جامعـــة الفيـــوم
كلية الحاسبات والذكاء الاصطناعى
مؤسسة تعليمية معتمدة

الكتاب الجامعى
2023/2024

جامعة الفيوم

كلية الحاسبات والذكاء الاصطناعى

Object Oriented Programming

2nd Level Students

Faculty of Computers and Artificial Intelligence

Object Oriented Programming

2nd Level Students

Prepared By

Computer Science Department

2023 – 2024

Contents

Chapter 1

Classes and Objects

In this chapter, we'll put these ideas together to create classes. We'll introduce several classes, starting with simple ones and working toward more complicated examples. We'll focus first on the details of classes and objects. At the end of the chapter, we'll take a wider view, discussing what is to be gained by using the OOP approach.

1.1 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:

```
// smallobj.cpp
// demonstrates a small, simple object
#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. The first member function sets the data item to a value, and the second displays the value. (This may sound like Greek, but we'll see what these terms mean as we go along.) Placing data and functions together into a single entity is a central idea in object-oriented programming. This is shown in Figure 1.1.

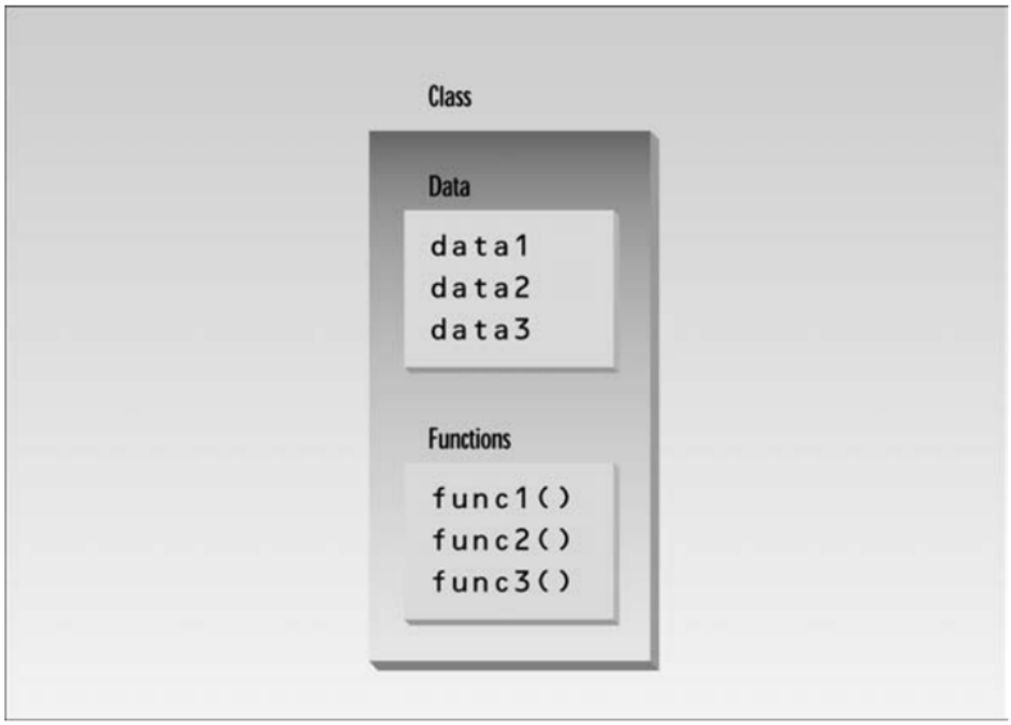

FIGURE 1.1: Classes contain data and functions.

This information should be prepared and react to ongoing streaming information utilizing continuous inquiries, so it is conceivable to perform the fly's examination inside the stream continuously. Stream preparing arrangements must have the option to deal with a constant, high volume of information from different sources placing into thought accessibility, adaptability, and adaptation to internal failure. Therefore, the idea of enormous details will be clarified and canvassed in the accompanying area [1].

1.2 Classes and Objects

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
Data is 1776
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.

1.3 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. (Don't forget the semicolon. Remember, data constructs such as structures and classes end with a semicolon, while control constructs such as functions and loops do not.)

1.4 private and public

The body of the class contains two unfamiliar keywords: private and public. What is their purpose? A key feature of object-oriented programming is data hiding. This term does not refer to the activities of particularly paranoid programmers; rather it 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 **??**

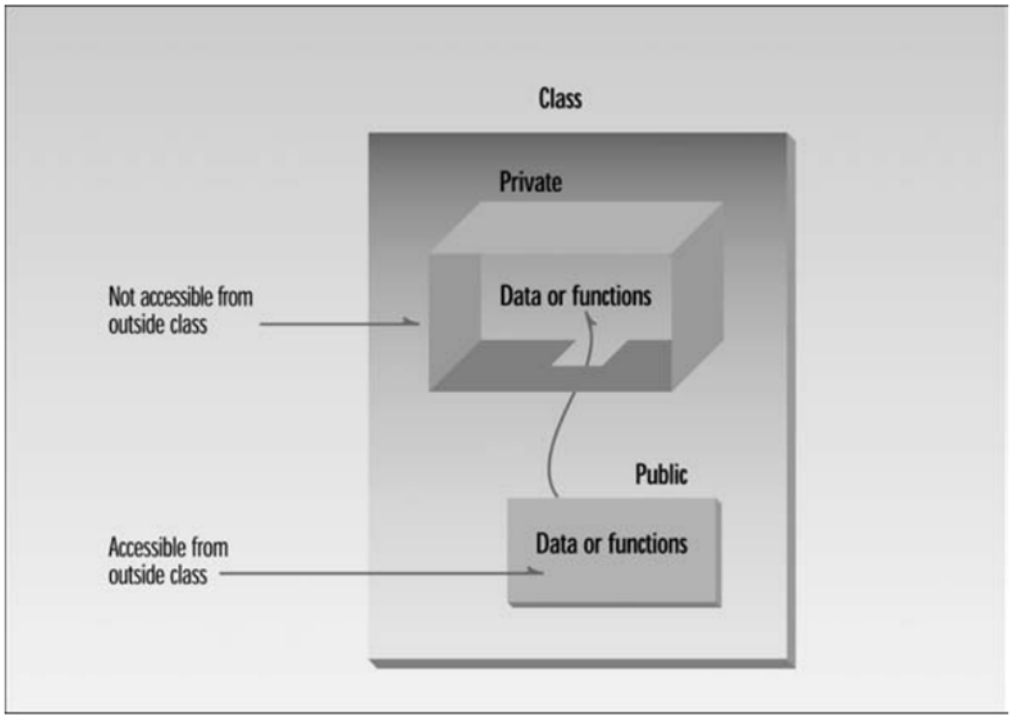

FIGURE 1.2: Private and public.

1.5 why hidden the data and from whom

Don't confuse data hiding with the security techniques used to protect computer databases. To provide a security measure you might, for example, require a user to supply a password before granting access to a database. The password is meant to keep unauthorized or malevolent users from altering (or often even reading) the data. Data hiding, on the other hand, means hiding data from parts of the program that don't need to access it. More specifically, one class's data is hidden from other classes. Data hiding is designed to protect well-intentioned programmers from honest mistakes. Programmers who really want to can figure out a way to access private data, but they will find it hard to do so by accident.

1.6 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.

1.7 Member functions

Member functions are functions that are included within a class. (In some object-oriented languages, such as Smalltalk, member functions are called methods; some writers use this termin C++ as well.) 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 thatdelimit them. You could also use the more traditional format for these function definitions:

```
void setdata(int d)
{
    somedata = d;
}
```

and

```
void showdata()
{
    cout << "\nData is " << somedata;
}
```

However, when member functions are small, it is common to compress their definitions this way to save space.

1.8 Functions Are Public, Data Is Private

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.

1.9 Member Functions Within Class Definition

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.
Note that the member functions setdata() and showdata() are definitions in that the actual code for the function is contained within the class definition. (The functions are not definitions in the sense that memory is set aside for the function code; this doesn't happen until an object of the class is created.) Member functions defined inside a class this way are created as inline functions by default.

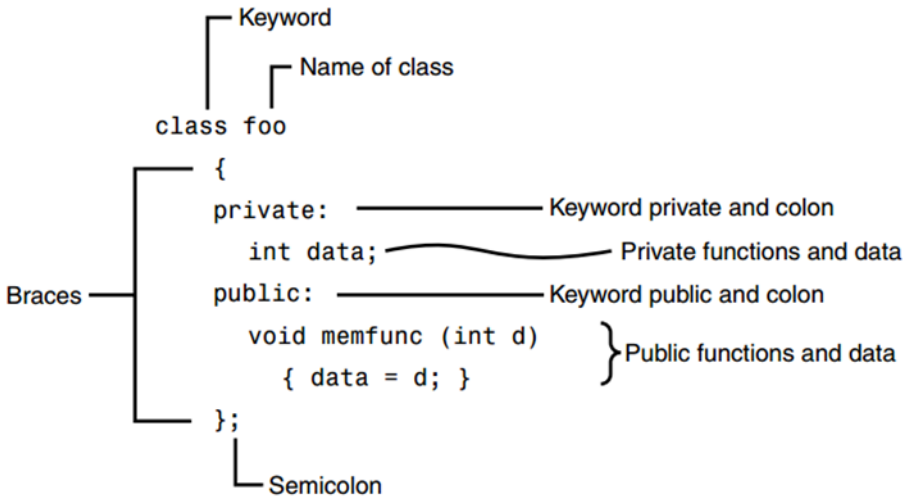

FIGURE 1.3: Syntax of a class definition.

1.10 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.

1.11 Defining Objects

The first statement in main()
smallobj s1, s2;
defines two objects, s1 and s2, of class smallobj. Remember that 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 (that is, a specific example) of a class. Objects are sometimes called instance variables.

1.12 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. Why are the object names s1 and s2 connected to the function names with a period? This strange 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. Attempting to access the class this way would be like trying to drive the blueprint of a car. 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 dot operator is also called the class member access operator.)
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 1.4

Similarly, the following two calls to the showdata() function will cause the two objects to display their values:
s1.showdata();
s2.showdata();

1.13 C++ Objects as Physical Objects

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

1.14 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. (These are not the same ambitious classes discussed in political science courses.)

```
// objpart.cpp
// widget part as an object
```

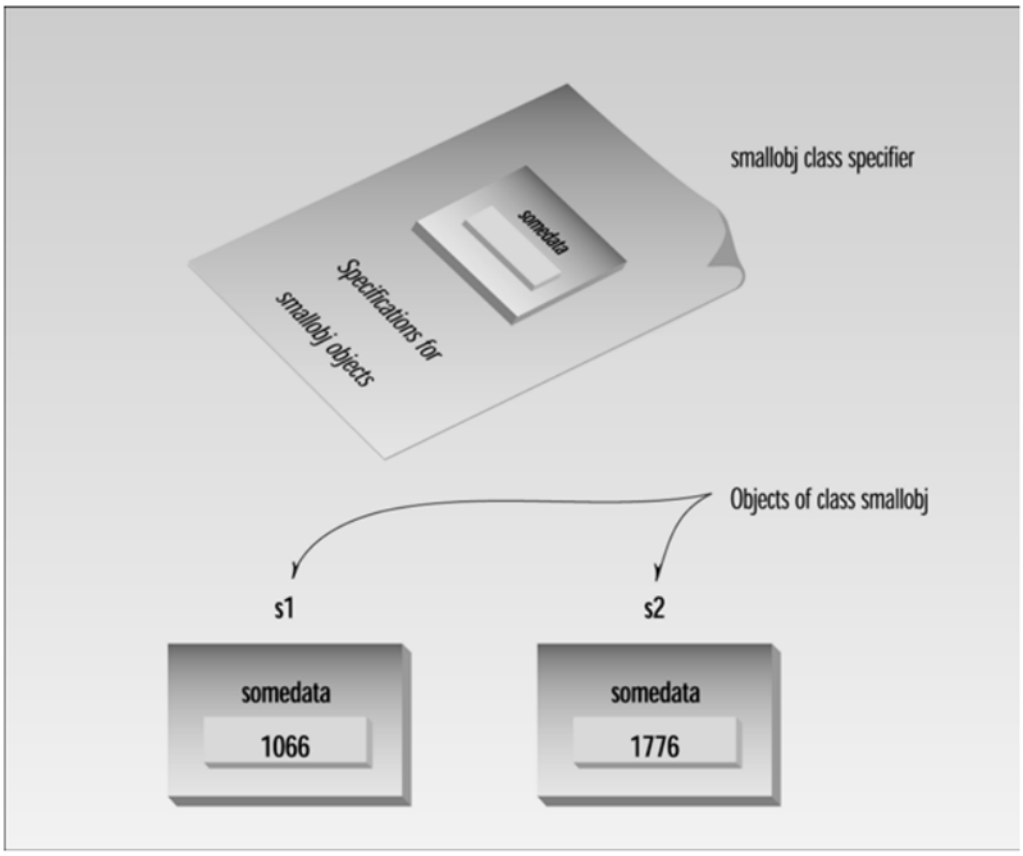

FIGURE 1.4: Two objects of class smallobj.

```
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////
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.

1.15 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. Here's the listing for ENGLOBJ:

```
// englobj.cpp
// objects using English measurements
#include <iostream>
using namespace std;
//////////////////////////////////////////////////////////////
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 lecture 2, 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

1.16 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. (The term constructor is sometimes abbreviated ctor, especially in comments in program listings.)

1.16.1 A Counter Example

As an example, we'll create a class of objects that might be useful as a general-purpose programming element. A counter is a variable that counts things. Maybe it counts file accesses, or the number of times the user presses the Enter key, or the number of customers entering a bank. Each time such an event takes place, the counter is incremented (1 is added to it). The counter can also be accessed to find the current count.
Let's assume that this counter is important in the program and must be accessed by many different functions. In procedural languages such as C, a counter would probably be implemented as a global variable. However, as we noted in Chapter 1, global variables complicate the program's design and may be modified accidentally. 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>
using namespace std;
////////////////////////////////////////////////////////////////
class Counter
{
private:
    unsigned int count; //count
public:
    Counter() : count(0) //constructor
         { /*empty body*/ }
    void inc_count() //increment count
        { count++; }
    int get_count() //return count
        { return count; }
};
////////////////////////////////////////////////////////////////
int 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;
    return 0;
}
```

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; inc_count(), which adds 1 to count; and get_count(), which returns the current value of count.

1.16.2 Automatic Initialization

When an object of type Counter is first created, we want its count to be initialized to 0. After all, most counts start at 0. We could provide a set_count() function to do this and call it with an argument of 0, or we could provide a zero_count() function, which would always set count to 0. However, such functions would need to be executed every time we created a Counter object. Counter c1; //every time we do this,
c1.zero_count(); //we must do this too
This is mistake-prone, because the programmer may forget to initialize the object after creating it. 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.

1.16.3 Same Name as the Class

There are some unusual aspects of constructor functions. First, it is no accident that they have 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.

1.16.4 Initializer 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. You might think that this would be done in the constructor's function body, like this:

```
count()
{ count = 0; }
```

However, this is not the preferred approach (although it does work). Here's how you should initialize a data member:

```
    count() : count(0)
{ }
```

The initialization takes place following the member function declarator but before the function body. It's preceded by a colon. The value is placed in parentheses following the member data. If multiple members must be initialized, they're separated by commas. The result is the initializer list (sometimes called by other names, such as the member-initialization list).
someClass() : m1(7), m2(33), m2(4) ←—— initializer list
Why not initialize members in the body of the constructor? The reasons are complex, but have to do with the fact that members initialized in the initializer list are given a value before the constructor even starts to execute. This is important in some situations. For example, the initializer list is the only way to initialize const member data and references. Actions more complicated than simple initialization must be carried out in the constructor body, as with ordinary functions.

1.16.5 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
If this isn't enough proof that the constructor is operating as advertised, we can rewrite the constructor to print a message when it executes.
Counter() : count(0)
cout « "I'm the constructor";
Now the program's output looks like this:
I'm the constructor
I'm the constructor

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()

1.17 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 by the constructor. We'll investigate these activities in Chapter 10, "Pointers." Until then we won't have much use for destructors.

1.18 Objects as Function Arguments

Our next program adds some embellishments to the ENGLOBJ example. It also demonstrates some new aspects of classes: constructor overloading, defining member functions outside the class, and—perhaps most importantly—objects as function arguments. Here's the listing for ENGLCON:

```
// englcon.cpp
// constructors, adds objects using member function
#include <iostream>
using namespace std;
```

```
////////////////////////////////////////////////////////////
class Distance //English Distance class
{
private:
    int feet;
    float inches;
public: //constructor (no args)
    Distance() : feet(0), inches(0.0)
    { }
    //constructor (two args)
    Distance(int ft, float in) : 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 << '\"'; }
    void add_dist( Distance, Distance ); //declaration
};
//-----------------------------------------------------------
//add lengths d2 and d3
void Distance::add_dist(Distance d2, Distance d3)
{
    inches = d2.inches + d3.inches; //add the inches
    feet = 0; //(for possible carry)
    if(inches >= 12.0) //if total exceeds 12.0,
    { //then decrease inches
        inches -= 12.0; //by 12.0 and
        feet++; //increase feet
    } //by 1
    feet += d2.feet + d3.feet; //add the feet
}
////////////////////////////////////////////////////////////
int main()
{
    Distance dist1, dist3; //define two lengths
    Distance dist2(11, 6.25); //define and initialize dist2
    dist1.getdist(); //get dist1 from user
```

```
    dist3.add_dist(dist1, dist2); //dist3 = dist1 + dist2
    //display all lengths
    cout << ‘‘\ndist1 = ‘‘; dist1.showdist();
    cout << ‘‘\ndist2 = ‘‘; dist2.showdist();
    cout << ‘‘\ndist3 = ‘‘; dist3.showdist();
    cout << endl;
    return 0;
}
```

This program starts with a distance dist2 set to an initial value and adds to it a distance dist1, whose value is supplied by the user, to obtain the sum of the distances. It then displays all three distances:
Enter feet: 17
Enter inches: 5.75
dist1 = 17′-5.75″
dist2 = 11′-6.25″
dist3 = 29′-0″
Let's see how the new features in this program are implemented.

1.18.1 Overloaded Constructors

It's convenient to be able to give variables of type Distance a value when they are first created. That is, we would like to use definitions like
Distance width(5, 6.25);

which defines an object, width, and simultaneously initializes it to a value of 5 for feet and 6.25 for inches. To do this we write a constructor like this:
Distance(int ft, float in) : feet(ft), inches(in)
This sets the member data feet and inches to whatever values are passed as arguments to the constructor. So far so good. However, we also want to define variables of type Distance without initializing them, as we did in ENGLOBJ.
Distance dist1, dist2;
In that program there was no constructor, but our definitions worked just fine. How could they work without a constructor? Because an implicit no-argument constructor is built into the program automatically by the compiler, and it's this constructor that created the objects, even though we didn't define it in the class. This no-argument constructor is called the default constructor. If it weren't created automatically by the constructor, you wouldn't be able to create objects of a class for which no constructor was defined. Often we want to initialize data members in the default (no-argument) constructor as well. If we let the default constructor do it, we don't really know what values the data members may be given. If we care what values they may be given, we need to explicitly define the constructor. In ENGLECON we show how this looks:
Distance() : feet(0), inches(0.0) //default constructor
{ } //no function body, doesn't do anything

The data members are initialized to constant values, in this case the integer value 0 and the float value 0.0, for feet and inches respectively. Now we can use objects initialized with the no-argument constructor and be confident that they represent no distance (0 feet plus 0.0 inches) rather than some arbitrary value. Since there are now two explicit constructors with the same name, Distance(), we say the constructor is overloaded. Which of the two constructors is executed when an object is created depends on how many arguments are used in the definition:
Distance length; // calls first constructor
Distance width(11, 6.0); // calls second constructor

1.19 Member Functions Defined Outside the Class

So far we have seen member functions that were defined inside the class definition. This need not always be the case. ENGLCON shows a member function, add_dist(), that is not defined within the Distance class definition. It is only declared inside the class, with the statement
void add_dist(Distance, Distance);
This tells the compiler that this function is a member of the class but that it will be defined outside the class declaration, someplace else in the listing. In ENGLCON the add_dist() function is defined following the class definition.

```
    void Distance::add_dist(Distance d2, Distance d3)
{
inches = d2.inches + d3.inches; //add the inches
feet = 0; //(for possible carry)
if(inches >= 12.0) //if total exceeds 12.0,
{ //then decrease inches
inches -= 12.0; //by 12.0 and
feet++; //increase feet
} //by 1
feet += d2.feet + d3.feet; //add the feet
}
```

The declarator in this definition contains some unfamiliar syntax. The function name, add_dist(), is preceded by the class name, Distance, and a new symbol—the double colon (::). This symbol is called the scope resolution operator. It is a way of specifying what class something is associated with. In this situation, Distance::add_dist() means "the add_dist() member function of the Distance class." Figure **??** shows its usage.

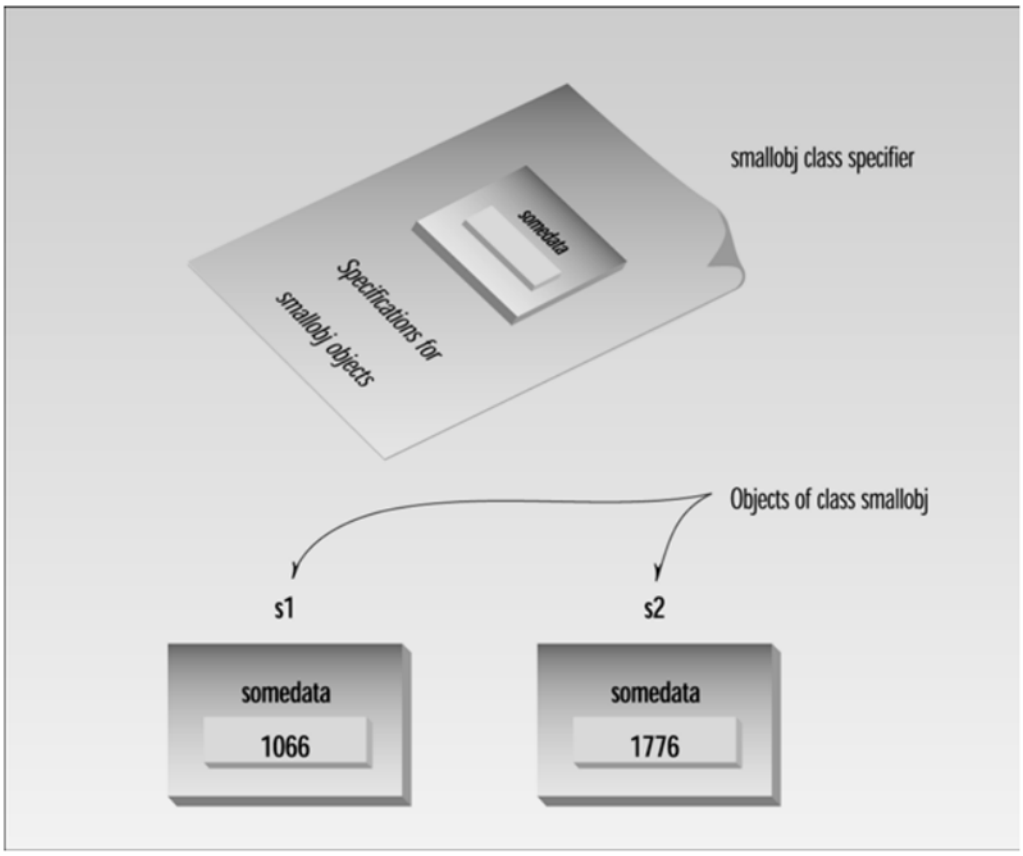

FIGURE 1.5: The scope resolution operator.

1.20 Objects as Arguments

Now we can see how ENGLCON works. The distances dist1 and dist3 are created using the default constructor (the one that takes no arguments). The distance dist2 is created with the constructor that takes two arguments, and is initialized to the values passed in these arguments. A value is obtained for dist1 by calling the member function getdist(), which obtains values from the user.
Now we want to add dist1 and dist2 to obtain dist3. The function call in main()
dist3.add_dist(dist1, dist2); does this. The two distances to be added, dist1 and dist2, are supplied as arguments to add_dist(). The syntax for arguments that are objects is the same as that for arguments that are simple data types such as int: The object name is supplied as the argument. Since add_dist() is a member function of the Distance class, it can access the private data in any object of class Distance supplied to it as an argument, using names like dist1.inches and dist2.feet.
Close examination of add_dist() emphasizes some important truths about member functions. A member function is always given access to the object for which it was called: the object connected to it with the dot

operator. But it may be able to access other objects. In the following statement in ENGLCON, what objects can add_dist() access?
dist3.add_dist(dist1, dist2);
Besides dist3, the object for which it was called, it can also access dist1 and dist2, because they are supplied as arguments. You might think of dist3 as a sort of phantom argument; the member function always has access to it, even though it is not supplied as an argument. That's what this statement means: "Execute the add_dist() member function of dist3." When the variables feet and inches are referred to within this function, they refer to dist3.feet and dist3.inches.
Notice that the result is not returned by the function. The return type of add_dist() is void. The result is stored automatically in the dist3 object. Figure **??** shows the two distances dist1 and dist2 being added together, with the result stored in dist3.
To summarize, every call to a member function is associated with a particular object (unless it's a static function; we'll get to that later). Using the member names alone (feet and inches), the function has direct access to all the members, whether private or public, of that object. It also has indirect access, using the object name and the member name, connected with the dot operator (dist1.inches or dist2.feet) to other objects of the same class that are passed as arguments.

1.21 The Default Copy Constructor

We've seen two ways to initialize objects. A no-argument constructor can initialize data members to constant values, and a multi-argument constructor can initialize data members to values passed as arguments. Let's mention another way to initialize an object: you can initialize it with another object of the same type. Surprisingly, you don't need to create a special constructor for this; one is already built into all classes. It's called the default copy constructor. It's a one-argument constructor whose argument is an object of the same class as the constructor. The ECOPYCON program shows how this constructor is used.

```
// ecopycon.cpp
// initialize objects using default copy constructor
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////

class Distance //English Distance class
{
private:
    int feet;
    float inches;
public:
//constructor (no args)
    Distance() : feet(0), inches(0.0)
```

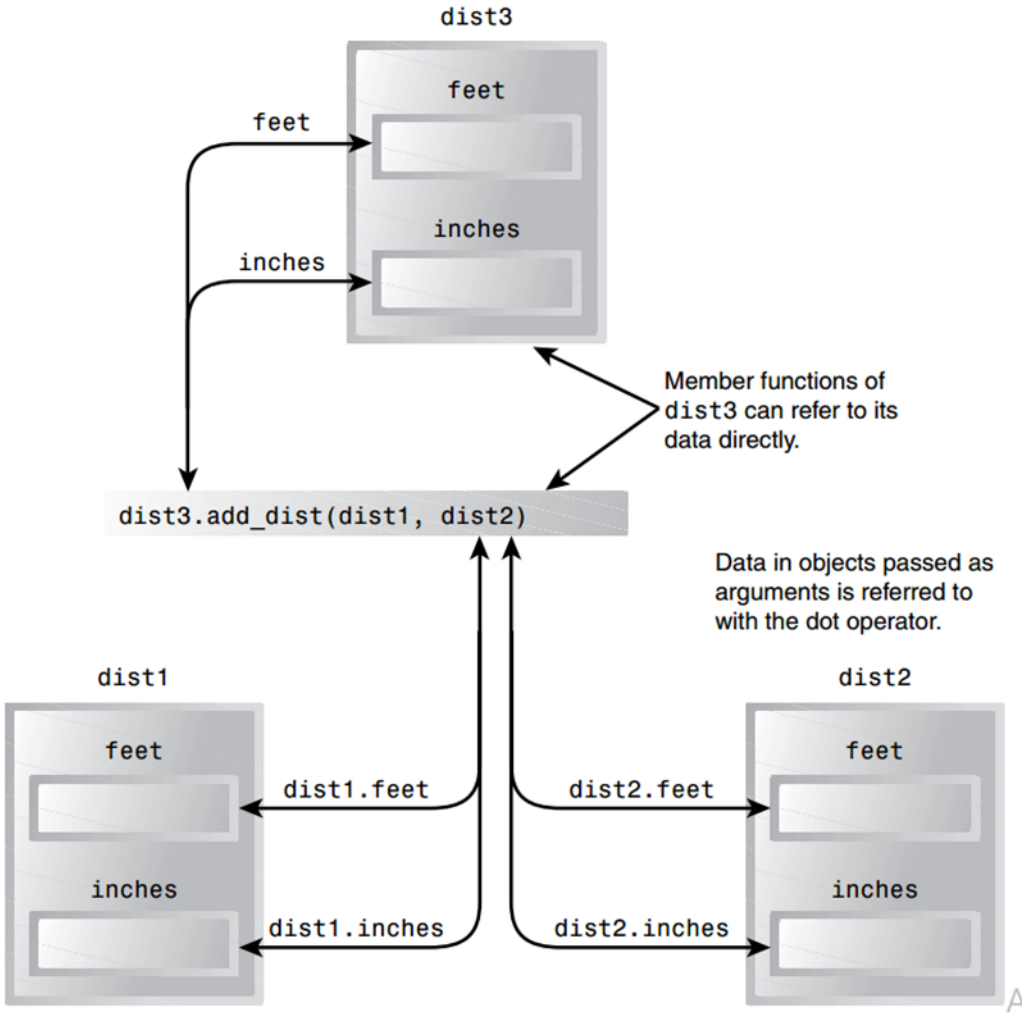

FIGURE 1.6: Result in this object.

```
{ }
//Note: no one-arg constructor
//constructor (two args)
Distance(int ft, float in) : 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(11, 6.25); //two-arg constructor
    Distance dist2(dist1); //one-arg constructor
    Distance dist3 = dist1; //also one-arg constructor
    //display all lengths
    cout << "\ndist1 = "; dist1.showdist();
    cout << "\ndist2 = "; dist2.showdist();
    cout << "\ndist3 = "; dist3.showdist();
    cout << endl;
    return 0;
}
```

We initialize dist1 to the value of 11'-6.25" using the two-argument constructor. Then we define two more objects of type Distance, dist2 and dist3, initializing both to the value of dist1. You might think this would require us to define a one-argument constructor, but initializing an object with another object of the same type is a special case. These definitions both use the default copy constructor. The object dist2 is initialized in the statement
Distance dist2(dist1);

This causes the default copy constructor for the Distance class to perform a member-by-member copy of dist1 into dist2. Surprisingly, a different format has exactly the same effect, causing dist1 to be copied member-by-member into dist3:
Distance dist3 = dist1;
Although this looks like an assignment statement, it is not. Both formats invoke the default copy constructor, and can be used interchangeably. Here's the output from the program:
dist1 = 11'-6.25"
dist2 = 11'-6.25"
dist3 = 11'-6.25"
This shows that the dist2 and dist3 objects have been initialized to the same value as dist1.

1.22 Returning Objects from Functions

In the ENGLCON example, we saw objects being passed as arguments to functions. Now we'll see an example of a function that returns an object. We'll modify the ENGLCON program to produce ENGLRET:

```
     // englret.cpp
// function returns value of type Distance
```

```
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////
class Distance //English Distance class
{
    private:
    int feet;
    float inches;
public: //constructor (no args)
    Distance() : feet(0), inches(0.0)
    { } //constructor (two args)
    Distance(int ft, float in) : 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 << '\"'; }
    Distance add_dist(Distance); //add
};
//-----------------------------------------------------------
//add this distance to d2, return the sum
Distance Distance::add_dist(Distance d2)
{
    Distance temp; //temporary variable
    temp.inches = inches + d2.inches; //add the inches
    if(temp.inches >= 12.0) //if total exceeds 12.0,
        { //then decrease inches
            temp.inches -= 12.0; //by 12.0 and
            temp.feet = 1; //increase feet
         } //by 1
    temp.feet += feet + d2.feet; //add the feet
    return temp;
}
////////////////////////////////////////////////////////////
int main()
{
    Distance dist1, dist3; //define two lengths
```

```
    Distance dist2(11, 6.25); //define, initialize dist2
    dist1.getdist(); //get dist1 from user
    dist3 = dist1.add_dist(dist2); //dist3 = dist1 + dist2
    //display all lengths
    cout << ‘‘\ndist1 = ‘‘; dist1.showdist();
    cout << ‘‘\ndist2 = ‘‘; dist2.showdist();
    cout << ‘‘\ndist3 = ‘‘; dist3.showdist();
    cout << endl;
    return 0;
}
```

The ENGLRET program is very similar to ENGLCON, but the differences reveal important aspects of how functions work with objects

1.23 Arguments and Objects

In ENGLCON, two distances were passed to add_dist() as arguments, and the result was stored in the object of which add_dist() was a member, namely dist3. In ENGLRET, one distance, dist2, is passed to add_dist() as an argument. It is added to the object, dist1, of which add_dist() is a member, and the result is returned from the function. In main(), the result is assigned to dist3 in the statement
dist3 = dist1.add_dist(dist2);
Here's the add_dist() function from ENGLRET:

```
//add this distance to d2, return the sum
Distance Distance::add_dist(Distance d2)
{
    Distance temp; //temporary variable
    temp.inches = inches + d2.inches; //add the inches
    if(temp.inches >= 12.0) //if total exceeds 12.0,
    { //then decrease inches
        temp.inches -= 12.0; //by 12.0 and
        temp.feet = 1; //increase feet
    } //by 1
    temp.feet += feet + d2.feet; //add the feet
    return temp;
}
```

Compare this with the same function in ENGLCON. As you can see, there are some subtle differences. In the ENGLRET version, a temporary object of class Distance is created. This object holds the sum until it can be returned to the calling program. The sum is calculated by adding two distances. The first is the object of which add_dist() is a member, dist1. Its member data is accessed in the function as feet and inches. The

second is the object passed as an argument, dist2. Its member data is accessed as d2.feet and d2.inches. The result is stored in temp and accessed as temp.feet and temp.inches. The temp object is then returned by the function using the statement
return temp;
and the statement in main() assigns it to dist3. Notice that dist1 is not modified; it simply supplies data to add_dist(). Figure 1.7 shows how this looks

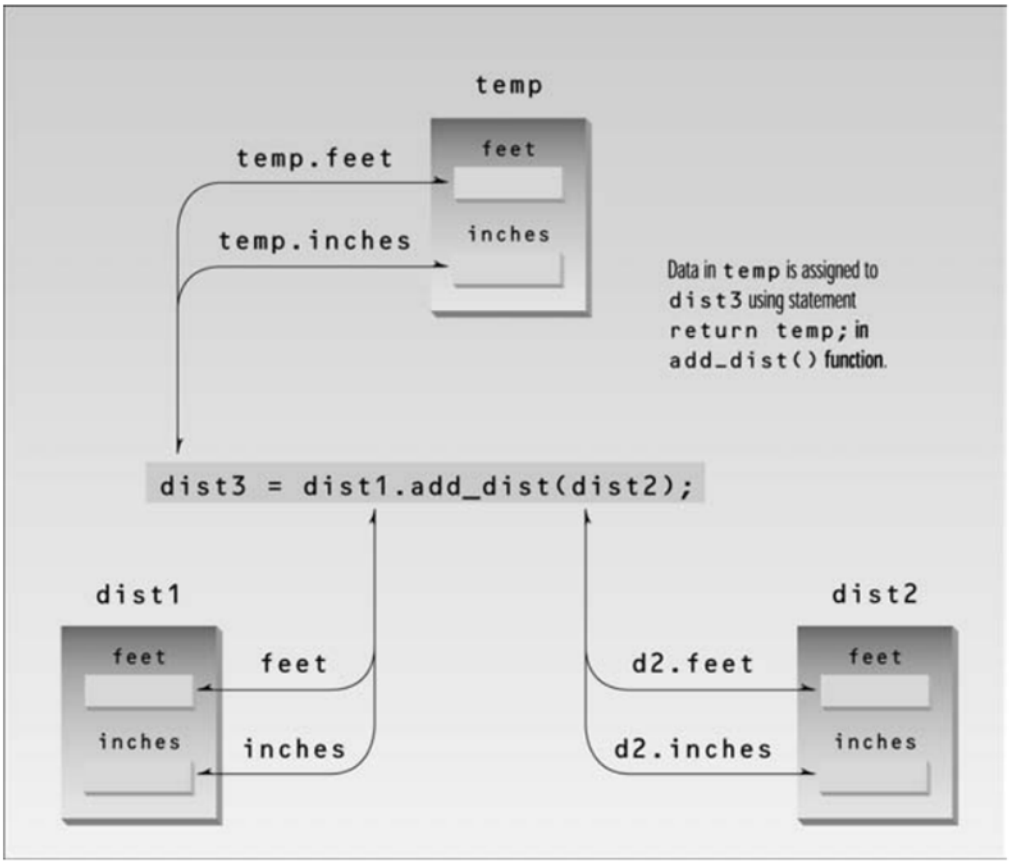

FIGURE 1.7: Result returned from the temporary object.

1.24 Structures and Classes

The examples so far in this book have portrayed structures as a way to group data and classes as a way to group both data and functions. 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

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

and the data will still be private. Many programmers prefer this style. 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.

1.25 Classes, Objects, and Memory

We have 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 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 1.8 shows how this looks.

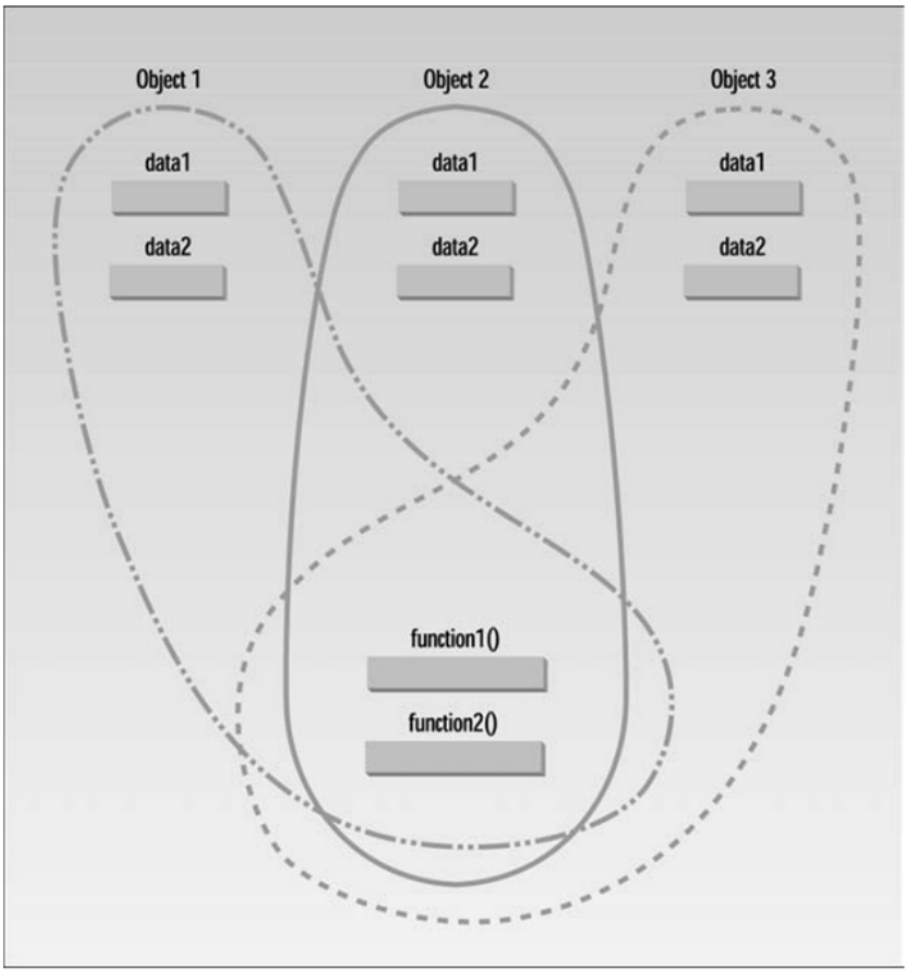

FIGURE 1.8: Objects, data, functions, and memory.

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.

1.26 Static Class Data

Having said that each object contains its own separate data, we must now amend that slightly. 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 itslifetime 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.

1.26.1 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.

1.26.2 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 1.9 shows how static variables compare with automatic variables.

1.27 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,

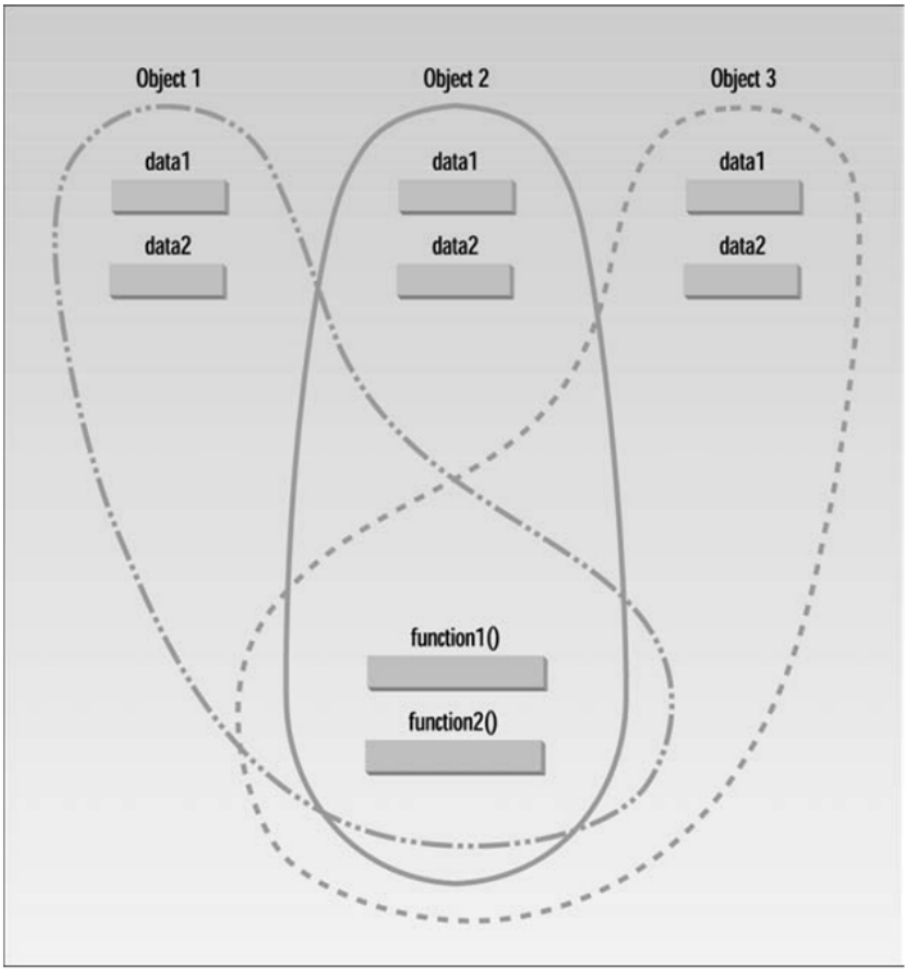

FIGURE 1.9: Static versus automatic member variables.

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 TATDATA example).

1.28 const and Classes

We've seen several examples of const used on normal variables to prevent them from being modified, and in Chapter 5 we saw that const can be used with function arguments to keep a function from modifying a variable passed to it by reference. Now that we know about classes, we can introduce some other uses of const: on member functions, on member function arguments, and on objects. These concepts work together to provide some surprising benefits.

1.28.1 const Member Functions

A const member function guarantees that it will never modify any of its class's member data. The CONSTFU program shows how this works

```
//constfu.cpp
//demonstrates const member functions
/
class aClass
{
private:
    int alpha;
public:
    void nonFunc() //non-const member function
          { alpha = 99; } //OK
    void conFunc() const //const member function
          { alpha = 99; } //ERROR: can't modify a member
};
```

The non-const function nonFunc() can modify member data alpha, but the constant function conFunc() can't. If it tries to, a compiler error results.
A function is made into a constant function by placing the keyword const after the declarator but before the function body. If there is a separate function declaration, const must be used in both declaration and definition. Member functions that do nothing but acquire data from an object are obvious candidates for being made const, because they don't need to modify any data. Making a function const helps the compiler flag errors, and tells anyone looking at the listing that you intended the function not to modify anything in its object. It also makes possible the creation and use of const objects, which we'll discuss soon.

1.28.2 A Distance Example

To avoid raising too many subjects at once we have, up to now, avoided using const member functions in the example programs. However, there are many places where const member functions should be used. For example, in the Distance class, shown in several programs, the showdist() member function could be made const because it doesn't (or certainly shouldn't!) modify any of the data in the object for which it was called. It should simply display the data. Also, in ENGLRET, the add_dist() function should not modify any of the data in the object for which it was called. This object should simply be added to the object passed as an argument, and the resulting sum should be returned. We've modified the ENGLRET program to show how these two constant functions look. Note that const is used in both the declaration and the definition of add_dist(). Here's the listing for ENGCONST:

```
// engConst.cpp
// const member functions and const arguments to member functions
```

```
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////
class Distance //English Distance class
{
private:
    int feet;
    float inches;
public: //constructor (no args)
    Distance() : feet(0), inches(0.0)
        { } //constructor (two args)
    Distance(int ft, float in) : feet(ft), inches(in)
         { }
    void getdist() //get length from user
    {
        cout << "\nEnter feet: "; cin >> feet;
        cout << "Enter inches: "; cin >> inches;
    }
    void showdist() const //display distance
        { cout << feet << "\'-" << inches << '\"'; }
    Distance add_dist(const Distance&) const; //add
};
//-----------------------------------------------------------
//add this distance to d2, return the sum
Distance Distance::add_dist(const Distance& d2) const
{
    Distance temp; //temporary variable
    // feet = 0; //ERROR: can't modify this
    // d2.feet = 0; //ERROR: can't modify d2
    temp.inches = inches + d2.inches; //add the inches
    if(temp.inches >= 12.0) //if total exceeds 12.0,
    { //then decrease inches
        temp.inches -= 12.0; //by 12.0 and
        temp.feet = 1; //increase feet
    } //by 1
    temp.feet += feet + d2.feet; //add the feet
    return temp;
}
////////////////////////////////////////////////////////////
int main()
```

```
{
    Distance dist1, dist3; //define two lengths
    Distance dist2(11, 6.25); //define, initialize dist2
    dist1.getdist(); //get dist1 from user
    dist3 = dist1.add_dist(dist2); //dist3 = dist1 + dist2
    //display all lengths
    cout << ‘‘\ndist1 = ‘‘; dist1.showdist();
    cout << ‘‘\ndist2 = ‘‘; dist2.showdist();
    cout << ‘‘\ndist3 = ‘‘; dist3.showdist();
    cout << endl;
    return 0;
}
```

Here, showdist() and add_dist() are both constant member functions. In add_dist() we show in the first commented statement, feet = 0, that a compiler error is generated if you attempt to modify any of the data in the object for which this constant function was called.

1.29 const Member Function Arguments

We mentioned in Chapter 5 that if an argument is passed to an ordinary function by reference, and you don't want the function to modify it, the argument should be made const in the function declaration (and definition). This is true of member functions as well. In ENGCONST the argument to add_dist() is passed by reference, and we want to make sure that ENGCONST won't modify this variable, which is dist2 in main(). Therefore we make the argument d2 to add_dist() const in both declaration and definition. The second commented statement shows that the compiler will flag as an error any attempt by add_dist() to modify any member data of its argument dist2.

1.30 const Objects

In several example programs, we've seen that we can apply const to variables of basic types such as int to keep them from being modified. In a similar way, we can apply const to objects of classes. When an object is declared as const, you can't modify it. It follows that you can use only const member functions with it, because they're the only ones that guarantee not to modify it. The CONSTOBJ program shows an example.

```
// constObj.cpp
// constant Distance objects
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////
class Distance //English Distance class
{
```

```
private:
    int feet;
    float inches;
public: //2-arg constructor
    Distance(int ft, float in) : feet(ft), inches(in)
    { }
    void getdist() //user input; non-const func
    {
        cout << "\nEnter feet: "; cin >> feet;
        cout << "Enter inches: "; cin >> inches;
    }
    void showdist() const //display distance; const func
      { cout << feet << "\'-" << inches << '\"'; }
};
////////////////////////////////////////////////////////////
int main()
{
    const Distance football(300, 0);
    // football.getdist(); //ERROR: getdist() not const
    cout << "football = ";
    football.showdist(); //OK
    cout << endl;
    return 0;
}
```

A football field (for American-style football) is exactly 300 feet long. If we were to use the length of a football field in a program, it would make sense to make it const, because changing it would represent the end of the world for football fans. The CONSTOBJ program makes football a const variable. Now only const functions, such as showdist(), can be called for this object. Non-const functions, such as getdist(), which gives the object a new value obtained from the user, are illegal. In this way the compiler enforces the const value of football.

When you're designing classes it's a good idea to make const any function that does not modify any of the data in its object. This allows the user of the class to create const objects. These objects can use any const function, but cannot use any non-const function. Remember, using const helps the compiler to help you.

Chapter 2

Operator Overloading

Closely related to function overloading is operator overloading. In C++, you can overload most operators so that they perform special operations relative to classes that you create. For example, a class that maintains a stack might overload + to perform a push operation and – – to perform a pop. When an operator is overloaded, none of its original meanings are lost. Instead, the type of objects it can be applied to is expanded.
The ability to overload operators is one of C++'s most powerful features. It allows the full integration of new class types into the programming environment. After overloading the appropriate operators, you can use objects in expressions in just the same way that you use C++'s built-in data types. Operator overloading also forms the basis of C++'s approach to I/O.
You overload operators by creating operator functions. ill perform relative to the class upon which i twill work. An operator function is created using the keyword operator. Operator functions can be either members or nonmembers of a class. Nonmember operator functions are almost always friend functions of the class, however. The way operator functions are written differs between member and nonmember functions. Therefore, each will be examined separately, beginning with member operator functions.

2.1 Creating a Member Operator Function

A member operator function takes this general form:

```
ret-type class-name::operator#(arg-list)
{
// operations
}
```

Often, operator functions return an object of the class they operate on, but rettype can be any valid type. The # is a placeholder. When you create an operator function, substitute the operator for the #. For example, if you are overloading the operator, use operator/. When you are overloading a unary operator, arglist will be empty. When you are overloading binary operators, arglist will contain one parameter. (The reasons for this seemingly unusual situation will be made clear in a moment.) Here is a simple first example of operator overloading. This program creates a class called loc, which stores longitude and latitude values. It

overloads the + operator relative to this class. Examine this program carefully, paying special attention to the definition of operator+( ):

```
#include <iostream>
using namespace std;
class loc {
    int longitude, latitude;
public:
    loc() {}
    loc(int lg, int lt) {
        longitude = lg;
        latitude = lt;
    }
    void show() {
        cout << longitude << " ";
        cout << latitude << "\n";
    }
    loc operator+(loc op2);
};
// Overload + for loc.
loc loc::operator+(loc op2)
{
loc temp;
    temp.longitude = op2.longitude + longitude;
    temp.latitude = op2.latitude + latitude;
    return temp;
}
int main()
{
    loc ob1(10, 20), ob2( 5, 30);
    ob1.show(); // displays 10 20
    ob2.show(); // displays 5 30
    ob1 = ob1 + ob2;
    ob1.show(); // displays 15 50
    return 0;
}
```

As you can see, operator+( ) has only one parameter even though it overloads the binary + operator. (You might expect two parameters corresponding to the two operands of a binary operator.) The reason that operator+( ) takes only one parameter is that the operand on the left side of the + is passed implicitly to the function through the this pointer. The operand on the right is passed in the parameter op2. The fact that the

left operand is passed using this also implies one important point: When binary operators are overloaded, it is the object on the left that generates the call to the operator function. As mentioned, it is common for an overloaded operator function to return an object of the class it operates upon. By doing so, it allows the operator to be used in larger expressions. For example, if the operator+( ) function returned some other type, this expression would not have been valid:

```
ob1 = ob1 + ob2;
```

In order for the sum of ob1 and ob2 to be assigned to ob1, the outcome of that operation must be an object of type loc. Further, having operator+( ) return an object of type loc makes possible the following statement:

```
(ob1+ob2).show(); // displays outcome of ob1+ob2
```

In this situation, ob1+ob2 generates a temporary object that ceases to exist after the call to show( ) terminates. It is important to understand that an operator function can return any type and that the type returned depends solely upon your specific application. It is just that, often, an operator function will return an object of the class upon which it operates. One last point about the operator+( ) function: It does not modify either operand. Because the traditional use of the + operator does not modify either operand, it makes sense for the overloaded version not to do so either. (For example, 5+7 yields 12, but neither 5 nor 7 is changed.) Although you are free to perform any operation you want inside an operator function, it is usually best to stay within the context of the normal use of the operator. The next program adds three additional overloaded operators to the loc class: the –, the =, and the unary ++. Pay special attention to how these functions are defined.

```
#include <iostream>
using namespace std;
class loc {
    int longitude, latitude;
public:
    loc() {} // needed to construct temporaries
    loc(int lg, int lt) {
        longitude = lg;
        latitude = lt;
    }
    void show() {
        cout << longitude << " ";
        cout << latitude << "\n";
    }
    loc operator+(loc op2);
    loc operator-(loc op2);
    loc operator=(loc op2);
    loc operator++();
```

```
};
// Overload + for loc.
loc loc::operator+(loc op2)
{
    loc temp;
    temp.longitude = op2.longitude + longitude;
    temp.latitude = op2.latitude + latitude;
    return temp;
}
// Overload - for loc.
loc loc::operator-(loc op2)
{
    loc temp;
    // notice order of operands
    temp.longitude = longitude - op2.longitude;
    temp.latitude = latitude - op2.latitude;
    return temp;
}
// Overload asignment for loc.
loc loc::operator=(loc op2)
{
    longitude = op2.longitude;
    latitude = op2.latitude;
    return *this; // i.e., return object that generated call
}
// Overload prefix ++ for loc.
loc loc::operator++()
{
    longitude++;
    latitude++;
    return *this;
}
int main()
{
    loc ob1(10, 20), ob2( 5, 30), ob3(90, 90);
    ob1.show();
    ob2.show();
    ++ob1;
    ob1.show(); // displays 11 21
    ob2 = ++ob1;
```

```
    ob1.show(); // displays 12 22
    ob2.show(); // displays 12 22
    ob1 = ob2 = ob3; // multiple assignment
    ob1.show(); // displays 90 90
    ob2.show(); // displays 90 90
    return 0;
}
```

First, examine the operator–( ) function. Notice the order of the operands in the subtraction. In keeping with the meaning of subtraction, the operand on the right side of the minus sign is subtracted from the operand on the left. Because it is the object on the left that generates the call to the operator–( ) function, op2's data must be subtracted from the data pointed to by this. It is important to remember which operand generates the call to the function. In C++, if the = is not overloaded, a default assignment operation is created automatically for any class you define. The default assignment is simply a member- by-member, bitwise copy. By overloading the =, you can define explicitly what the assignment does relative to a class. In this example, the overloaded = does exactly the same thing as the default, but in other situations, it could perform other operations. Notice that the operator=( ) function returns *this, which is the object that generated the call. This arrangement is necessary if you want to be able to use multiple assignment operations such as this:

```
ob1 = ob2 = ob3; // multiple assignment
```

Now, look at the definition of operator++( ). As you can see, it takes no parameters. Since ++ is a unary operator, its only operand is implicitly passed by using the this pointer.
Notice that both operator=( ) and operator++( ) alter the value of an operand. In the case of assignment, the operand on the left (the one generating the call to the operator=( ) function) is assigned a new value. In the case of the ++, the operand is incremented. As stated previously, although you are free to make these operators do anything you please, it is almost always wisest to stay consistent with their original meanings.

2.1.1 Creating Prefix and Postfix Forms of the Increment and Decrement Operators

In the preceding program, only the prefix form of the increment operator was overloaded. However, Standard C++ allows you to explicitly create separate prefix and postfix versions of the increment or decrement operators. To accomplish this, you must define two versions of the operator++( ) function. One is defined as shown in the foregoing program. The other is declared like this:

```
loc operator++(int x);
```

If the ++ precedes its operand, the operator++( ) function is called. If the ++ follows its operand, the operator++(int x) is called and x has the value zero. The preceding example can be generalized. Here are the general forms for the prefix and postfix ++ and – – operator functions.

```
 // Prefix increment
type operator++( ) {
    // body of prefix operator
}
// Postfix increment
type operator++(int x) {
    // body of postfix operator
}
// Prefix decrement
type operator- -( ) {
    // body of prefix operator
}
// Postfix decrement
type operator- -(int x) {
    // body of postfix operator
}
```

2.1.1.1 Note

You should be careful when working with older C++ programs where the increment and decrement operators are concerned. In older versions of C++, it was not possible to specify separate prefix and postfix versions of an overloaded ++ or – –. The prefixform w as used for both.

2.1.2 Overloading the Shorthand Operators

You can overload any of C++'s "shorthand" operators, such as +=, –=, and the like. For example, this function overloads += relative to loc:

```
loc loc::operator+=(loc op2)
{
    longitude = op2.longitude + longitude;
    latitude = op2.latitude + latitude;
    return *this;
}
```

When overloading one of these operators, keep in mind that you are simply combining an assignment with another type of operation.

2.1.3 Operator Overloading Restrictions

There are some restrictions that apply to operator overloading. You cannot alter the precedence of an operator. You cannot change the number of operands that an operator takes. (You can choose to ignore an

operand, however.) Except for the function call operator (described later), operator functions cannot have default arguments. Finally, these operators cannot be overloaded:
. : : .* ?
As stated, technically you are free to perform any activity inside an operator function. For example, if you want to overload the + operator in such a way that it writes I like C++ 10 times to a disk file, you can do so. However, when you stray significantly from the normal meaning of an operator, you run the risk of dangerously destructuring your program. When someone reading your program sees a statement like Ob1+Ob2, he or she expects something resembling addition to be taking place— not a disk access, for example. Therefore, before decoupling an overloaded operator from its normal meaning, be sure that you have sufficient reason to do so. One good example where decoupling is successful is found in the way C++ overloads the « and » operators for I/O. Although the I/O operations have no relationship to bit shifting, these operators provide a visual "clue" as to their meaning relative to both I/O and bit shifting, and this decoupling works. In general, however, it is best to stay within the context of the expected meaning of an operator when overloading it. Except for the = operator, operator functions are inherited by a derived class. However, a derived class is free to overload any operator (including those overloaded by the base class) it chooses relative to itself.

2.2 Operator Overloading Using a Friend Function

You can overload an operator for a class by using a nonmember function, which is usually a friend of the class. Since a friend function is not a member of the class, it does not have a this pointer. Therefore, an overloaded friend operator function is passed the operands explicitly. This means that a friend function that overloads a binary operator has two parameters, and a friend function that overloads a unary operator has one parameter. When overloading a binary operator using a friend function, the left operand is passed in the first parameter and the right operand is passed in the second parameter.
In this program, the operator+( ) function is made into a friend:

```
#include <iostream>
using namespace std;
class loc {
    int longitude, latitude;
public:
    loc() {} // needed to construct temporaries
    loc(int lg, int lt) {
        longitude = lg;
        latitude = lt;
    }
        void show() {
    cout << longitude << " ";
    cout << latitude << "\n";
```

```
    }
    friend loc operator+(loc op1, loc op2); // now a friend
    loc operator-(loc op2);
    loc operator=(loc op2);
    loc operator++();
};
// Now, + is overloaded using friend function.
loc operator+(loc op1, loc op2)
{
    loc temp;
    temp.longitude = op1.longitude + op2.longitude;
    temp.latitude = op1.latitude + op2.latitude;
    return temp;
}
// Overload - for loc.
loc loc::operator-(loc op2)
{
    loc temp;
    // notice order of operands
    temp.longitude = longitude - op2.longitude;
    temp.latitude = latitude - op2.latitude;
    return temp;
}
// Overload assignment for loc.
loc loc::operator=(loc op2)
{
    longitude = op2.longitude;
    latitude = op2.latitude;
    return *this; // i.e., return object that generated call
}
// Overload ++ for loc.
loc loc::operator++()
{
    longitude++;
    latitude++;
    return *this;
}
int main()
{
    loc ob1(10, 20), ob2( 5, 30);
```

```
    ob1 = ob1 + ob2;
    ob1.show();
    return 0;
}
```

There are some restrictions that apply to friend operator functions. First, you may not overload the =, ( ), [ ], or –> operators by using a friend function. Second, as explained in the next section, when overloading the increment or decrement operators, you will need to use a reference parameter when using a friend function.

2.2.1 Using a Friend to Overload ++ or – –

If you want to use a friend function to overload the increment or decrement operators, you must pass the operand as a reference parameter. This is because friend functions do not have this pointers. Assuming that you stay true to the original meaning of the ++ and – – operators, these operations imply the modification of the operand they operate upon. However, if you overload these operators by using a friend, then the operand is passed by value as a parameter. This means that a friend operator function has no way to modify the operand. Since the friend operator function is not passed a this pointer to the operand, but rather a copy of the operand, no changes made to that parameter affect the operand that generated the call. However, you can remedy this situation by specifying the parameter to the friend operator function as a reference parameter. This causes any changes made to the parameter inside the function to affect the operand that generated the call. For example, this program uses friend functions to overload the prefix versions of ++ and – – operators relative to the loc class:

```
#include <iostream>
using namespace std;
class loc {
    int longitude, latitude;
public:
    loc() {}
    loc(int lg, int lt) {
        longitude = lg;
        latitude = lt;
    }
void show() {
    cout << longitude << " ";
    cout << latitude << "\n";
    }
    loc operator=(loc op2);
    friend loc operator++(loc &op);
    friend loc operator--(loc &op);
```

```
};
// Overload assignment for loc.
loc loc::operator=(loc op2)
{
    longitude = op2.longitude;
    latitude = op2.latitude;
    return *this; // i.e., return object that generated call
}
// Now a friend; use a reference parameter.
loc operator++(loc &op)
{
    op.longitude++;
    op.latitude++;
    return op;
}
// Make op-- a friend; use reference.
loc operator--(loc &op)
{
    op.longitude--;
    op.latitude--;
    return op;
}
int main()
{
    loc ob1(10, 20), ob2;
    ob1.show();
    ++ob1;
    ob1.show(); // displays 11 21
    ob2 = ++ob1;
    ob2.show(); // displays 12 22
    --ob2;
    ob2.show(); // displays 11 21
    return 0;
}
```

If you want to overload the postfix versions of the increment and decrement operators using a friend, simply specify a second, dummy integer parameter. For example, this shows the prototype for the friend, postfix version of the increment operator relative to loc.

```
// friend, postfix version of ++
```

```
friend loc operator++(loc &op, int x);
```

2.2.2 Friend Operator Functions Add Flexibility

In many cases, whether you overload an operator by using a friend or a member function makes no functional difference. In those cases, it is usually best to overload by using member functions. However, there is one situation in which overloading by using a friend increases the flexibility of an overloaded operator. Let's examine this case now.

As you know, when you overload a binary operator by using a member function, the object on the left side of the operator generates the call to the operator function. Further, a pointer to that object is passed in the this pointer. Now, assume some class that defines a member operator+( ) function that adds an object of the class to an integer. Given an object of that class called Ob, the following expression is valid:

Ob + 100 // valid

In this case, Ob generates the call to the overloaded + function, and the addition is performed. But what happens if the expression is written like this?

100 + Ob // invalid

In this case, it is the integer that appears on the left. Since an integer is a built-in type, no operation between an integer and an object of Ob's type is defined. Therefore, the compiler will not compile this expression. As you can imagine, in some applications, having to always position the object on the left could be a significant burden and cause of frustration.

The solution to the preceding problem is to overload addition using a friend, not a member, function. When this is done, both arguments are explicitly passed to the operator function. Therefore, to allow both object+integer and integer+object, simply overload the function twice—one version for each situation. Thus, when you overload an operator by using two friend functions, the object may appear on either the left or right side of the operator.

This program illustrates how friend functions are used to define an operation that involves an object and built-in type:

```
#include <iostream>
using namespace std;
class loc {
    int longitude, latitude;
public:
    loc() {}
    loc(int lg, int lt) {
        longitude = lg;
        latitude = lt;
    }
void show() {
    cout << longitude << " ";
    cout << latitude << "\n";
```

```
    }
    friend loc operator+(loc op1, int op2);
    friend loc operator+(int op1, loc op2);
};
// + is overloaded for loc + int.
loc operator+(loc op1, int op2)
{
    loc temp;
    temp.longitude = op1.longitude + op2;
    temp.latitude = op1.latitude + op2;
    return temp;
}
// + is overloaded for int + loc.
loc operator+(int op1, loc op2)
{
    loc temp;
    temp.longitude = op1 + op2.longitude;
    temp.latitude = op1 + op2.latitude;
    return temp;
}
int main()
{
    loc ob1(10, 20), ob2( 5, 30), ob3(7, 14);
    ob1.show();
    ob2.show();
    ob3.show();
    ob1 = ob2 + 10; // both of these
    ob3 = 10 + ob2; // are valid
    ob1.show();
    ob3.show();
    return 0;
}
```

Chapter 3

Inheritance

Inheritance is one of the cornerstones of OOP because it allows the creation of hierarchical classifications. Using inheritance, you can create a general class that defines traits common to a set of related items. This class may then be inherited by other, more specific classes, each adding only those things that are unique to the inheriting class.
In keeping with standard C++ terminology, a class that is inherited is referred to as a base class. The class that does the inheriting is called the derived class. Further, a derived class can be used as a base class for another derived class. In this way, multiple inheritance is achieved.

3.1 Base-Class Access Control

When a class inherits another, the members of the base class become members of the derived class. Class inheritance uses this general form:

```
    class derived-class-name : access base-class-name {
// body of class
};
```

The access status of the base-class members inside the derived class is determined by access. The base-class access specifier must be either public, private, or protected. If no access specifier is present, the access specifier is private by default if the derived class is a class. If the derived class is a struct, then public is the default in the absence of an explicit access specifier. Let's examine the ramifications of using public or private access. (The protected specifier is examined in the next section.) When the access specifier for a base class is public, all public members of the base become public members of the derived class, and all protected members of the base become protected members of the derived class. In all cases, the base's private elements remain private to the base and are not accessible by members of the derived class. For example, as illustrated in this program, objects of type derived can directly access the public members of base:

```
#include <iostream>
using namespace std;
```

```
class base {
    int i, j;
public:
    void set(int a, int b) { i=a; j=b; }
    void show() { cout << i << " " << j << "\n"; }
};
class derived : public base {
    int k;
public:
    derived(int x) { k=x; }
    void showk() { cout << k << "\n"; }
};
int main()
{
    derived ob(3);
    ob.set(1, 2); // access member of base
    ob.show(); // access member of base
    ob.showk(); // uses member of derived class
    return 0;
}
```

When the base class is inherited by using the private access specifier, all public and protected members of the base class become private members of the derived class. For example, the following program will not even compile because both set( ) and show( ) are now private elements of derived:

```
// This program won't compile.
#include <iostream>
using namespace std;
class base {
    int i, j;
public:
    void set(int a, int b) { i=a; j=b; }
    void show() { cout << i << " " << j << "\n";}
};
// Public elements of base are private in derived.
class derived : private base {
    int k;
public:
    derived(int x) { k=x; }
    void showk() { cout << k << "\n"; }
```

```
};
int main()
{
    derived ob(3);
    ob.set(1, 2); // error, can't access set()
    ob.show(); // error, can't access show()
    return 0;
}
```

When a base class' access specifier is private, public and protected members of the base become private members of the derived class. This means that they are still accessible by members of the derived class but cannot be accessed by parts of your program that are not members of either the base or derived class.

3.2 Inheritance and protected Members

The protected keyword is included in C++ to provide greater flexibility in the inheritance mechanism. When a member of a class is declared as protected, that member is not accessible by other, nonmember elements of the program. With one important exception, access to a protected member is the same as access to a private member—it can be accessed only by other members of its class. The sole exception to this is when a protected member is inherited. In this case, a protected member differs substantially from a private one.
As explained in the preceding section, a private member of a base class is not accessible by other parts of your program, including any derived class. However, protected members behave differently. If the base class is inherited as public, then the base class' protected members become protected members of the derived class and are, therefore, accessible by the derived class. By using protected, you can create class members that are private to their class but that can still be inherited and accessed by a derived class. Here is an example:

```
#include <iostream>
using namespace std;
class base {
protected:
    int i, j; // private to base, but accessible by derived
public:
    void set(int a, int b) { i=a; j=b; }
    void show() { cout << i << " " << j << "\n"; }
};
class derived : public base {
    int k;
public:
    // derived may access base's i and j
```

```
    void setk() { k=i*j; }
    void showk() { cout << k << "\n"; }
};
int main()
{
    derived ob;
    ob.set(2, 3); // OK, known to derived
    ob.show(); // OK, known to derived
    ob.setk();
    ob.showk();
    return 0;
}
```

In this example, because base is inherited by derived as public and because i and j are declared as protected, derived's function setk( ) may access them. If i and j had been declared as private by base, then derived would not have access to them, and the program would not compile. When a derived class is used as a base class for another derived class, any protected member of the initial base class that is inherited (as public) by the first derived class may also be inherited as protected again by a second derived class. For example, this program is correct, and derived2 does indeed have access to i and j.

```
#include <iostream>
using namespace std;
class base {
protected:
    int i, j;
public:
    void set(int a, int b) { i=a; j=b; }
    void show() { cout << i << " " << j << "\n"; }
};
// i and j inherited as protected.
class derived1 : public base {
    int k;
public:
    void setk() { k = i*j; } // legal
    void showk() { cout << k << "\n"; }
};
// i and j inherited indirectly through derived1.
class derived2 : public derived1 {
    int m;
public:
    void setm() { m = i-j; } // legal
```

```
    void showm() { cout << m << "\n"; }
};
int main()
{
    derived1 ob1;
    derived2 ob2;
    ob1.set(2, 3);
    ob1.show();
    ob1.setk();
    ob1.showk();
    ob2.set(3, 4);
    ob2.show();
    ob2.setk();
    ob2.setm();
    ob2.showk();
    ob2.showm();
    return 0;
}
```

If, however, base were inherited as private, then all members of base would become private members of derived1, which means that they would not be accessible by derived2. (However, i and j would still be accessible by derived1.) This situation is illustrated by the following program, which is in error (and won't compile). The comments describe each error:

```
// This program won't compile.
#include <iostream>
using namespace std;
class base {
protected:
    int i, j;
public:
    void set(int a, int b) { i=a; j=b; }
    void show() { cout << i << " " << j << "\n"; }
};
// Now, all elements of base are private in derived1.
class derived1 : private base {
int k;
public:
    // this is legal because i and j are private to derived1
    void setk() { k = i*j; } // OK
```

```
    void showk() { cout << k << "\n"; }
};
// Access to i, j, set(), and show() not inherited.
class derived2 : public derived1 {
    int m;
public:
    // illegal because i and j are private to derived1
    void setm() { m = i-j; } // Error
    void showm() { cout << m << "\n"; }
};
int main()
{
    derived1 ob1;
    derived2 ob2;
    ob1.set(1, 2); // error, can't use set()
    ob1.show(); // error, can't use show()
    ob2.set(3, 4); // error, can't use set()
    ob2.show(); // error, can't use show()
    return 0;
}
```

Even though base is inherited as private by derived1, derived1 still has access to base's public and protected elements. However, it cannot pass along this privilege.

3.3 Protected Base-Class Inheritance

It is possible to inherit a base class as protected. When this is done, all public and protected members of the base class become protected members of the derived class.
For example,

```
#include <iostream>
using namespace std;
class base {
protected:
    int i, j; // private to base, but accessible by derived
public:
    void setij(int a, int b) { i=a; j=b; }
    void showij() { cout << i << " " << j << "\n"; }
};
// Inherit base as protected.
```

```
class derived : protected base{
    int k;
    public:
    // derived may access base's i and j and setij().
    void setk() { setij(10, 12); k = i*j; }
    // may access showij() here
    void showall() { cout << k << " "; showij(); }
};
int main()
{
    derived ob;
    // ob.setij(2, 3); // illegal, setij() is
    // protected member of derived
    ob.setk(); // OK, public member of derived
    ob.showall(); // OK, public member of derived
    // ob.showij(); // illegal, showij() is protected
    // member of derived
    return 0;
}
```

As you can see by reading the comments, even though setij( ) and showij( ) are public members of base, they become protected members of derived when it is inherited using the protected access specifier. This means that they will not be accessible inside main( ).

3.4 Inheriting Multiple Base Classes

It is possible for a derived class to inherit two or more base classes. For example, in this short example, derived inherits both base1 and base2.

```
// An example of multiple base classes.
#include <iostream>
using namespace std;
class base1 {
protected:
    int x;
public:
    void showx() { cout << x << "\n"; }
};
class base2 {
protected:
    int y;
```

```
public:
    void showy() {cout << y << "\n";}
};
// Inherit multiple base classes.
class derived: public base1, public base2 {
public:
    void set(int i, int j) { x=i; y=j; }
};
int main()
{
    derived ob;
    ob.set(10, 20); // provided by derived
    ob.showx(); // from base1
    ob.showy(); // from base2
    return 0;
}
```

As the example illustrates, to inherit more than one base class, use a comma separated list. Further, be sure to use an access-specifier for each base inherited.

3.5 Constructors, Destructors, and Inheritance

There are two major questions that arise relative to constructors and destructors when inheritance is involved. First, when are base-class and derived-class constructors and destructors called? Second, how can parameters be passed to base-class constructors? This section examines these two important topics.

3.5.1 When Constructors and Destructors Are Executed

It is possible for a base class, a derived class, or both to contain constructors and/or destructors. It is important to understand the order in which these functions are executed when an object of a derived class comes into existence and when it goes out of existence. To begin, examine this short program:

```
#include <iostream>
using namespace std;
class base {
public:
    base() { cout << "Constructing base\n"; }
    ~base() { cout << "Destructing base\n"; }
};
class derived: public base {
```

```
public:
    derived() { cout << "Constructing derived\n"; }
    ~derived() { cout << "Destructing derived\n"; }
};
int main()
{
    derived ob;
    // do nothing but construct and destruct ob
    return 0;
}
```

As the comment in main( ) indicates, this program simply constructs and then destroys an object called ob that is of class derived. When executed, this program displays

```
Constructing base
Constructing derived
Destructing derived
Destructing base
```

As you can see, first base's constructor is executed followed by derived's. Next (because ob is immediately destroyed in this program), derived's destructor is called, followed by base's.

The results of the foregoing experiment can be generalized. When an object of a derived class is created, the base class' constructor will be called first, followed by the derived class' constructor. When a derived object is destroyed, its destructor is called first, followed by the base class' destructor. Put differently, constructors are executed in their order of derivation. Destructors are executed in reverse order of derivation.

If you think about it, it makes sense that constructors are executed in order of derivation. Because a base class has no knowledge of any derived class, any initialization it needs to perform is separate from and possibly prerequisite to any initialization performed by the derived class. Therefore, it must be executed first.

Likewise, it is quite sensible that destructors be executed in reverse order of derivation. Because the base class underlies the derived class, the destruction of the base object implies the destruction of the derived object. Therefore, the derived destructor must be called before the object is fully destroyed. In cases of multiple inheritance (that is, where a derived class becomes the base class for another derived class), the general rule applies: Constructors are called in order of derivation, destructors in reverse order. For example, this program

```
 #include <iostream>
using namespace std;
class base {
public:
    base() { cout << "Constructing base\n"; }
    ~base() { cout << "Destructing base\n"; }
```

```
};
class derived1 : public base {
public:
    derived1() { cout << "Constructing derived1\n"; }
    ~derived1() { cout << "Destructing derived1\n"; }
};
class derived2: public derived1 {
public:
    derived2() { cout << "Constructing derived2\n"; }
    ~derived2() { cout << "Destructing derived2\n"; }
};
int main()
{
    derived2 ob;
    // construct and destruct ob
    return 0;
}
```

displays this output:

```
    Constructing base
    Constructing derived1
    Constructing derived2
    Destructing derived2
    Destructing derived1
    Destructing base
```

The same general rule applies in situations involving multiple base classes. For example, this program

```
    #include <iostream>
using namespace std;
class base1 {
public:
    base1() { cout << "Constructing base1\n"; }
    ~base1() { cout << "Destructing base1\n"; }
};
class base2 {
public:
    base2() { cout << "Constructing base2\n"; }
    ~base2() { cout << "Destructing base2\n"; }
```

```
};
class derived: public base1, public base2 {
public:
    derived() { cout << "Constructing derived\n"; }
    ~derived() { cout << "Destructing derived\n"; }
};
int main()
{
    derived ob;
    // construct and destruct ob
    return 0;
}
```

produces this output:

```
Constructing base1
Constructing base2
Constructing derived
Destructing derived
Destructing base2
Destructing base1
```

As you can see, constructors are called in order of derivation, left to right, as specified in derived's inheritance list. Destructors are called in reverse order, right to left. This means that had base2 been specified before base1 in derived's list, as shown here:

```
class derived: public base2, public base1 {
```

then the output of this program would have looked like this:

```
Constructing base2
Constructing base1
Constructing derived
Destructing derived
Destructing base1
Destructing base2
```

3.5.2 Passing Parameters to Base-Class Constructors

So far, none of the preceding examples have included constructors that require arguments. In cases where only the derived class' constructor requires one or more parameters, you simply use the standard parameterized constructor syntax . However, how do you pass arguments to a constructor in a base class? The

answer is to use an expanded form of the derived class's constructor declaration that passes along arguments to one or more base-class constructors. The general form of this expanded derived-class constructor declaration is shown here:

```
derived-constructor(arg-list) : base1(arg-list),
base2(arg-list),
// ...
baseN(arg-list)
{
// body of derived constructor
}
```

Here, base1 through baseN are the names of the base classes inherited by the derived class. Notice that a colon separates the derived class' constructor declaration from the base-class specifications, and that the base-class specifications are separated from each other by commas, in the case of multiple base classes. Consider this program:

```
    #include <iostream>
using namespace std;
class base {
protected:
    int i;
public:
    base(int x) { i=x; cout << "Constructing base\n"; }
    ~base() { cout << "Destructing base\n"; }
};
class derived: public base {
    int j;
public:
    // derived uses x; y is passed along to base.
        derived(int x, int y): base(y)
        { j=x; cout << "Constructing derived\n"; }
        ~derived() { cout << "Destructing derived\n"; }
        void show() { cout << i << " " << j << "\n"; }
};
int main()
{
    derived ob(3, 4);
    ob.show(); // displays 4 3
    return 0;
```

```
}
```

Here, derived's constructor is declared as taking two parameters, x and y. However, derived( ) uses only x; y is passed along to base( ). In general, the derived class' constructor must declare both the parameter(s) that it requires as well as any required by the base class. As the example illustrates, any parameters required by the base class are passed to it in the base class' argument list specified after the colon.
Here is an example that uses multiple base classes:

```
#include <iostream>
using namespace std;
class base1 {
protected:
    int i;
public:
    base1(int x) { i=x; cout << "Constructing base1\n"; }
    ~base1() { cout << "Destructing base1\n"; }
};
class base2 {
protected:
    int k;
public:
    base2(int x) { k=x; cout << "Constructing base2\n"; }
    ~base2() { cout << "Destructing base1\n"; }
};
class derived: public base1, public base2 {
    int j;
public:
    derived(int x, int y, int z): base1(y), base2(z)
    { j=x; cout << "Constructing derived\n"; }
    ~derived() { cout << "Destructing derived\n"; }
    void show() { cout << i << " " << j << " " << k << "\n"; }
};
int main()
{
    derived ob(3, 4, 5);
    ob.show(); // displays 4 3 5
    return 0;
}
```

It is important to understand that arguments to a base-class constructor are passed via arguments to the derived class' constructor. Therefore, even if a derived class' constructor does not use any arguments, it will still need to declare one if the base class requires it. In this situation, the arguments passed to the derived

class are simply passed along to the base. For example, in this program, the derived class' constructor takes no arguments, but base1( ) and base2( ) do:

```
#include <iostream>
using namespace std;
class base1 {
protected:
    int i;
public:
    base1(int x) { i=x; cout << "Constructing base1\n"; }
    ~base1() { cout << "Destructing base1\n"; }
};
class base2 {
protected:
    int k;
public:
    base2(int x) { k=x; cout << "Constructing base2\n"; }
    ~base2() { cout << "Destructing base2\n"; }
};
class derived: public base1, public base2 {
public:
    /* Derived constructor uses no parameter,
    but still must be declared as taking them to
    pass them along to base classes.
    */
    derived(int x, int y): base1(x), base2(y)
        { cout << "Constructing derived\n"; }
    ~derived() { cout << "Destructing derived\n"; }
    void show() { cout << i << " " << k << "\n"; }
};
int main()
{
    derived ob(3, 4);
    ob.show(); // displays 3 4
     return 0;
}
```

A derived class' constructor is free to make use of any and all parameters that it is declared as taking, even if one or more are passed along to a base class. Put differently, passing an argument along to a base class

does not preclude its use by the derived class as well. For example, this fragment is perfectly valid:

```
    class derived: public base {
    int j;
public:
    // derived uses both x and y and then passes them to base.
    derived(int x, int y): base(x, y)
         { j = x*y; cout << "Constructing derived\n"; }
```

One final point to keep in mind when passing arguments to base-class constructors: The argument can consist of any expression valid at the time. This includes function calls and variables. This is in keeping with the fact that C++ allows dynamic initialization.

3.6 Virtual Base Classes

An element of ambiguity can be introduced into a C++ program when multiple base classes are inherited. For example, consider this incorrect program:

```
// This program contains an error and will not compile.
#include <iostream>
using namespace std;
class base {
public:
    int i;
};
// derived1 inherits base.
class derived1 : public base {
public:
    int j;
};
// derived2 inherits base.
class derived2 : public base {
public:
    int k;
};
/* derived3 inherits both derived1 and derived2.
This means that there are two copies of base
in derived3! */
class derived3 : public derived1, public derived2 {
```

```
public:
    int sum;
};
int main()
{
    derived3 ob;
    ob.i = 10; // this is ambiguous, which i???
    ob.j = 20;
    ob.k = 30;
    // i ambiguous here, too
    ob.sum = ob.i + ob.j + ob.k;
    // also ambiguous, which i?
    cout << ob.i << " ";
    cout << ob.j << " " << ob.k << " ";
    cout << ob.sum;
    return 0;
}
```

As the comments in the program indicate, both derived1 and derived2 inherit base. However, derived3 inherits both derived1 and derived2. This means that there are two copies of base present in an object of type derived3. Therefore, in an expression like

ob.i = 10;

which i is being referred to, the one in derived1 or the one in derived2? Because there are two copies of base present in object ob, there are two ob.is! As you can see, the statement is inherently ambiguous. There are two ways to remedy the preceding program. The first is to apply the scope resolution operator to i and manually select one i. For example, this version of the program does compile and run as expected:

```
// This program uses explicit scope resolution to select i.
#include <iostream>
using namespace std;
class base {
public:
    int i;
};
// derived1 inherits base.

class derived1 : public base {
public:
    int j;
```

```
};
// derived2 inherits base.
class derived2 : public base {
public:
    int k;
};
/* derived3 inherits both derived1 and derived2.
This means that there are two copies of base
in derived3! */
class derived3 : public derived1, public derived2 {
public:
    int sum;
};
int main()
{
    derived3 ob;
    ob.derived1::i = 10; // scope resolved, use derived1's i
    ob.j = 20;
    ob.k = 30;
    // scope resolved
    ob.sum = ob.derived1::i + ob.j + ob.k;
    // also resolved here
    cout << ob.derived1::i << " ";
    cout << ob.j << " " << ob.k << " ";
    cout << ob.sum;
    return 0;
}
```

As you can see, because the :: was applied, the program has manually selected derived1's version of base. However, this solution raises a deeper issue: What if only one copy of base is actually required? Is there some way to prevent two copies from being included in derived3? The answer, as you probably have guessed, is yes. This solution is achieved using virtual base classes. When two or more objects are derived from a common base class, you can prevent multiple copies of the base class from being present in an object derived from those objects by declaring the base class as virtual when it is inherited. You accomplish this by preceding the base class' name with the keyword virtual when it is inherited. For example, here is another version of the example program in which derived3 contains only one copy of base:

```
// This program uses virtual base classes.
#include <iostream>
using namespace std;
```

```
class base {
public:
    int i;
};
// derived1 inherits base as virtual.
class derived1 : virtual public base {
public:
    int j;
};
// derived2 inherits base as virtual.
class derived2 : virtual public base {
public:
    int k;
};
/* derived3 inherits both derived1 and derived2.
This time, there is only one copy of base class. */
class derived3 : public derived1, public derived2 {
public:
    int sum;
};
int main()
{
    derived3 ob;
    ob.i = 10; // now unambiguous
    ob.j = 20;
    ob.k = 30;
    // unambiguous
    ob.sum = ob.i + ob.j + ob.k;
    // unambiguous
    cout << ob.i << " ";
    cout << ob.j << " " << ob.k << " ";
    cout << ob.sum;
    return 0;
}
```

As you can see, the keyword virtual precedes the rest of the inherited class' specification. Now that both derived1 and derived2 have inherited base as virtual, any multiple inheritance involving them will cause only one copy of base to be present. Therefore, in derived3, there is only one copy of base and ob.i = 10 is perfectly valid and unambiguous. One further point to keep in mind: Even though both derived1 and derived2 specify base as virtual, base is still present in objects of either type. For example, the following

sequence is perfectly valid:

```
// define a class of type derived1
derived1 myclass;
myclass.i = 88;
```

The only difference between a normal base class and a virtual one is what occurs when an object inherits the base more than once. If virtual base classes are used, then only one base class is present in the object. Otherwise, multiple copies will be found.

Chapter 4

Virtual Functions and Polymorphism

Polymorphism is supported by C++ both at compile time and at run time. As discussed in earlier chapters, compile-time polymorphism is achieved by overloading functions and operators. Run-time polymorphism is accomplished by using inheritance and virtual functions, and these are the topics of this chapter

4.1 Virtual Functions

A virtual function is a member function that is declared within a base class and redefined by a derived class. To create a virtual function, precede the function's declaration in the base class with the keyword virtual. When a class containing a virtual function is inherited, the derived class redefines the virtual function to fit its own needs. In essence, virtual functions implement the "one interface, multiple methods" philosophy that underlies polymorphism. The virtual function within the base class defines the form of the interface to that function. Each redefinition of the virtual function by a derived class implements its operation as it relates specifically to the derived class. That is, the redefinition creates a specific method.
When accessed "normally," virtual functions behave just like any other type of class member function. However, what makes virtual functions important and capable of supporting run-time polymorphism is how they behave when accessed via a pointer. As discussed in Chapter 13, a base-class pointer can be used to point to an object of any class derived from that base. When a base pointer points to a derived object that contains a virtual function, C++ determines which version of that function to call based upon the type of object pointed to by the pointer. And this determination is made at run time. Thus, when different objects are pointed to, different versions of the virtual function are executed. The same effect applies to base-class references.
To begin, examine this short example:

```
#include <iostream>
using namespace std;
class base {
public:
    virtual void vfunc() {
    cout << "This is base's vfunc().\n";
```

```
    }
};
class derived1 : public base {
public:
    void vfunc() {
         cout << "This is derived1's vfunc().\n";
    }
};
class derived2 : public base {
public:
    void vfunc() {
         cout << "This is derived2's vfunc().\n";
    }
};
int main()
{
    base *p, b;
    derived1 d1;
    derived2 d2;
    // point to base
    p = &b;
    p->vfunc(); // access base's vfunc()
    // point to derived1
    p = &d1;
    p->vfunc(); // access derived1's vfunc()
    // point to derived2
    p = &d2;
    p->vfunc(); // access derived2's vfunc()
    return 0;
}
```

This program displays the following:

```
This is base's vfunc().
This is derived1's vfunc().
This is derived2's vfunc().
```

As the program illustrates, inside base, the virtual function vfunc( ) is declared. Notice that the keyword virtual precedes the rest of the function declaration. When vfunc( ) is redefined by derived1 and derived2, the keyword virtual is not needed. (However, it is not an error to include it when redefining a virtual function inside a derived class; it's just not needed.) In this program, base is inherited by both derived1 and

derived2. Inside each class definition, vfunc( ) is redefined relative to that class. Inside main( ), four variables are declared:

```
Name        Type
p           base class pointer
b           object of base
d1          object of derived1
d2          object of derived2
```

Next, p is assigned the address of b, and vfunc( ) is called via p. Since p is pointing to an object of type base, that version of vfunc( ) is executed. Next, p is set to the address of d1, and again vfunc( ) is called by using p. This time p points to an object of type derived1. This causes derived1::vfunc( ) to be executed. Finally, p is assigned the address of d2, and p >vfunc( ) causes the version of vfunc( ) redefined inside derived2 to be executed. The key point here is that the kind of object to which p points determines which version of vfunc( ) is executed. Further, this determination is made at run time, and this process forms the basis for run-time polymorphism.
Although you can call a virtual function in the "normal" manner by using an object's name and the dot operator, it is only when access is through a base-class pointer (or reference) that run-time polymorphism is achieved. For example, assuming the preceding example, this is syntactically valid:

```
d2.vfunc(); // calls derived2's vfunc()
```

Although calling a virtual function in this manner is not wrong, it simply does not take advantage of the virtual nature of vfunc( ). At first glance, the redefinition of a virtual function by a derived class appears similar to function overloading. However, this is not the case, and the term overloading is not applied to virtual function redefinition because several differences exist. Perhaps the most important is that the prototype for a redefined virtual function must match exactly the prototype specified in the base class. This differs from overloading a normal function, in which return types and the number and type of parameters may differ. (In fact, when you overload a function, either the number or the type of the parameters must differ! It is through these differences that C++ can select the correct version of an overloaded function.) However, when a virtual function is redefined, all aspects of its prototype must be the same. If you change the prototype when you attempt to redefine a virtual function, the function will simply be considered overloaded by the C++ compiler, and its virtual nature will be lost. Another important restriction is that virtual functions must be nonstatic members of the classes of which they are part. They cannot be friends. Finally, constructor functions cannot be virtual, but destructor functions can. Because of the restrictions and differences between function overloading and virtual function redefinition, the term overriding is used to describe virtual function.

4.1.1 Calling a Virtual Function Through a Base Class Reference

In the preceding example, a virtual function was called through a base-class pointer, but the polymorphic nature of a virtual function is also available when called through a base-class reference. As explained in Chapter 13, a reference is an implicit pointer. Thus, a base-class reference can be used to refer to an object of the base class or any object derived from that base. When a virtual function is called through a base-class reference, the version of the function executed is determined by the object being referred to at the time of the call. The most common situation in which a virtual function is invoked through a base class reference is when the reference is a function parameter. For example, consider the following variation on the preceding program.

```
/* Here, a base class reference is used to access
a virtual function. */
#include <iostream>
using namespace std;
class base {
public:
    virtual void vfunc() {
          cout << "This is base's vfunc().\n";
    }
};
class derived1 : public base {
public:
    void vfunc() {
         cout << "This is derived1's vfunc().\n";
    }
};
class derived2 : public base {
public:
    void vfunc() {
         cout << "This is derived2's vfunc().\n";
    }
};
// Use a base class reference parameter.
void f(base &r) {
        r.vfunc();
}
int main()
{
    base b;
```

```
    derived1 d1;
    derived2 d2;
    f(b); // pass a base object to f()
    f(d1); // pass a derived1 object to f()
    f(d2); // pass a derived2 object to f()
    return 0;
}
```

This program produces the same output as its preceding version. In this example, the function f( ) defines a reference parameter of type base. Inside main( ), the function is called using objects of type base, derived1, and derived2. Inside f( ), the specific version of vfunc( ) that is called is determined by the type of object being referenced when the function is called.
For the sake of simplicity, the rest of the examples in this chapter will call virtual functions through base-class pointers, but the effects are same for base-class references.

4.2 The Virtual Attribute Is Inherited

When a virtual function is inherited, its virtual nature is also inherited. This means that when a derived class that has inherited a virtual function is itself used as a base class for another derived class, the virtual function can still be overridden. Put differently, no matter how many times a virtual function is inherited, it remains virtual. For example, consider this program:

```
#include <iostream>
using namespace std;
class base {
public:
    virtual void vfunc() {
       cout << "This is base's vfunc().\n";
    }
};
class derived1 : public base {
public:
    void vfunc() {
            cout << "This is derived1's vfunc().\n";
    }
};
/* derived2 inherits virtual function vfunc()
from derived1. */
class derived2 : public derived1 {
public:
```

```
    // vfunc() is still virtual
    void vfunc() {
         cout << "This is derived2's vfunc().\n";
    }
};
int main()
{
    base *p, b;
    derived1 d1;
    derived2 d2;
    // point to base
    p = &b;
    p->vfunc(); // access base's vfunc()
    // point to derived1
    p = &d1;
    p->vfunc(); // access derived1's vfunc()
    // point to derived2
    p = &d2;
    p->vfunc(); // access derived2's vfunc()
    return 0;
}
```

As expected, the preceding program displays this output:

```
This is base's vfunc().
This is derived1's vfunc().
This is derived2's vfunc().
```

In this case, derived2 inherits derived1 rather than base, but vfunc( ) is still virtual.

4.3 Virtual Functions Are Hierarchical

As explained, when a function is declared as virtual by a base class, it may be overridden by a derived class. However, the function does not have to be overridden. When a derived class fails to override a virtual function, then when an object of that derived class accesses that function, the function defined by the base class is used. For example, consider this program in which derived2 does not override vfunc( ):

```
#include <iostream>
using namespace std;
class base {
```

```
public:
    virtual void vfunc() {
        cout << "This is base's vfunc().\n";
    }
};
class derived1 : public base {
public:
    void vfunc() {
        cout << "This is derived1's vfunc().\n";
    }
};
class derived2 : public base {
public:
    // vfunc() not overridden by derived2, base's is used
};
int main()
{
    base *p, b;
    derived1 d1;
    derived2 d2;
    // point to base
    p = &b;
    p->vfunc(); // access base's vfunc()
    // point to derived1
    p = &d1;
    p->vfunc(); // access derived1's vfunc()
    // point to derived2
    p = &d2;
    p->vfunc(); // use base's vfunc()
    return 0;
}
```

The program produces this output:

```
This is base's vfunc().
This is derived1's vfunc().
This is base's vfunc().
```

Because derived2 does not override vfunc( ), the function defined by base is used when vfunc( ) is referenced relative to objects of type derived2. The preceding program illustrates a special case of a more general rule. Because inheritance is hierarchical in C++, it makes sense that virtual functions are also hierarchical. This

means that when a derived class fails to override a virtual function, the first redefinition found in reverse order of derivation is used. For example, in the following program, derived2 is derived from derived1, which is derived from base. However, derived2 does not override vfunc( ). This means that, relative to derived2,the closest version of vfunc( ) is in derived1. Therefore, it is derived1::vfunc( ) that is used when an object of derived2 attempts to call vfunc( ).

```
#include <iostream>
using namespace std;
class base {
public:
    virtual void vfunc() {
         cout << "This is base's vfunc().\n";
    }
};
class derived1 : public base {
public:
    void vfunc() {
        cout << "This is derived1's vfunc().\n";
    }
};
class derived2 : public derived1 {
public:
    /* vfunc() not overridden by derived2.
    In this case, since derived2 is derived from
    derived1, derived1's vfunc() is used.
    */
};
int main()
{
    base *p, b;
    derived1 d1;
    derived2 d2;
    // point to base
    p = &b;
    p->vfunc(); // access base's vfunc()
    // point to derived1
    p = &d1;
    p->vfunc(); // access derived1's vfunc()
    // point to derived2
    p = &d2;
```

```
    p->vfunc(); // use derived1's vfunc()
    return 0;
}
```

4.4 Pure Virtual Functions

As the examples in the preceding section illustrate, when a virtual function is not redefined by a derived class, the version defined in the base class will be used. However, in many situations there can be no meaningful definition of a virtual function within a base class. For example, a base class may not be able to define an object sufficiently to allow a base-class virtual function to be created. Further, in some situations you will want to ensure that all derived classes override a virtual function. To handle these two cases, C++ supports the pure virtual function.A pure virtual function is a virtual function that has no definition within the base class. To declare a pure virtual function, use this general form:

```
    virtual type func-name(parameter-list) = 0;
```

When a virtual function is made pure, any derived class must provide its own definition. If the derived class fails to override the pure virtual function, a compile-time error will result. The following program contains a simple example of a pure virtual function. The base class, number, contains an integer called val, the function setval( ), and the pure virtual function show( ). The derived classes hextype, dectype, and octtype inherit number and redefine show( ) so that it outputs the value of val in each respective number base (that is, hexadecimal, decimal, or octal).

```
#include <iostream>
using namespace std;
class number {
protected:
    int val;
public:
    void setval(int i) { val = i; }
    // show() is a pure virtual function
    virtual void show() = 0;
};
class hextype : public number {
public:
    void show() {
         cout << hex << val << "\n";
    }
};
class dectype: public number {
```

```
public:
    void show() {
        cout << val << "\n";
        }
};
class octtype: public number {
public:
    void show() {
         cout << oct << val << "\n";
    }
};
int main()
{
    dectype d;
    hextype h;
    octtype o;
    d.setval(20);
    d.show(); // displays 20 - decimal
    h.setval(20);
    h.show(); // displays 14 - hexadecimal
    454 C++: The Complete Reference
    o.setval(20);
    o.show(); // displays 24 - octal
    return 0;
}
```

Although this example is quite simple, it illustrates how a base class may not be able to meaningfully define a virtual function. In this case, number simply provides the common interface for the derived types to use. There is no reason to define show( ) inside number since the base of the number is undefined. Of course, you can always create a placeholder definition of a virtual function. However, making show( ) pure also ensures that all derived classes will indeed redefine it to meet their own needs. Keep in mind that when a virtual function is declared as pure, all derived classes must override it. If a derived class fails to do this, a compile-time error will result.

4.4.1 Abstract Classes

A class that contains at least one pure virtual function is said to be abstract. Because an abstract class contains one or more functions for which there is no definition (that is,a pure virtual function), no objects of an abstract class may be created. Instead, an abstract class constitutes an incomplete type that is used as a foundation for derived classes. Although you cannot create objects of an abstract class, you can create

pointers and references to an abstract class. This allows abstract classes to support run-time polymorphism, which relies upon base-class pointers and references to select the proper virtual function.

4.5 Using Virtual Functions

One of the central aspects of object-oriented programming is the principle of "one interface, multiple methods." This means that a general class of actions can be defined,th e interface to which is constant, with each derivation defining its own specific operations. In concrete C++ terms, a base class can be used to define the nature of the interface to a general class. Each derived class then implements the specific operations as they relate to the type of data used by the derived type. One of the most powerful and flexible ways to implement the "one interface, multiple methods" approach is to use virtual functions, abstract classes, and run-time polymorphism. Using these features, you create a class hierarchy that moves from general to specific (base to derived). Following this philosophy, you define all common features and interfaces in a base class. In cases where certain actions can be implemented only by the derived class, use a virtual function. In essence, in the base class you create and define everything you can that relates to the general case. The derived class fills in the specific details. Following is a simple example that illustrates the value of the "one interface, multiple methods" philosophy. A class hierarchy is created that performs conversions from one system of units to another. (For example, liters to gallons.) The base class convert declares two variables, val1 and val2, which hold the initial and converted values, respectively. It also defines the functions getinit( ) and getconv( ), which return the initial value and the converted value. These elements of convert are fixed and applicable to all derived classes that will inherit convert. However, the function that will actually perform the conversion, compute( ), is a pure virtual function that must be defined by the classes derived from convert. The specific nature of compute( )will be determined by what type of conversion is taking place.

```
// Virtual function practical example.
#include <iostream>
using namespace std;
class convert {
protected:
    double val1; // initial value
    double val2; // converted value
public:
    convert(double i) {
        val1 = i;
    }
    double getconv() { return val2; }
    double getinit() { return val1; }
    virtual void compute() = 0;
};
// Liters to gallons.
```

```
class l_to_g : public convert {
public:
    l_to_g(double i) : convert(i) { }
    void compute() {
         val2 = val1 / 3.7854;
    }
};
// Fahrenheit to Celsius
class f_to_c : public convert {
public:
    f_to_c(double i) : convert(i) { }
    void compute() {
         val2 = (val1-32) / 1.8;
    }
};
int main()
{
    convert *p; // pointer to base class
    l_to_g lgob(4);
    f_to_c fcob(70);
    // use virtual function mechanism to convert
    p = &lgob;
    cout << p->getinit() << " liters is ";
    p->compute();
    cout << p->getconv() << " gallons\n"; // l_to_g
    p = &fcob;
    cout << p->getinit() << " in Fahrenheit is ";
    p->compute();
    cout << p->getconv() << " Celsius\n"; // f_to_c
    return 0;
}
```

The preceding program creates two derived classes from convert, called l_to_g and f_to_c. These classes perform the conversions of liters to gallons and Fahrenheit to Celsius, respectively. Each derived class overrides compute( ) in its own way to perform the desired conversion. However, even though the actual conversion (that is, method) differs between l_to_g and f_to_c, the interface remains constant.One of the benefits of derived classes and virtual functions is that handling a new case is a very easy matter. For example, assuming the preceding program, you can add a conversion from feet to meters by including this class:

```
// Feet to meters
```

```
class f_to_m : public convert {
public:
    f_to_m(double i) : convert(i) { }
    void compute() {
         val2 = val1 / 3.28;
    }
};
```

An important use of abstract classes and virtual functions is in class libraries. You can create a generic, extensible class library that will be used by other programmers. Another programmer will inherit your general class, which defines the interface and all elements common to all classes derived from it, and will add those functions specific to the derived class. By creating class libraries, you are able to create and control the interface of a general class while still letting other programmers adapt it to their specific needs. One final point: The base class converts is an example of an abstract class. The virtual function compute( ) is not defined within convert because no meaningful definition can be provided. The class convert simply does not contain sufficient information for compute( ) to be defined. It is only when convert is inherited by a derived class that a complete type is created.

Reference

1. Brett, M., Gary, P., & David, W. (2006). Head First Object-Oriented Analysis and Design. O'Reilly, 338, 349.
2. Prata, S. (2002). C++ primer plus. Sams Publishing.
3. Schildt, H., Ytreberg, F. M., McKay, S. R., & Christian, W. (1996). C++: The complete reference. Computers in Physics, 10(6), 549-550.
4. Balagurusamy, E. (2021). Object oriented programming with C++.
5. Lafore, R. (1992). Object-oriented programming in Microsoft C++. Waite Group Press.
6. Kirch-Prinz, U., & Prinz, P. (2002). A complete guide to programming in C++. Jones & Bartlett Learning.
7. Meyers, S. (2014). Effective modern C++: 42 specific ways to improve your use of C++ 11 and C++ 14. " O'Reilly Media, Inc.".
8. Stroustrup, B. (2013). The C++ programming language. Pearson Education.
9. Rao, S. (2022). C++ in One Hour a Day, Sams Teach Yourself. Sams Publishing.
10. Loudon, K. (2008). C++ Pocket Reference: C++ Syntax and Fundamentals. " O'Reilly Media, Inc.".