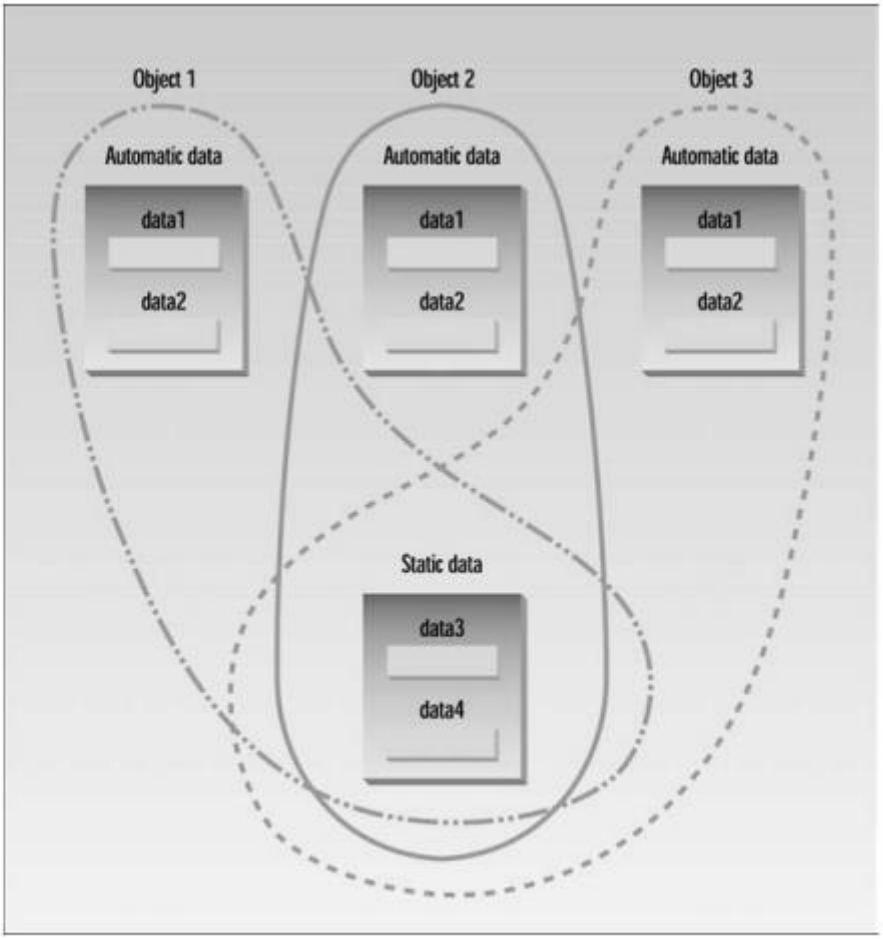

FIGURE 6.9
Static versus automatic member variables.

Separate Declaration and Definition

Static member data requires an unusual format. Ordinary variables are usually declared (the compiler is told about their name and type) and defined (the compiler sets aside memory to hold the variable) in the same statement. Static member data, on the other hand, requires two separate statements. The variable's declaration appears in the class definition, but

the variable is actually defined outside the class, in much the same way as a global variable.

Why is this two-part approach used? If static member data were defined inside the class (as it actually was in early versions of C++), it would violate the idea that a class definition is only a blueprint and does not set aside any memory.

Putting the definition of static member data outside the class also serves to emphasize that the memory space for such data is allocated only once, before the program starts to execute, and that one static member variable is accessed by an entire class; each object does not have its own version of the variable, as it would with ordinary member data. In this way a static member variable is more like a global variable.

It's easy to handle static data incorrectly, and the compiler is not helpful about such errors. If you include the declaration of a static variable but forget its definition, there will be no warning from the compiler. Everything looks fine until you get to the linker, which will tell you that you're trying to reference an undeclared global variable. This happens even if you include the definition but forget the class name (the foo:: in the STATDATA example).