Virtual Functions

CHAPTER

11

IN THIS CHAPTER

Now that we understand something about pointers, we can delve into more advanced C++ topics. This chapter covers a rather loosely related collection of such subjects: virtual functions, friend functions, static functions, the overloaded = operator, the overloaded copy constructor, and the `this` pointer. These are advanced features; they are not necessary for every C++ program, especially very short ones. However, they are widely used, and are essential for most full-size programs. Virtual functions in particular are essential for polymorphism, one of the cornerstones of object-oriented programming.

Virtual Functions

Virtual means *existing in appearance but not in reality*. When virtual functions are used, a program that appears to be calling a function of one class may in reality be calling a function of a different class. Why are virtual functions needed? Suppose you have a number of objects of different classes but you want to put them all in an array and perform a particular operation on them using the same function call. For example, suppose a graphics program includes several different shapes: a triangle, a ball, a square, and so on, as in the MULTSHAP program in Chapter 9, "Inheritance." Each of these classes has a member function `draw()` that causes the object to be drawn on the screen.

Now suppose you plan to make a picture by grouping a number of these elements together, and you want to draw the picture in a convenient way. One approach is to create an array that holds pointers to all the different objects in the picture. The array might be defined like this:

```
shape* ptrarr[100];  // array of 100 pointers to shapes
```

If you insert pointers to all the shapes into this array, you can then draw an entire picture using a simple loop:

```
for(int j=0; j<N; j++)
   ptrarr[j]->draw();
```

This is an amazing capability: Completely different functions are executed by the same function call. If the pointer in `ptrarr` points to a ball, the function that draws a ball is called; if it points to a triangle, the triangle-drawing function is called. This is called *polymorphism*, which means *different forms*. The functions have the same appearance, the `draw()` expression, but different actual functions are called, depending on the contents of `ptrarr[j]`. Polymorphism is one of the key features of object-oriented programming, after classes and inheritance.

For the polymorphic approach to work, several conditions must be met. First, all the different classes of shapes, such as balls and triangles, must be descended from a single base class (called `shape` in MULTSHAP). Second, the `draw()` function must be declared to be `virtual` in the base class.

This is all rather abstract, so let's start with some short programs that show parts of the situation, and put everything together later.

Normal Member Functions Accessed with Pointers

Our first example shows what happens when a base class and derived classes all have functions with the same name, and you access these functions using pointers but without using virtual functions. Here's the listing for NOTVIRT:

```
// notvirt.cpp
// normal functions accessed from pointer
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class Base                          //base class
   {
   public:
      void show()                   //normal function
         { cout << "Base\n"; }
   };
////////////////////////////////////////////////////////////////
class Derv1 : public Base           //derived class 1
   {
   public:
      void show()
         { cout << "Derv1\n"; }
   };
////////////////////////////////////////////////////////////////
class Derv2 : public Base           //derived class 2
   {
   public:
      void show()
         { cout << "Derv2\n"; }
   };
////////////////////////////////////////////////////////////////
int main()
   {
   Derv1 dv1;             //object of derived class 1
   Derv2 dv2;             //object of derived class 2
   Base* ptr;             //pointer to base class

   ptr = &dv1;            //put address of dv1 in pointer
   ptr->show();           //execute show()
```

```
ptr = &dv2;          //put address of dv2 in pointer
ptr->show();         //execute show()
return 0;
}
```

The `Derv1` and `Derv2` classes are derived from class `Base`. Each of these three classes has a member function `show()`. In `main()` we create objects of class `Derv1` and `Derv2`, and a pointer to class `Base`. Then we put the address of a derived class object in the base class pointer in the line

```
ptr = &dv1;  // derived class address in base class pointer
```

But wait—how can we get away with this? Doesn't the compiler complain that we're assigning an address of one type (`Derv1`) to a pointer of another (`Base`)? On the contrary, the compiler is perfectly happy, because type checking has been relaxed in this situation, for reasons that will become apparent soon. The rule is that pointers to objects of a derived class are type-compatible with pointers to objects of the base class.

Now the question is, when you execute the line

```
ptr->show();
```

what function is called? Is it `Base::show()` or `Derv1::show()`? Again, in the last two lines of NOTVIRT we put the address of an object of class `Derv2` in the pointer, and again execute

```
ptr->show();
```

Which of the `show()` functions is called here? The output from the program answers these questions:

```
Base
Base
```

As you can see, the function in the base class is always executed. The compiler ignores the *contents* of the pointer `ptr` and chooses the member function that matches the *type* of the pointer, as shown in Figure 11.1.

Sometimes this is what we want, but it doesn't solve the problem posed at the beginning of this section: accessing objects of different classes using the same statement.

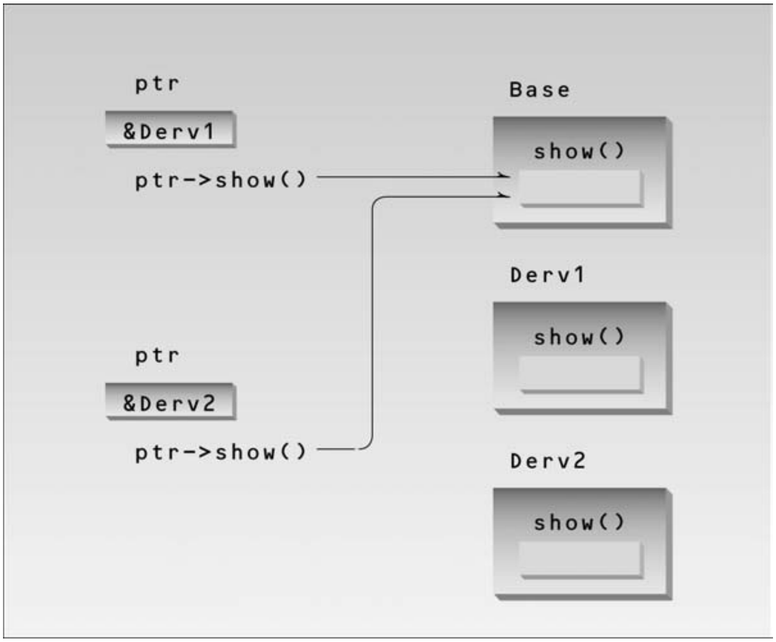

FIGURE 11.1
Nonvirtual pointer access.

Virtual Member Functions Accessed with Pointers

Let's make a single change in our program: We'll place the keyword *virtual* in front of the declarator for the `show()` function in the base class. Here's the listing for the resulting program, VIRT:

```
// virt.cpp
// virtual functions accessed from pointer
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class Base                        //base class
   {
   public:
      virtual void show()         //virtual function
         { cout << "Base\n"; }
   };
////////////////////////////////////////////////////////////////
class Derv1 : public Base         //derived class 1
   {
```

```
   public:
      void show()
         { cout << "Derv1\n"; }
   };
////////////////////////////////////////////////////////////////
class Derv2 : public Base         //derived class 2
   {
   public:
      void show()
         { cout << "Derv2\n"; }
   };
////////////////////////////////////////////////////////////////
int main()
   {
   Derv1 dv1;            //object of derived class 1
   Derv2 dv2;            //object of derived class 2
   Base* ptr;            //pointer to base class

   ptr = &dv1;           //put address of dv1 in pointer
   ptr->show();          //execute show()

   ptr = &dv2;           //put address of dv2 in pointer
   ptr->show();          //execute show()
   return 0;
   }
```

The output of this program is

```
Derv1
Derv2
```

Now, as you can see, the member functions of the derived classes, not the base class, are executed. We change the contents of `ptr` from the address of `Derv1` to that of `Derv2`, and the particular instance of `show()` that is executed also changes. So the same function call

```
ptr->show();
```

executes different functions, depending on the contents of `ptr`. The rule is that the compiler selects the function based on the *contents* of the pointer `ptr`, not on the *type* of the pointer, as in NOTVIRT. This is shown in Figure 11.2.

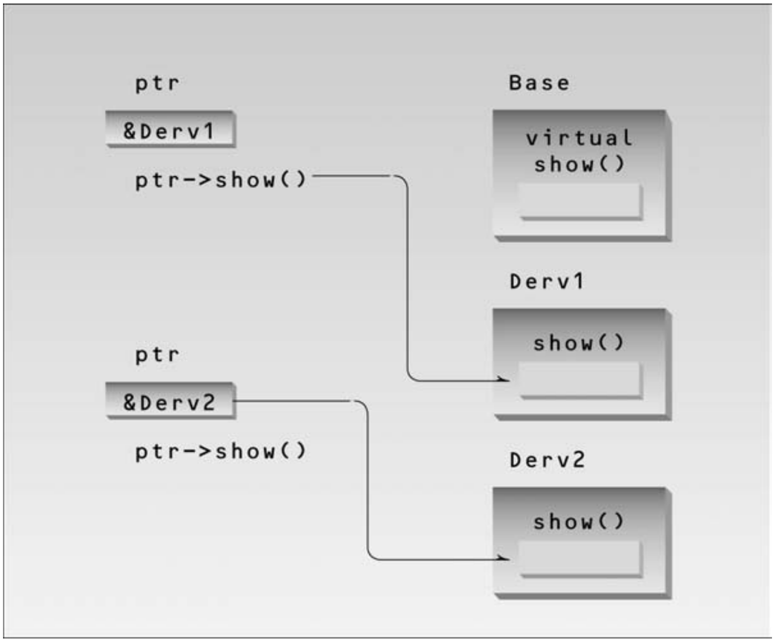

FIGURE 11.2
Virtual pointer access.

Late Binding

The astute reader may wonder how the compiler knows what function to compile. In NOTVIRT the compiler has no problem with the expression

```
ptr->show();
```

It always compiles a call to the `show()` function in the base class. But in VIRT the compiler doesn't know what class the contents of `ptr` may contain. It could be the address of an object of the `Derv1` class or of the `Derv2` class. Which version of `draw()` does the compiler call? In fact the compiler doesn't know what to do, so it arranges for the decision to be deferred until the program is running. At runtime, when it is known what class is pointed to by `ptr`, the appropriate version of `draw` will be called. This is called *late binding* or *dynamic binding*. (Choosing functions in the normal way, during compilation, is called *early binding* or *static binding*.) Late binding requires some overhead but provides increased power and flexibility.

We'll put these ideas to use in a moment, but first let's consider a refinement to the idea of virtual functions.

Abstract Classes and Pure Virtual Functions

Think of the `shape` class in the `multshap` program in Chapter 9. We'll never make an object of the `shape` class; we'll only make specific shapes such as circles and triangles. When we will never want to instantiate objects of a base class, we call it an *abstract class*. Such a class exists only to act as a parent of derived classes that will be used to instantiate objects. It may also provide an interface for the class hierarchy.

How can we make it clear to someone using our family of classes that we don't want anyone to instantiate objects of the base class? We could just say this in the documentation, and count on the users of the class to remember it, but of course it's much better to write our classes so that such instantiation is impossible. How can we can do that? By placing at least one *pure virtual function* in the base class. A pure virtual function is one with the expression `=0` added to the declaration. This is shown in the VIRTPURE example.

```
// virtpure.cpp
// pure virtual function
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class Base                          //base class
   {
   public:
      virtual void show() = 0;    //pure virtual function
   };
////////////////////////////////////////////////////////////////
class Derv1 : public Base         //derived class 1
   {
   public:
      void show()
         { cout << "Derv1\n"; }
   };
////////////////////////////////////////////////////////////////
class Derv2 : public Base         //derived class 2
   {
   public:
      void show()
         { cout << "Derv2\n"; }
   };
////////////////////////////////////////////////////////////////
int main()
   {
// Base bad;              //can't make object from abstract class
   Base* arr[2];          //array of pointers to base class
   Derv1 dv1;             //object of derived class 1
```

```
Derv2 dv2;             //object of derived class 2

arr[0] = &dv1;         //put address of dv1 in array
arr[1] = &dv2;         //put address of dv2 in array

arr[0]->show();        //execute show() in both objects
arr[1]->show();
return 0;
}
```

Here the virtual function `show()` is declared as

```
virtual void show() = 0;  // pure virtual function
```

The equal sign here has nothing to do with assignment; the value 0 is not assigned to anything. The `=0` syntax is simply how we tell the compiler that a virtual function will be pure. Now if in `main()` you attempt to create objects of class `Base`, the compiler will complain that you're trying to instantiate an object of an abstract class. It will also tell you the name of the pure virtual function that makes it an abstract class. Notice that, although this is only a declaration, you never need to write a definition of the base class `show()`, although you can if you need to.

Once you've placed a pure virtual function in the base class, you must override it in all the derived classes from which you want to instantiate objects. If a class doesn't override the pure virtual function, it becomes an abstract class itself, and you can't instantiate objects from it (although you might from classes derived from it). For consistency, you may want to make all the virtual functions in the base class pure.

As you can see, we've made another, unrelated, change in VIRTPURE: The addresses of the member functions are stored in an array of pointers and accessed using array elements. This works in just the same way as using a single pointer. The output of VIRTPURE is the same as VIRT:

```
Derv1
Derv2
```

Virtual Functions and the `person` Class

Now that we understand some of the mechanics of virtual functions, let's look at a situation where it makes sense to use them. Our example is an extension of the PTROBJ and PERSORT examples from Chapter 10, "Pointers." It uses the same `person` class, but adds two derived classes, `student` and `professor`. These derived classes each contain a function called `isOutstanding()`. This function makes it easy for the school administrators to create a list of outstanding students and professors for the venerable Awards Day ceremony. Here's the listing for VIRTPERS:

```
// virtpers.cpp
// virtual functions with person class
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class person                         //person class
   {
   protected:
      char name[40];
   public:
      void getName()
         { cout << "   Enter name: "; cin >> name; }
      void putName()
         { cout << "Name is: " << name << endl; }
      virtual void getData() = 0;        //pure virtual func
      virtual bool isOutstanding() = 0;  //pure virtual func
   };
////////////////////////////////////////////////////////////////
class student : public person        //student class
   {
   private:
      float gpa;              //grade point average
   public:
      void getData()          //get student data from user
         {
         person::getName();
         cout << "   Enter student's GPA: "; cin >> gpa;
         }
      bool isOutstanding()
         { return (gpa > 3.5) ? true : false; }
   };
////////////////////////////////////////////////////////////////
class professor : public person      //professor class
   {
   private:
      int numPubs;             //number of papers published
   public:
      void getData()           //get professor data from user
         {
         person::getName();
         cout << "   Enter number of professor's publications: ";
         cin >> numPubs;
         }
      bool isOutstanding()
         { return (numPubs > 100) ? true : false; }
   };
```

```
////////////////////////////////////////////////////////////////
int main()
   {
   person* persPtr[100];      //array of pointers to persons
   int n = 0;                 //number of persons on list
   char choice;

   do {
      cout << "Enter student or professor (s/p): ";
      cin >> choice;
      if(choice=='s')                  //put new student
         persPtr[n] = new student;     //   in array
      else                             //put new professor
         persPtr[n] = new professor;   //   in array
      persPtr[n++]->getData();         //get data for person
      cout << "   Enter another (y/n)? ";  //do another person?
      cin >> choice;
      } while( choice=='y' );          //cycle until not 'y'

   for(int j=0; j<n; j++)              //print names of all
      {                                //persons, and
      persPtr[j]->putName();           //say if outstanding
      if( persPtr[j]->isOutstanding() )
         cout << "   This person is outstanding\n";
      }
   return 0;
   }  //end main()
```

The Classes

The `person` class is an abstract class because it contains the pure virtual functions `getData()` and `isOutstanding()`. No `person` objects can ever be created. This class exists only to be the base class for the `student` and `professor` classes. The `student` and `professor` classes add new data items to the base class. The `student` class contains a variable `gpa` of type `float`, which represents the student's grade point average (GPA). The `professor` class contains a variable `numPubs`, of type `int`, which represents the number of scholarly publications the professor has published. Students with a GPA of over 3.5 and professors who have published more than 100 papers are considered outstanding. (We'll refrain from comment on the desirability of these criteria for judging educational excellence.)

The `isOutstanding()` Function

The `isOutstanding()` function is declared as a pure virtual function in `person`. In the `student` class this function returns a `bool` `true` if the student's GPA is greater than 3.5, and `false` otherwise. In `professor` it returns `true` if the professor's `numPubs` variable is greater than 100.

The `getData()` function asks the user for the GPA for a `student`, but for the number of publications for a `professor`.

The `main()` Program

In `main()` we first let the user enter a number of student and teacher names. For students, the program also asks for the GPA, and for professors it asks for the number of publications. When the user is finished, the program prints out the names of all the students and professors, noting those who are outstanding. Here's some sample interaction:

```
Enter student or professor (s/p): s
   Enter name: Timmy
   Enter student's GPA: 1.2
   Enter another (y/n)? y
Enter student or professor (s/p): s
   Enter name: Brenda
   Enter student's GPA: 3.9
   Enter another (y/n)? y
Enter student or professor (s/p): s
   Enter name: Sandy
   Enter student's GPA: 2.4
   Enter another (y/n)? y
Enter student or professor (s/p): p
   Enter name: Shipley
   Enter number of professor's publications: 714
   Enter another (y/n)? y
Enter student or professor (s/p): p
   Enter name: Wainright
   Enter number of professor's publications: 13
   Enter another (y/n)? n

Name is: Timmy
Name is: Brenda
   This person is outstanding
Name is: Sandy
Name is: Shipley
   This person is outstanding
Name is: Wainright
```

Virtual Functions in a Graphics Example

Let's try another example of virtual functions. This one is a graphics example derived from the MULTSHAP program in Chapter 9, "Inheritance." As we noted at the beginning of this section, you may want to draw a number of shapes using the same statement. The VIRTSHAP program does this. Remember that you must build this program with the appropriate console graphics file, as described in Appendix E, "Console Graphics Lite."

```
// virtshap.cpp
// virtual functions with shapes
#include <iostream>
using namespace std;
#include "msoftcon.h"         //for graphics functions
////////////////////////////////////////////////////////////////
class shape                   //base class
   {
   protected:
      int xCo, yCo;           //coordinates of center
      color fillcolor;        //color
      fstyle fillstyle;       //fill pattern
   public:                    //no-arg constructor
      shape() : xCo(0), yCo(0), fillcolor(cWHITE),
                                          fillstyle(SOLID_FILL)
         {  }                 //4-arg constructor
      shape(int x, int y, color fc, fstyle fs) :
                    xCo(x), yCo(y), fillcolor(fc), fillstyle(fs)
         {  }
      virtual void draw()=0   //pure virtual draw function
         {
         set_color(fillcolor);
         set_fill_style(fillstyle);
         }
   };
////////////////////////////////////////////////////////////////
class ball : public shape
   {
   private:
      int radius;             //(xCo, yCo) is center
   public:
      ball() : shape()        //no-arg constr
         {  }
                              //5-arg constructor
      ball(int x, int y, int r, color fc, fstyle fs)
                    : shape(x, y, fc, fs), radius(r)
         {  }
      void draw()             //draw the ball
         {
         shape::draw();
         draw_circle(xCo, yCo, radius);
         }
   };
////////////////////////////////////////////////////////////////
class rect : public shape
   {
```

```
   private:
      int width, height;      //(xCo, yCo) is upper left corner
   public:
      rect() : shape(), height(0), width(0)     //no-arg ctor
         {  }                                   //6-arg ctor
      rect(int x, int y, int h, int w, color fc, fstyle fs) :
                       shape(x, y, fc, fs), height(h), width(w)
         {  }
      void draw()             //draw the rectangle
         {
         shape::draw();
         draw_rectangle(xCo, yCo, xCo+width, yCo+height);
         set_color(cWHITE);   //draw diagonal
         draw_line(xCo, yCo, xCo+width, yCo+height);
         }
   };
////////////////////////////////////////////////////////////////
class tria : public shape
   {
   private:
      int height;             //(xCo, yCo) is tip of pyramid
   public:
      tria() : shape(), height(0) //no-arg constructor
         { }                  //5-arg constructor
      tria(int x, int y, int h, color fc, fstyle fs) :
                                 shape(x, y, fc, fs), height(h)
         {  }
      void draw()             //draw the triangle
         {
         shape::draw();
         draw_pyramid(xCo, yCo, height);
         }
   };
////////////////////////////////////////////////////////////////
int main()
   {
   int j;
   init_graphics();             //initialize graphics system

   shape* pShapes[3];           //array of pointers to shapes
                                //define three shapes
   pShapes[0] = new ball(40, 12, 5, cBLUE, X_FILL);
   pShapes[1] = new rect(12, 7, 10, 15, cRED, SOLID_FILL);
   pShapes[2] = new tria(60, 7,  11, cGREEN, MEDIUM_FILL);
```

```
   for(j=0; j<3; j++)           //draw all shapes
      pShapes[j]->draw();

   for(j=0; j<3; j++)           //delete all shapes
      delete pShapes[j];
   set_cursor_pos(1, 25);
   return 0;
   }
```

The class specifiers in VIRTSHAP are similar to those in MULTSHAP, except that the `draw()` function in the `shape` class has been made into a pure virtual function.

In `main()`, we set up an array, `ptrarr`, of pointers to shapes. Next we create three objects, one of each class, and place their addresses in an array. Now it's easy to draw all three shapes. The statement

```
ptrarr[j]->draw();
```

does this as the loop variable `j` changes.

This is a powerful approach to combining graphics elements, especially when a large number of objects need to be grouped together and drawn as a unit.

Virtual Destructors

Base class destructors should always be virtual. Suppose you use `delete` with a base class pointer to a derived class object to destroy the derived-class object. If the base-class destructor is not virtual then `delete`, like a normal member function, calls the destructor for the base class, not the destructor for the derived class. This will cause only the base part of the object to be destroyed. The VIRTDEST program shows how this looks.

```
//vertdest.cpp
//tests non-virtual and virtual destructors
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class Base
   {
   public:
      ~Base()                         //non-virtual destructor
//    virtual ~Base()                 //virtual destructor
         { cout << "Base destroyed\n"; }
   };
////////////////////////////////////////////////////////////////
class Derv : public Base
  {
```

```
   public:
      ~Derv()
         { cout << "Derv destroyed\n"; }
   };
////////////////////////////////////////////////////////////////
int main()
   {
   Base* pBase = new Derv;
   delete pBase;
   return 0;
   }
```

The output for this program as written is

```
Base destroyed
```

This shows that the destructor for the `Derv` part of the object isn't called. In the listing the base class destructor is not virtual, but you can make it so by commenting out the first definition for the destructor and substituting the second. Now the output is

```
Derv destroyed
Base destroyed
```

Now both parts of the derived class object are destroyed properly. Of course, if none of the destructors has anything important to do (like deleting memory obtained with `new`) then virtual destructors aren't important. But in general, to ensure that derived-class objects are destroyed properly, you should make destructors in all base classes virtual.

Most class libraries have a base class that includes a virtual destructor, which ensures that all derived classes have virtual destructors.

Virtual Base Classes

Before leaving the subject of virtual programming elements, we should mention *virtual base classes* as they relate to multiple inheritance.

Consider the situation shown in Figure 11.3, with a base class, `Parent`; two derived classes, `Child1` and `Child2`; and a fourth class, `Grandchild`, derived from both `Child1` and `Child2`.

In this arrangement a problem can arise if a member function in the `Grandchild` class wants to access data or functions in the `Parent` class. The NORMBASE program shows what happens.

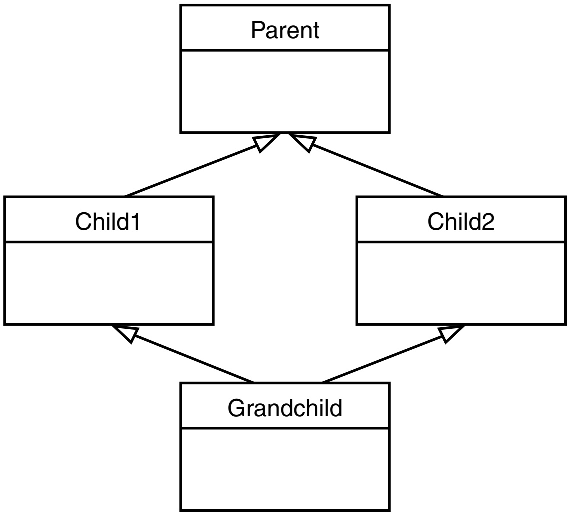

FIGURE 11.3
Virtual base classes.

```
// normbase.cpp
// ambiguous reference to base class

class Parent
   {
   protected:
      int basedata;
   };
class Child1 : public Parent
   { };
class Child2 : public Parent
   { };
class Grandchild : public Child1, public Child2
   {
   public:
      int getdata()
         { return basedata; }   // ERROR: ambiguous
   };
```

A compiler error occurs when the `getdata()` member function in `Grandchild` attempts to access `basedata` in `Parent`. Why? When the `Child1` and `Child2` classes are derived from `Parent`, each inherits a copy of `Parent`; this copy is called a *subobject*. Each of the two subobjects contains its own copy of `Parent`'s data, including `basedata`. Now, when `Grandchild` refers to `basedata`, which of the two copies will it access? The situation is ambiguous, and that's what the compiler reports.

To eliminate the ambiguity, we make `Child1` and `Child2` into virtual base classes, as shown by the example VIRTBASE.

```
// virtbase.cpp
// virtual base classes

class Parent
   {
   protected:
      int basedata;
   };
class Child1 : virtual public Parent   // shares copy of Parent
   { };
class Child2 : virtual public Parent   // shares copy of Parent
   { };
class Grandchild : public Child1, public Child2
   {
   public:
      int getdata()
         { return basedata; }    // OK: only one copy of Parent
   };
```

The use of the keyword `virtual` in these two classes causes them to share a single common subobject of their base class `Parent`. Since there is only one copy of `basedata`, there is no ambiguity when it is referred to in `Grandchild`.

The need for virtual base classes may indicate a conceptual problem with your use of multiple inheritance, so they should be used with caution.

Friend Functions

The concepts of encapsulation and data hiding dictate that nonmember functions should not be able to access an object's private or protected data. The policy is, if you're not a member, you can't get in. However, there are situations where such rigid discrimination leads to considerable inconvenience.

Friends as Bridges

Imagine that you want a function to operate on objects of two different classes. Perhaps the function will take objects of the two classes as arguments, and operate on their private data. In this situation there's nothing like a `friend` function. Here's a simple example, FRIEND, that shows how `friend` functions can act as a bridge between two classes:

```
// friend.cpp
// friend functions
```

```
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class beta;              //needed for frifunc declaration

class alpha
   {
   private:
      int data;
   public:
      alpha() : data(3) {  }            //no-arg constructor
      friend int frifunc(alpha, beta);  //friend function
   };
////////////////////////////////////////////////////////////////
class beta
   {
   private:
      int data;
   public:
      beta() : data(7) {  }             //no-arg constructor
      friend int frifunc(alpha, beta);  //friend function
   };
////////////////////////////////////////////////////////////////
int frifunc(alpha a, beta b)            //function definition
   {
   return( a.data + b.data );
   }
//--------------------------------------------------------------
int main()
   {
   alpha aa;
   beta bb;

   cout << frifunc(aa, bb) << endl;     //call the function
   return 0;
   }
```

In this program, the two classes are `alpha` and `beta`. The constructors in these classes initialize their single data items to fixed values (3 in `alpha` and 7 in `beta`).

We want the function `frifunc()` to have access to both of these private data members, so we make it a `friend` function. It's declared with the `friend` keyword in both classes:

```
friend int frifunc(alpha, beta);
```

This declaration can be placed anywhere in the class; it doesn't matter whether it goes in the `public` or the `private` section.

An object of each class is passed as an argument to the function `frifunc()`, and it accesses the private `data` member of both classes through these arguments. The function doesn't do much: It adds the data items and returns the sum. The `main()` program calls this function and prints the result.

A minor point: Remember that a class can't be referred to until it has been declared. Class `beta` is referred to in the declaration of the function `frifunc()` in class `alpha`, so `beta` must be declared before `alpha`. Hence the declaration

```
class beta;
```

at the beginning of the program.

Breaching the Walls

We should note that `friend` functions are controversial. During the development of C++, arguments raged over the desirability of including this feature. On the one hand, it adds flexibility to the language; on the other, it is not in keeping with *data hiding*, the philosophy that only member functions can access a class's private data.

How serious is the breach of data integrity when `friend` functions are used? A `friend` function must be declared as such within the class whose data it will access. Thus a programmer who does not have access to the source code for the class cannot make a function into a `friend`. In this respect, the integrity of the class is still protected. Even so, `friend` functions are conceptually messy, and potentially lead to a spaghetti-code situation if numerous `friends` muddy the clear boundaries between classes. For this reason `friend` functions should be used sparingly. If you find yourself using many `friends`, you may need to rethink the design of the program.

English Distance Example

However, sometimes `friend` functions are too convenient to avoid. Perhaps the most common example is when `friends` are used to increase the versatility of overloaded operators. The following program shows a limitation in the use of such operators when friends are not used. This example is a variation on the ENGLPLUS and ENGLCONV programs in Chapter 8, "Operator Overloading." It's called NOFRI.

```
// nofri.cpp
// limitation to overloaded + operator
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class Distance                      //English Distance class
   {
   private:
      int feet;
      float inches;
```

```
   public:
      Distance() : feet(0), inches(0.0)  //constructor (no args)
         {  }                            //constructor (one arg)
      Distance(float fltfeet)     //convert float to Distance
         {                        //feet is integer part
         feet = static_cast<int>(fltfeet);
          inches = 12*(fltfeet-feet); //inches is what's left
         }
      Distance(int ft, float in)  //constructor (two args)
         { feet = ft; inches = in; }
      void showdist()             //display distance
         { cout << feet << "\'-" << inches << '\"'; }
      Distance operator + (Distance);
   };
//---------------------------------------------------------------
                                  //add this distance to d2
Distance Distance::operator + (Distance d2)   //return the sum
   {
   int f = feet + d2.feet;        //add the feet
   float i = inches + d2.inches;  //add the inches
   if(i >= 12.0)                  //if total exceeds 12.0,
      { i -= 12.0; f++;  }        //less 12 inches, plus 1 foot
   return Distance(f,i);          //return new Distance with sum
   }
////////////////////////////////////////////////////////////////
int main()
   {
   Distance d1 = 2.5;             //constructor converts
   Distance d2 = 1.25;            //float feet to Distance
   Distance d3;
   cout << "\nd1 = "; d1.showdist();
   cout << "\nd2 = "; d2.showdist();

   d3 = d1 + 10.0;                //distance + float: OK
   cout << "\nd3 = "; d3.showdist();
// d3 = 10.0 + d1;                //float + Distance: ERROR
// cout << "\nd3 = "; d3.showdist();
   cout << endl;
   return 0;
   }
```

In this program, the + operator is overloaded to add two objects of type `Distance`. Also, there is a one-argument constructor that converts a value of type `float`, representing feet and decimal fractions of feet, into a `Distance` value. (That is, it converts 10.25' into 10'–3".)

When such a constructor exists, you can make statements like this in `main()`:

```
d3 = d1 + 10.0;
```

The overloaded `+` is looking for objects of type `Distance` both on its left and on its right, but if the argument on the right is type `float`, the compiler will use the one-argument constructor to convert this `float` to a `Distance` value, and then carry out the addition.

Here is what appears to be a subtle variation on this statement:

```
d3 = 10.0 + d1;
```

Does this work? No, because the object of which the overloaded `+` operator is a member must be the variable to the left of the operator. When we place a variable of a different type there, or a constant, then the compiler uses the `+` operator that adds that type (`float` in this case), not the one that adds `Distance` objects. Unfortunately, this operator does not know how to convert `float` to `Distance`, so it can't handle this situation. Here's the output from NOFRI:

```
d1 = 2'-6"
d2 = 1'-3"
d3 = 12'-6"
```

The second addition won't compile, so these statements are commented out. We could get around this problem by creating a new object of type `Distance`:

```
d3 = Distance(10, 0) + d1;
```

but this is nonintuitive and inelegant. How can we write natural-looking statements that have nonmember data types to the left of the operator? As you may have guessed, a `friend` can help you out of this dilemma. The FRENGL program shows how.

```
// frengl.cpp
// friend overloaded + operator
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class Distance                    //English Distance class
   {
   private:
      int feet;
      float inches;
   public:
      Distance()                  //constructor (no args)
         { feet = 0; inches = 0.0; }
      Distance( float fltfeet )   //constructor (one arg)
         {                        //convert float to Distance
         feet = int(fltfeet);           //feet is integer part
         inches = 12*(fltfeet-feet);    //inches is what's left
         }
```

```
      Distance(int ft, float in)  //constructor (two args)
         { feet = ft; inches = in; }
      void showdist()             //display distance
         { cout << feet << "\'-" << inches << '\"'; }
      friend Distance operator + (Distance, Distance); //friend
   };
//
Distance operator + (Distance d1, Distance d2) //add d1 to d2
   {
   int f = d1.feet + d2.feet;        //add the feet
   float i = d1.inches + d2.inches;  //add the inches
   if(i >= 12.0)                     //if inches exceeds 12.0,
      { i -= 12.0; f++;  }           //less 12 inches, plus 1 foot
   return Distance(f,i);             //return new Distance with sum
   }
//--------------------------------------------------------------
int main()
   {
   Distance d1 = 2.5;                //constructor converts
   Distance d2 = 1.25;               //float-feet to Distance
   Distance d3;
   cout << "\nd1 = "; d1.showdist();
   cout << "\nd2 = "; d2.showdist();

   d3 = d1 + 10.0;                   //distance + float: OK
   cout << "\nd3 = "; d3.showdist();
   d3 = 10.0 + d1;                   //float + Distance: OK
   cout << "\nd3 = "; d3.showdist();
   cout << endl;
   return 0;
   }
```

The overloaded + operator is made into a `friend`:

```
friend Distance operator + (Distance, Distance);
```

Notice that, while the overloaded + operator took one argument as a member function, it takes two as a `friend` function. In a member function, one of the objects on which the + operates is the object of which it was a member, and the second is an argument. In a `friend`, both objects must be arguments.

The only change to the body of the overloaded + function is that the variables `feet` and `inches`, used in NOFRI for direct access to the object's data, have been replaced in FRENGL by `d1.feet` and `d1.inches`, since this object is supplied as an argument.

Remember that, to make a function a friend, only the function declaration within the class is preceded by the keyword `friend`. The class definition is written normally, as are calls to the function.

`friends` for Functional Notation

Sometimes a `friend` allows a more obvious syntax for calling a function than does a member function. For example, suppose we want a function that will square (multiply by itself) an object of the English `Distance` class and return the result in square feet, as a type `float`. The MISQ example shows how this might be done with a member function.

```
// misq.cpp
// member square() function for 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 showdist()               //display distance
         { cout << feet << "\'-" << inches << '\"'; }
      float square();               //member function
   };
//--------------------------------------------------------------
float Distance::square()            //return square of
   {                                //this Distance
   float fltfeet = feet + inches/12;    //convert to float
   float feetsqrd = fltfeet * fltfeet;  //find the square
   return feetsqrd;                 //return square feet
   }
////////////////////////////////////////////////////////////////
int main()
   {
   Distance dist(3, 6.0);           //two-arg constructor (3'-6")
   float sqft;

   sqft = dist.square();            //return square of dist
                                    //display distance and square
```

```
   cout << "\nDistance = "; dist.showdist();
   cout << "\nSquare = " << sqft << " square feet\n";
   return 0;
   }
```

The `main()` part of the program creates a `Distance` value, squares it, and prints out the result. The output shows the original distance and the square:

```
Distance = 3'-6"
Square = 12.25 square feet
```

In `main()` we use the statement

```
sqft = dist.square();
```

to find the square of `dist` and assign it to `sqft`. This works all right, but if we want to work with `Distance` objects using the same syntax that we use with ordinary numbers, we would probably prefer a functional notation:

```
sqft = square(dist);
```

We can achieve this effect by making `square()` a `friend` of the `Distance` class, as shown in FRISQ:

```
// frisq.cpp
// friend square() function for Distance
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class Distance                        //English Distance class
   {
   private:
      int feet;
      float inches;
   public:
      Distance() : feet(0), inches(0.0) //constructor (no args)
         {  }
                                        //constructor (two args)
      Distance(int ft, float in) : feet(ft), inches(in)
         {  }
      void showdist()                   //display distance
         { cout << feet << "\'-" << inches << '\"'; }
      friend float square(Distance);    //friend function
   };
//--------------------------------------------------------------
float square(Distance d)              //return square of
   {                                  //this Distance
```

```
   float fltfeet = d.feet + d.inches/12;  //convert to float
   float feetsqrd = fltfeet * fltfeet;    //find the square
   return feetsqrd;               //return square feet
   }
////////////////////////////////////////////////////////////////
int main()
   {
   Distance dist(3, 6.0);         //two-arg constructor (3'-6")
   float sqft;

   sqft = square(dist);           //return square of dist
                                  //display distance and square
   cout << "\nDistance = "; dist.showdist();
   cout << "\nSquare = " << sqft << " square feet\n";
   return 0;
   }
```

Whereas `square()` takes no arguments as a member function in MISQ, it takes one as a `friend` in FRISQ. In general, the `friend` version of a function requires one more argument than when the function is a member. The `square()` function in FRISQ is similar to that in MISQ, but it refers to the data in the source `Distance` object as `d.feet` and `d.inches`, instead of as `feet` and `inches`.

`friend` Classes

The member functions of a class can all be made `friends` at the same time when you make the entire class a `friend`. The program FRICLASS shows how this looks.

```
// friclass.cpp
// friend classes
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class alpha
   {
   private:
      int data1;
   public:
      alpha() : data1(99) {  }   //constructor
      friend class beta;         //beta is a friend class
   };
////////////////////////////////////////////////////////////////
class beta
   {                             //all member functions can
```

```
   public:                       //access private alpha data
      void func1(alpha a)  { cout << "\ndata1=" << a.data1; }
      void func2(alpha a)  { cout << "\ndata1=" << a.data1; }
   };
////////////////////////////////////////////////////////////////
int main()
   {
   alpha a;
   beta b;

   b.func1(a);
   b.func2(a);
   cout << endl;
   return 0;
   }
```

In class `alpha` the entire class `beta` is proclaimed a `friend`. Now all the member functions of `beta` can access the private data of `alpha` (in this program, the single data item `data1`).

Note that in the `friend` declaration we specify that `beta` is a class using the `class` keyword:

```
friend class beta;
```

We could have also declared `beta` to be a class before the `alpha` class specifier, as in previous examples

```
class beta;
```

and then, within `alpha`, referred to `beta` without the `class` keyword:

```
friend beta;
```

Static Functions

In the STATIC example in Chapter 6, "Objects and Classes," we introduced `static` data members. As you may recall, a static data member is not duplicated for each object; rather a single data item is shared by all objects of a class. The STATIC example showed a class that kept track of how many objects of itself there were. Let's extend this concept by showing how functions as well as data may be `static`. Besides showing static functions, our example will model a class that provides an ID number for each of its objects. This allows you to query an object to find out which object it is—a capability that is sometimes useful in debugging a program, among other situations. The program also casts some light on the operation of destructors. Here's the listing for STATFUNC:

```
// statfunc.cpp
// static functions and ID numbers for objects
#include <iostream>
using namespace std;
```

```
/////////////////////////////////////////////////////////////////
class gamma
   {
   private:
      static int total;        //total objects of this class
                               //   (declaration only)
      int id;                  //ID number of this object
   public:
      gamma()                  //no-argument constructor
         {
         total++;              //add another object
         id = total;           //id equals current total
         }
      ~gamma()                 //destructor
         {
         total--;
         cout << "Destroying ID number " << id  << endl;
         }
      static void showtotal()  //static function
         {
         cout << "Total is " << total << endl;
         }
      void showid()            //non-static function
         {
         cout << "ID number is " << id << endl;
         }
   };
//---------------------------------------------------------------
int gamma::total = 0;           //definition of total
/////////////////////////////////////////////////////////////////
int main()
   {
   gamma g1;
   gamma::showtotal();

   gamma g2, g3;
   gamma::showtotal();

   g1.showid();
   g2.showid();
   g3.showid();
   cout << "----------end of program----------\n";
   return 0;
   }
```

Accessing `static` Functions

In this program there is a static data member, `total`, in the class `gamma`. This data keeps track of how many objects of the class there are. It is incremented by the constructor and decremented by the destructor.

Suppose we want to access `total` from outside the class. We construct a function, `showtotal()`, that prints the `total`'s value. But how do we access this function?

When a data member is declared `static`, there is only one such data value for the entire class, no matter how many objects of the class are created. In fact, there may be no such objects at all, but we still want to be able to learn this fact. We could create a dummy object to use in calling a member function, as in

```
gamma dummyObj;          // make an object so we can call function
dummyObj.showtotal();   // call function
```

But this is rather inelegant. We shouldn't need to refer to a specific object when we're doing something that relates to the entire class. It's more reasonable to use the name of the class itself with the scope-resolution operator.

```
gamma::showtotal();   // more reasonable
```

However, this won't work if `showtotal()` is a normal member function; an object and the dot member-access operator are required in such cases. To access `showtotal()` using only the class name, we must declare it to be a `static` member function. This is what we do in STATFUNC, in the declarator

```
static void showtotal()
```

Now the function can be accessed using only the class name. Here's the output:

```
Total is 1
Total is 3
ID number is 1
ID number is 2
ID number is 3
----------end of program--------
Destroying ID number 3
Destroying ID number 2
Destroying ID number 1
```

We define one object, `g1`, and then print out the value of `total`, which is 1. Then we define two more objects, `g2` and `g3`, and again print out the total, which is now 3.

Numbering the Objects

We've placed another function in `gamma()` to print out the ID number of individual members. This ID number is set equal to `total` when an object is created, so each object has a unique number. The `showid()` function prints out the ID of its object. We call it three times in `main()`, in the statements

```
g1.showid();
g2.showid();
g3.showid();
```

As the output shows, each object has a unique number. The `g1` object is numbered 1, `g2` is 2, and `g3` is 3.

Investigating Destructors

Now that we know how to number objects, we can investigate an interesting fact about destructors. STATFUNC prints an *end of program* message in its last statement, but it's not done yet, as the output shows. The three objects created in the program must be destroyed before the program terminates, so that memory is not left in an inaccessible state. The compiler takes care of this by invoking the destructor.

We can see that this happens by inserting a statement in the destructor that prints a message. Since we've numbered the objects, we can also find out the order in which the objects are destroyed. As the output shows, the last object created, `g3`, is destroyed first. One can infer from this last-in-first-out approach that local objects are stored on the stack.

Assignment and Copy Initialization

The C++ compiler is always busy on your behalf, doing things you can't be bothered to do. If you take charge, it will defer to your judgment; otherwise it will do things its own way. Two important examples of this process are the assignment operator and the copy constructor.

You've used the assignment operator many times, probably without thinking too much about it. Suppose `a1` and `a2` are objects. Unless you tell the compiler otherwise, the statement

```
a2 = a1;       // set a2 to the value of a1
```

will cause the compiler to copy the data from `a1`, member by member, into `a2`. This is the default action of the assignment operator, `=`.

You're also familiar with initializing variables. Initializing an object with another object, as in

```
alpha a2(a1);   // initialize a2 to the value of a1
```

causes a similar action. The compiler creates a new object, a2, and copies the data from a1, member by member, into a2. This is the default action of the copy constructor.

Both of these default activities are provided, free of charge, by the compiler. If member-by-member copying is what you want, you need take no further action. However, if you want assignment or initialization to do something more complex, you can override the default functions. We'll discuss the techniques for overloading the assignment operator and the copy constructor separately, and then put them together in an example that gives a String class a more efficient way to manage memory. We'll also introduce a new UML feature: the object diagram.

Overloading the Assignment Operator

Let's look at a short example that demonstrates the technique of overloading the assignment operator. Here's the listing for ASSIGN:

```
// assign.cpp
// overloads assignment operator (=)
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class alpha
   {
   private:
      int data;
   public:
      alpha()                        //no-arg constructor
         { }
      alpha(int d)                   //one-arg constructor
         { data = d; }
      void display()                 //display data
         { cout << data; }
      alpha operator = (alpha& a)   //overloaded = operator
         {
         data = a.data;             //not done automatically
         cout << "\nAssignment operator invoked";
         return alpha(data);        //return copy of this alpha
         }
   };
////////////////////////////////////////////////////////////////
int main()
   {
   alpha a1(37);
   alpha a2;
```

```
a2 = a1;                        //invoke overloaded =
cout << "\na2="; a2.display();  //display a2

alpha a3 = a2;                  //does NOT invoke =
cout << "\na3="; a3.display();  //display a3
cout << endl;
return 0;
}
```

The `alpha` class is very simple; it contains only one data member. Constructors initialize the data, and a member function can print out its value. The new aspect of ASSIGN is the function `operator=()`, which overloads the = operator.

In `main()`, we define `a1` and give it the value 37, and define `a2` but give it no value. Then we use the assignment operator to set `a2` to the value of `a1`:

```
a2 = a1;   // assignment statement
```

This causes our overloaded `operator=()` function to be invoked. Here's the output from ASSIGN:

```
Assignment operator invoked
a2=37
a3=37
```

Initialization Is Not Assignment

In the last two lines of ASSIGN, we initialize the object `a3` to the value `a2` and display it. Don't be confused by the syntax here. The equal sign in

```
alpha a3 = a2;   // copy initialization, not an assignment
```

is not an assignment but an initialization, with the same effect as

```
alpha a3(a2);    // alternative form of copy initialization
```

This is why the assignment operator is executed only once, as shown by the single invocation of the line

```
Assignment operator invoked
```

in the output of ASSIGN.

Taking Responsibility

When you overload the = operator you assume responsibility for doing whatever the default assignment operator did. Often this involves copying data members from one object to another. The `alpha` class in ASSIGN has only one data item, `data`, so the `operator=()` function copies its value with the statement

```
data = a.data;
```

The function also prints the `Assignment operator invoked` message so that we can tell when it executes.

Passing by Reference

Notice that the argument to `operator=()` is passed by reference. It is not absolutely necessary to do this, but it's usually a good idea. Why? As you know, an argument passed by value generates a copy of itself in the function to which it is passed. The argument passed to the `operator=()` function is no exception. If such objects are large, the copies can waste a lot of memory. Values passed by reference don't generate copies, and thus help to conserve memory.

Also, there are certain situations in which you want to keep track of the number of objects (as in the STATFUNC example, where we assigned numbers to the objects). If the compiler is generating extra objects every time you use the assignment operator, you may wind up with more objects than you expected. Passing by reference helps avoid such spurious object creation.

Returning a Value

As we've seen, a function can return information to the calling program by value or by reference. When an object is returned by value, a new object is created and returned to the calling program. In the calling program, the value of this object can be assigned to a new object or it can be used in other ways. When an object is returned by reference, no new object is created. A reference to the original object in the function is all that's returned to the calling program.

The `operator=()` function in ASSIGN returns a value by creating a temporary `alpha` object and initializing it using the one-argument constructor in the statement

```
return alpha(data);
```

The value returned is a copy of, but not the same object as, the object of which the overloaded = operator is a member. Returning a value makes it possible to chain = operators:

```
a3 = a2 = a1;
```

However, returning by value has the same disadvantages as passing an argument by value: It creates an extra copy that wastes memory and can cause confusion. Can we return this value with a reference, using the declarator shown here for the overloaded = operator?

```
alpha& operator = (alpha& a)  // bad idea in this case
```

Unfortunately, we can't use reference returns on variables that are local to a function. Remember that local (`automatic`) variables—that is, those created within a function (and not designated `static`)—are destroyed when the function returns. A return by reference returns only the address of the data being returned, and, for local data, this address points to data within the function. When the function is terminated and this data is destroyed, the pointer is left with a meaningless value. Your compiler may flag this usage with a warning. (We'll see one way to solve this problem in the section "The `this` Pointer" later in this chapter.)

Not Inherited

The assignment operator is unique among operators in that it is not inherited. If you overload the assignment operator in a base class, you can't use this same function in any derived classes.

The Copy Constructor

As we discussed, you can define and at the same time initialize an object to the value of another object with two kinds of statements:

```
alpha a3(a2);    // copy initialization
alpha a3 = a2;   // copy initialization, alternate syntax
```

Both styles of definition invoke a copy constructor: a constructor that creates a new object and copies its argument into it. The default copy constructor, which is provided automatically by the compiler for every object, performs a member-by-member copy. This is similar to what the assignment operator does; the difference is that the copy constructor also creates a new object.

Like the assignment operator, the copy constructor can be overloaded by the user. The XOFXREF example shows how it's done.

```
// xofxref.cpp
// copy constructor: X(X&)
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class alpha
   {
   private:
      int data;
   public:
      alpha()                    //no-arg constructor
         { }
      alpha(int d)               //one-arg constructor
         { data = d; }
      alpha(alpha& a)            //copy constructor
         {
         data = a.data;
         cout << "\nCopy constructor invoked";
         }
      void display()             //display
         { cout << data; }
      void operator = (alpha& a) //overloaded = operator
         {
         data = a.data;
         cout << "\nAssignment operator invoked";
         }
   };
```

```
////////////////////////////////////////////////////////////////
int main()
   {
   alpha a1(37);
   alpha a2;

   a2 = a1;                         //invoke overloaded =
   cout << "\na2="; a2.display();   //display a2

   alpha a3(a1);                    //invoke copy constructor
// alpha a3 = a1;                   //equivalent definition of a3
   cout << "\na3="; a3.display();   //display a3
   cout << endl;
   return 0;
   }
```

This program overloads both the assignment operator and the copy constructor. The overloaded assignment operator is similar to that in the ASSIGN example. The copy constructor takes one argument: an object of type `alpha`, passed by reference. Here's its declarator:

```
alpha(alpha&)
```

This declarator has the form `X(X&)` (pronounced "X of X ref"). Here's the output of XOFXREF:

```
Assignment operator invoked
a2=37
Copy constructor invoked
a3=37
```

The statement

```
a2 = a1;
```

invokes the assignment operator, while

```
alpha a3(a1);
```

invokes the copy constructor. The equivalent statement

```
alpha a3 = a1;
```

could also be used to invoke the copy constructor.

We've seen that the copy constructor may be invoked when an object is defined. It is also invoked when arguments are passed by value to functions and when values are returned from functions. Let's discuss these situations briefly.

Function Arguments

The copy constructor is invoked when an object is passed by value to a function. It creates the copy that the function operates on. Thus if the function

```
void func(alpha);
```

were declared in XOFXREF, and this function were called by the statement

```
func(a1);
```

then the copy constructor would be invoked to create a copy of the `a1` object for use by `func()`. (Of course, the copy constructor is not invoked if the argument is passed by reference or if a pointer to it is passed. In these cases no copy is created; the function operates on the original variable.)

Function Return Values

The copy constructor also creates a temporary object when a value is returned from a function. Suppose there was a function like this in XOFXREF

```
alpha func();
```

and this function was called by the statement

```
a2 = func();
```

The copy constructor would be invoked to create a copy of the value returned by `func()`, and this value would be assigned (invoking the assignment operator) to `a2`.

Why Not an `X(X)` Constructor?

Do we need to use a reference in the argument to the copy constructor? Could we pass by value instead? No, the compiler complains that it is `out of memory` if we try to compile

```
alpha(alpha a)
```

Why? Because when an argument is passed by value, a copy of it is constructed. What makes the copy? The copy constructor. But this *is* the copy constructor, so it calls itself. In fact, it calls itself over and over until the compiler runs out of memory. So, in the copy constructor, the argument must be passed by reference, which creates no copies.

Watch Out for Destructors

In the sections "Passing by Reference" and "Returning a Value," we discussed passing arguments to a function by value and returning by value. These situations cause the destructor to be called as well, when the temporary objects created by the function are destroyed when the function returns. This can cause considerable consternation if you're not expecting it. The moral is, when working with objects that require more than member-by-member copying, pass and return by reference—not by value—whenever possible.

Define Both Copy Constructor and Assignment Operator

When you overload the assignment operator, you almost always want to overload the copy constructor as well (and vice versa). You don't want your custom copying routine used in some situations, and the default member-by-member scheme used in others. Even if you don't think you'll use one or the other, you may find the compiler using them in nonobvious situations, such as passing an argument to a function by value, and returning from a function by value.

In fact, if the constructor to a class involves the use of system resources such as memory or disk files, you should almost always overload both the assignment operator and the copy constructor, and make sure they do what you want.

How to Prohibit Copying

We've discussed how to customize the copying of objects using the assignment operator and the copy constructor. Sometimes, however, you may want to prohibit the copying of an object using these operations. For example, it might be essential that each member of a class be created with a unique value for some member, which is provided as an argument to the constructor. If an object is copied, the copy will be given the same value. To avoid copying, overload the assignment operator and the copy constructor as private members.

```
class alpha
   {
   private:
      alpha& operator = (alpha&);  // private assignment operator
      alpha(alpha&);               // private copy constructor
   };
```

As soon as you attempt a copying operation, such as

```
alpha a1, a2;
a1 = a2;          // assignment
alpha a3(a1);     // copy constructor
```

the compiler will tell you that the function is not accessible. You don't need to define the functions, since they will never be called.

UML Object Diagrams

We've seen examples of class diagrams in previous chapters. It will probably not surprise you to know that the UML supports object diagrams as well. Object diagrams depict specific objects (for instance, the `Mike_Gonzalez` object of the `Professor` class). Because the relationships among objects change during the course of a program's operation (indeed, objects may even be created and destroyed) an object diagram is like a snapshot, representing objects at a particular moment in time. It's said to be a *static* UML diagram.

You use an object diagram to model a particular thing your program does. You freeze the program at a moment in time and look at the objects that participate in the behavior you're interested in, and the communications among these objects at that point in time.

In object diagrams, objects are represented by rectangles, just as classes are in class diagrams. The object's name, attributes, and operations are shown in a similar way. However, objects are distinguished from classes by having their names underlined. Both the object name and the class name can be used, separated by a colon:

`anObj:aClass`

If you don't know the name of the object (because it's only known through a pointer, for example) you can use just the class name preceded by the colon:

`:aClass`

Lines between the objects are called *links*, and represent one object communicating with another. Navigability can be shown. The value of an attribute can be shown using an equal sign:

```
count = 0
```

Notice there's no semicolon at the end; this is the UML, not C++.

Another UML feature we'll encounter is the *note*. Notes are shown as rectangles with a dog-eared (turned down) corner. They hold comments or explanations. A dotted line connects a note to the relevent element in the diagram. Unlike associations and links, a note can refer to an element inside a class or object rectangle. Notes can be used in any kind of UML diagram.

We'll see a number of object diagrams in the balance of this chapter.

A Memory-Efficient `String` Class

The ASSIGN and XOFXREF examples don't really need to have overloaded assignment operators and copy constructors. They use straightforward classes with only one data item, so the default assignment operator and copy constructor would work just as well. Let's look at an example where it is essential for the user to overload these operators.

Defects with the `String` Class

We've seen various versions of our homemade `String` class in previous chapters. However, these versions are not very sophisticated. It would be nice to overload the = operator so that we could assign the value of one `String` object to another with the statement

```
s2 = s1;
```

If we overload the = operator, the question arises of how we will handle the actual string (the array of type `char`), which is the principal data item in the `String` class.

One possibility is for each `String` object to have a place to store a string. If we assign one `String` object to another (from `s1` into `s2` in the previous statement), we simply copy the string from the source into the destination object. If you're concerned with conserving memory, the problem with this is that the same string now exists in two (or more) places in memory. This is not very efficient, especially if the strings are long. Figure 11.4 shows how this looks.

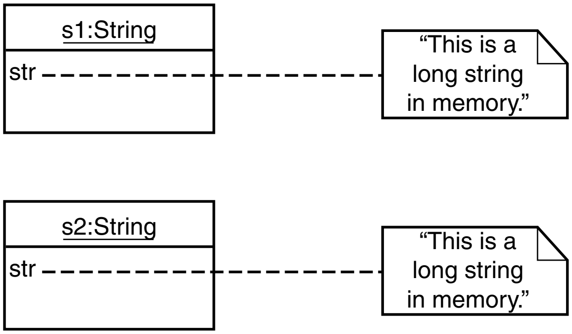

FIGURE 11.4
UML object diagram: replicating strings.

Instead of having each `String` object contain its own `char*` string, we could arrange for it to contain only a *pointer* to a string. Now, if we assign one `String` object to another, we need only copy the pointer from one object to another; both pointers will point to the same string. This is efficient, since only a single copy of the string itself needs to be stored in memory. Figure 11.5 shows how this looks.

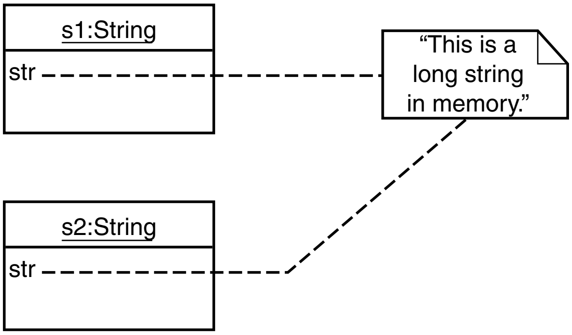

FIGURE 11.5
UML object diagram: replicating pointers to strings.

However, if we use this system we need to be careful when we destroy a `String` object. If a `String`'s destructor uses `delete` to free the memory occupied by the string, and if there are several objects with pointers pointing to the string, these other objects will be left with pointers pointing to memory that may no longer hold the string they think it does; they become dangling pointers.

To use pointers to strings in `String` objects, we need a way to keep track of how many `String` objects point to a particular string, so that we can avoid using `delete` on the string until the last `String` that points to it is itself deleted. Our next example, STRIMEM, does just this.

A String-Counter Class

Suppose we have several `String` objects pointing to the same string and we want to keep a count of how many `Strings` point to the string. Where will we store this count?

It would be cumbersome for every `String` object to maintain a count of how many of its fellow `Strings` were pointing to a particular string, so we don't want to use a member variable in `String` for the count. Could we use a static variable? This is a possibility; we could create a static array and use it to store a list of string addresses and counts. However, this requires considerable overhead. It's more efficient to create a new class to store the count. Each object of this class, which we call `strCount`, contains a count and also a pointer to the string itself. Each `String` object contains a pointer to the appropriate `strCount` object. Figure 11.6 shows how this looks.

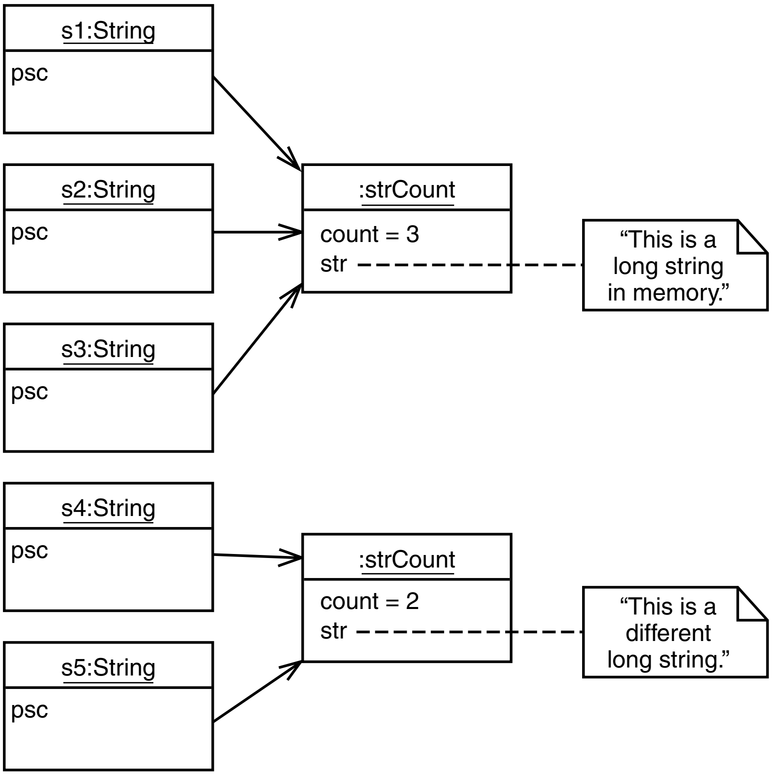

FIGURE 11.6
`String` *and* `strCount` *objects.*

To ensure that String objects have access to strCount objects, we make String a friend of strCount. Also, we want to ensure that the strCount class is used only by the String class. To prevent access to any of its functions, we make all member functions of strCount private. Because String is a friend, it can nevertheless access any part of strCount. Here's the listing for STRIMEM:

```
// strimem.cpp
// memory-saving String class
// overloaded assignment and copy constructor
#include <iostream>
#include <cstring>                 //for strcpy(), etc.
using namespace std;
////////////////////////////////////////////////////////////////
class strCount                     //keep track of number
   {                               //of unique strings
   private:
      int count;                   //number of instances
      char* str;                   //pointer to string
      friend class String;         //make ourselves available
      //member functions are private
//--------------------------------------------------------------
      strCount(char* s)            //one-arg constructor
         {
         int length = strlen(s);  //length of string argument
         str = new char[length+1];  //get memory for string
         strcpy(str, s);           //copy argument to it
         count=1;                  //start count at 1
         }
//--------------------------------------------------------------
      ~strCount()                  //destructor
         { delete[] str; }         //delete the string
   };
////////////////////////////////////////////////////////////////
class String                       //String class
   {
   private:
      strCount* psc;               //pointer to strCount
   public:
      String()                     //no-arg constructor
         { psc = new strCount("NULL"); }
//--------------------------------------------------------------
      String(char* s)              //1-arg constructor
         { psc = new strCount(s); }
//--------------------------------------------------------------
      String(String& S)            //copy constructor
         {
```

```
         psc = S.psc;
         (psc->count)++;
         }
//----------------------------------------------------------------
      ~String()                   //destructor
         {
         if(psc->count==1)        //if we are its last user,
            delete psc;           //   delete our strCount
         else                     //   otherwise,
            (psc->count)--;       //   decrement its count
         }
//----------------------------------------------------------------
      void display()              //display the String
         {
         cout << psc->str;                 //print string
         cout << " (addr=" << psc << ")";  //print address
         }
//----------------------------------------------------------------
      void operator = (String& S) //assign the string
         {
         if(psc->count==1)        //if we are its last user,
            delete psc;           //   delete our strCount
         else                     //   otherwise,
            (psc->count)--;       //   decrement its count
         psc = S.psc;             //use argument's strCount
         (psc->count)++;          //increment its count
         }
   };
////////////////////////////////////////////////////////////////
int main()
   {
   String s3 = "When the fox preaches, look to your geese.";
   cout << "\ns3="; s3.display();   //display s3

   String s1;                       //define String
   s1 = s3;                         //assign it another String
   cout << "\ns1="; s1.display();   //display it

   String s2(s3);                   //initialize with String
   cout << "\ns2="; s2.display();   //display it
   cout << endl;
   return 0;
   }
```

In the `main()` part of STRIMEM we define a `String` object, `s3`, to contain the proverb "When the fox preaches, look to your geese." We define another `String s1` and set it equal to `s3`; then we define `s2` and initialize it to `s3`. Setting `s1` equal to `s3` invokes the overloaded assignment operator; initializing `s2` to `s3` invokes the overloaded copy constructor. We print out all three strings, and also the address of the `strCount` object pointed to by each object's `psc` pointer, to show that these objects are all the same. Here's the output from STRIMEM:

```
s3=When the fox preaches, look to your geese. (addr=0x8f510e00)
s1=When the fox preaches, look to your geese. (addr=0x8f510e00)
s2=When the fox preaches, look to your geese. (addr=0x8f510e00)
```

The other duties of the `String` class are divided between the `String` and `strCount` classes. Let's see what they do.

The `strCount` Class

The `strCount` class contains the pointer to the actual string and the count of how many `String` class objects point to this string. Its single constructor takes a pointer to a string as an argument and creates a new memory area for the string. It copies the string into this area and sets the count to 1, since just one `String` points to it when it is created. The destructor in `strCount` frees the memory used by the string. (We use `delete[]` with brackets because a string is an array.)

The `String` Class

The `String` class uses three constructors. If a new string is being created, as in the zero-argument and C-string-argument constructors, a new `strCount` object is created to hold the string, and the `psc` pointer is set to point to this object. If an existing `String` object is being copied, as in the copy constructor and the overloaded assignment operator, the pointer `psc` is set to point to the old `strCount` object, and the count in this object is incremented.

The overloaded assignment operator, as well as the destructor, must also delete the old `strCount` object pointed to by `psc` if the count is 1. (We don't need brackets on `delete` because we're deleting only a single `strCount` object.) Why must the assignment operator worry about deletion? Remember that the `String` object on the left of the equal sign (call it `s1`) was pointing at some `strCount` object (call it `oldStrCnt`) before the assignment. After the assignment `s1` will be pointing to the object on the right of the equal sign. If there are now no `String` objects pointing to `oldStrCnt`, it should be deleted. If there are other objects pointing to it, its count must be decremented. Figure 11.7 shows the action of the overloaded assignment operator, and Figure 11.8 shows the copy constructor.

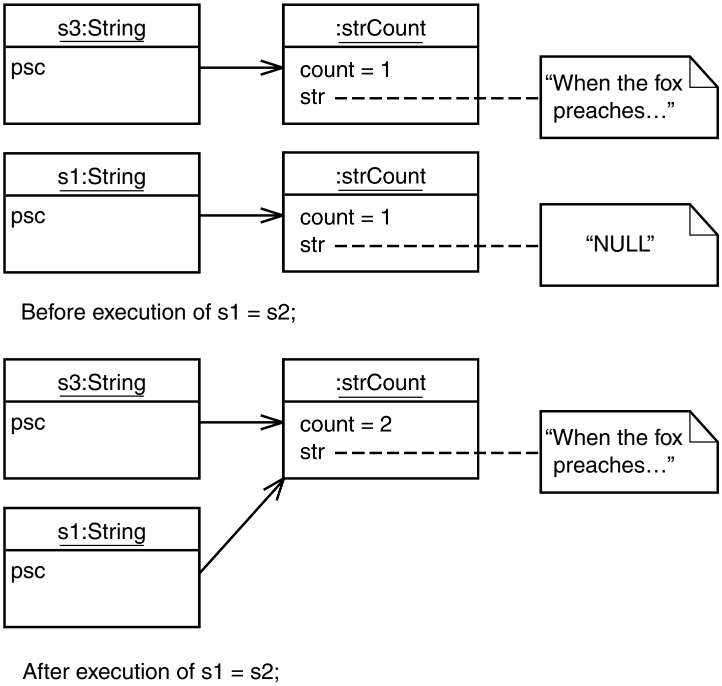

FIGURE 11.7

Assignment operator in STRIMEM.

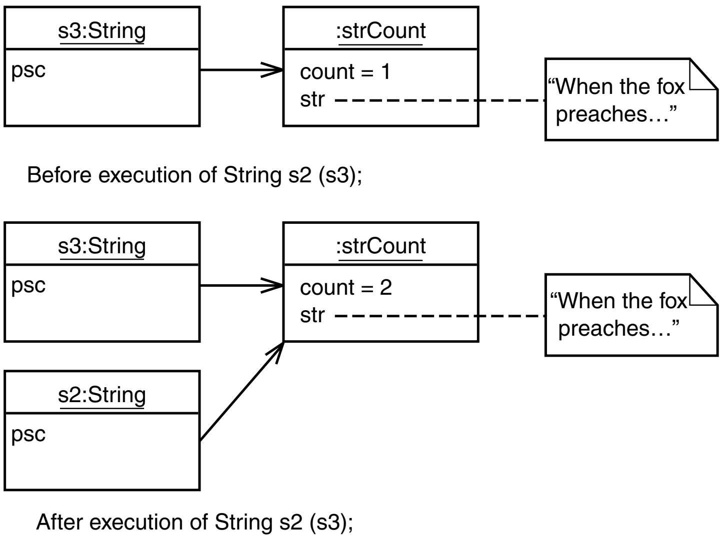

FIGURE 11.8

Copy constructor in STRIMEM.

The this Pointer

The member functions of every object have access to a sort of magic pointer named `this`, which points to the object itself. Thus any member function can find out the address of the object of which it is a member. Here's a short example, WHERE, that shows the mechanism:

```
// where.cpp
// the this pointer
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class where
   {
   private:
      char charray[10];   //occupies 10 bytes
   public:
      void reveal()
         { cout << "\nMy object's address is " << this; }
   };
////////////////////////////////////////////////////////////////
int main()
   {
   where w1, w2, w3;      //make three objects
   w1.reveal();           //see where they are
   w2.reveal();
   w3.reveal();
   cout << endl;
   return 0;
   }
```

The `main()` program in this example creates three objects of type `where`. It then asks each object to print its address, using the `reveal()` member function. This function prints out the value of the `this` pointer. Here's the output:

```
My object's address is 0x8f4effec
My object's address is 0x8f4effe2
My object's address is 0x8f4effd8
```

Since the data in each object consists of an array of 10 bytes, the objects are spaced 10 bytes apart in memory. (EC minus E2 is 10 decimal, as is E2 minus D8.) Some compilers may place extra bytes in objects, making them slightly larger than 10 bytes.

Accessing Member Data with this

When you call a member function, it comes into existence with the value of `this` set to the address of the object for which it was called. The `this` pointer can be treated like any other pointer to an object, and can thus be used to access the data in the object it points to, as shown in the DOTHIS program:

```
// dothis.cpp
// the this pointer referring to data
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class what
   {
   private:
      int alpha;
   public:
      void tester()
         {
         this->alpha = 11;     //same as alpha = 11;
         cout << this->alpha;  //same as cout << alpha;
         }
   };
////////////////////////////////////////////////////////////////
int main()
   {
   what w;
   w.tester();
   cout << endl;
   return 0;
   }
```

This program simply prints out the value 11. Notice that the `tester()` member function accesses the variable `alpha` as

```
this->alpha
```

This is exactly the same as referring to `alpha` directly. This syntax works, but there is no reason for it except to show that `this` does indeed point to the object.

Using `this` for Returning Values

A more practical use for `this` is in returning values from member functions and overloaded operators.

Recall that in the ASSIGN program we could not return an object by reference, because the object was local to the function returning it and thus was destroyed when the function returned. We need a more permanent object if we're going to return it by reference. The object of which a function is a member is more permanent than its individual member functions. An object's member functions are created and destroyed every time they're called, but the object itself endures until it is destroyed by some outside agency (for example, when it is `delete`d). Thus returning by reference the object of which a function is a member is a better bet than returning a temporary object created in a member function. The `this` pointer makes this easy.

Here's the listing for ASSIGN2, in which the `operator=()` function returns by reference the object that invoked it:

```
//assign2.cpp
// returns contents of the this pointer
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class alpha
   {
   private:
      int data;
   public:
      alpha()                      //no-arg constructor
         { }
      alpha(int d)                 //one-arg constructor
         { data = d; }
      void display()               //display data
         { cout << data; }
      alpha& operator = (alpha& a) //overloaded = operator
         {
         data = a.data;            //not done automatically
         cout << "\nAssignment operator invoked";
         return *this;             //return copy of this alpha
         }
   };
////////////////////////////////////////////////////////////////
int main()
   {
   alpha a1(37);
   alpha a2, a3;

   a3 = a2 = a1;                   //invoke overloaded =, twice
   cout << "\na2="; a2.display();  //display a2
   cout << "\na3="; a3.display();  //display a3
   cout << endl;
   return 0;
   }
```

In this program we can use the declaration

```
alpha& operator = (alpha& a)
```

which returns by reference, instead of

```
alpha operator = (alpha& a)
```

which returns by value. The last statement in this function is

```
return *this;
```

Since `this` is a pointer to the object of which the function is a member, `*this` is that object itself, and the statement returns it by reference. Here's the output of ASSIGN2:

```
Assignment operator invoked
Assignment operator invoked
a2=37
a3=37
```

Each time the equal sign is encountered in

```
a3 = a2 = a1;
```

the overloaded `operator=()` function is called, which prints the messages. The three objects all end up with the same value.

You usually want to return by reference from overloaded assignment operators, using `*this`, to avoid the creation of extra objects.

Revised STRIMEM Program

Using the `this` pointer we can revise the `operator=()` function in STRIMEM to return a value by reference, thus making possible multiple assignment operators for `String` objects, such as

```
s1 = s2 = s3;
```

At the same time, we can avoid the creation of spurious objects, such as those that are created when objects are returned by value. Here's the listing for STRIMEM2:

```
// strimem2.cpp
// memory-saving String class
// the this pointer in overloaded assignment
#include <iostream>
#include <cstring>                  //for strcpy(), etc
using namespace std;
////////////////////////////////////////////////////////////////
class strCount                      //keep track of number
   {                                //of unique strings
   private:
      int count;                    //number of instances
      char* str;                    //pointer to string
      friend class String;          //make ourselves available
   //member functions are private
      strCount(char* s)             //one-arg constructor
         {
         int length = strlen(s);    //length of string argument
         str = new char[length+1];  //get memory for string
         strcpy(str, s);            //copy argument to it
         count=1;                   //start count at 1
         }
```

```
//--------------------------------------------------------------
      ~strCount()                    //destructor
         { delete[] str; }           //delete the string
   };
////////////////////////////////////////////////////////////////
class String                         //String class
   {
   private:
      strCount* psc;                 //pointer to strCount
   public:
      String()                       //no-arg constructor
         { psc = new strCount("NULL"); }
//--------------------------------------------------------------
      String(char* s)                //1-arg constructor
         { psc = new strCount(s); }
//--------------------------------------------------------------
      String(String& S)              //copy constructor
         {
         cout << "\nCOPY CONSTRUCTOR";
         psc = S.psc;
            (psc->count)++;
         }
//--------------------------------------------------------------
      ~String()                      //destructor
         {
         if(psc->count==1)           //if we are its last user,
            delete psc;              //  delete our strCount
         else                        //  otherwise,
            (psc->count)--;          //  decrement its count
         }
//--------------------------------------------------------------
      void display()                    //display the String
         {
         cout << psc->str;                    //print string
         cout << " (addr=" << psc << ")";  //print address
         }
//--------------------------------------------------------------
      String& operator = (String& S)       //assign the string
         {
         cout << "\nASSIGNMENT";
         if(psc->count==1)           //if we are its last user,
            delete psc;              //delete our strCount
         else                        //  otherwise,
            (psc->count)--;          //  decrement its count
         psc = S.psc;                //use argument's strCount
         (psc->count)++;             //increment count
```

```
        return *this;            //return this object
        }
   };

int main()
   {
   String s3 = "When the fox preaches, look to your geese.";
   cout << "\ns3="; s3.display();  //display s3

   String s1, s2;                  //define Strings
   s1 = s2 = s3;                   //assign them
   cout << "\ns1="; s1.display();  //display it
   cout << "\ns2="; s2.display();  //display it
   cout << endl;                   //wait for keypress
   return 0;
   }
```

Now the declarator for the = operator is

```
String& operator = (String& S)  // return by reference
```

And, as in ASSIGN2, this function returns a pointer to `this`. Here's the output:

```
s3=When the fox preaches, look to your geese. (addr=0x8f640d3a)
ASSIGNMENT
ASSIGNMENT
s1=When the fox preaches, look to your geese. (addr=0x8f640d3a)
s2=When the fox preaches, look to your geese. (addr=0x8f640d3a)
```

The output shows that, following the assignment statement, all three `String` objects point to the same `strCount` object.

We should note that the `this` pointer is not available in static member functions, since they are not associated with a particular object.

Beware of Self-Assignment

A corollary of Murphy's Law states that whatever is possible, someone will eventually do. This is certainly true in programming, so you can expect that if you have overloaded the = operator, someone will use it to set an object equal to itself:

```
alpha = alpha;
```

Your overloaded assignment operator should be prepared to handle such self-assignment. Otherwise, bad things may happen. For example, in the `main()` part of the STRIMEM2 program, if you set a `String` object equal to itself, the program will crash (unless there are other `String` objects using the same `strCount` object). The problem is that the code for the assignment operator deletes the `strCount` object if it thinks the object that called it is the only object using the `strCount`. Self-assignment will cause it to believe this, even though nothing should be deleted.

To fix this, you should check for self-assignment at the start of any overloaded assignment operator. You can do this in most cases by comparing the address of the object for which the operator was called with the address of its argument. If the addresses are the same, the objects are identical and you should return immediately. (You don't need to assign one to the other; they're already the same.) For example, in STRIMEM2, you can insert the lines

```
if(this == &S)
   return *this;
```

at the start of `operator=()`. That should solve the problem.

Dynamic Type Information

It's possible to find out information about an object's class and even change the class of an object at runtime. We'll look briefly at two mechanisms: the `dynamic_cast` operator, and the `typeid` operator. These are advanced capabilities, but you may find them useful someday.

These capabilities are usually used in situations where a variety of classes are descended (sometimes in complicated ways) from a base class. For dynamic casts to work, the base class must be polymorphic; that is, it must have at least one virtual function.

For both `dynamic_cast` and `typeid` to work, your compiler must enable Run-Time Type Information (RTTI). Borland C++Builder has this capability enabled by default, but in Microsoft Visual C++ you'll need to turn it on overtly. See Appendix C, "Microsoft Visual C++," for details on how this is done. You'll also need to include the header file TYPEINFO.

Checking the Type of a Class with `dynamic_cast`

Suppose some other program sends your program an object (as the operating system might do with a callback function). It's supposed to be a certain type of object, but you want to check it to be sure. How can you tell if an object is a certain type? The `dynamic_cast` operator provides a way, assuming that the classes whose objects you want to check are all descended from a common ancestor. The DYNCAST1 program shows how this looks.

```
//dyncast1.cpp
//dynamic cast used to test type of object
//RTTI must be enabled in compiler
#include <iostream>
#include <typeinfo>           //for dynamic_cast
using namespace std;
////////////////////////////////////////////////////////////////
class Base
   {
   virtual void vertFunc()    //needed for dynamic cast
      {  }
   };
```

```
class Derv1 : public Base
   {  };
class Derv2 : public Base
   {  };
////////////////////////////////////////////////////////////////
//checks if pUnknown points to a Derv1
bool isDerv1(Base* pUnknown)  //unknown subclass of Base
   {
   Derv1* pDerv1;
   if( pDerv1 = dynamic_cast<Derv1*>(pUnknown) )
      return true;
   else
      return false;
   }
//--------------------------------------------------------------
int main()
   {
   Derv1* d1 = new Derv1;
   Derv2* d2 = new Derv2;

   if( isDerv1(d1) )
      cout << "d1 is a member of the Derv1 class\n";
   else
      cout << "d1 is not a member of the Derv1 class\n";

   if( isDerv1(d2) )
      cout << "d2 is a member of the Derv1 class\n";
   else
      cout << "d2 is not a member of the Derv1 class\n";
   return 0;
   }
```

Here we have a base class `Base` and two derived classes `Derv1` and `Derv2`. There's also a function, `isDerv1()`, which returns true if the pointer it received as an argument points to an object of class `Derv1`. This argument is of class `Base`, so the object passed can be either `Derv1` or `Derv2`. The `dynamic_cast` operator attempts to convert this unknown pointer `pUnknown` to type `Derv1`. If the result is not zero, `pUnknown` did point to a `Derv1` object. If the result is zero, it pointed to something else.

Changing Pointer Types with `dynamic_cast`

The `dynamic_cast` operator allows you to cast upward and downward in the inheritance tree. However, it allows such casting only in limited ways. The DYNCAST2 program shows examples of such casts.

```
//dyncast2.cpp
//tests dynamic casts
//RTTI must be enabled in compiler
#include <iostream>
#include <typeinfo>            //for dynamic_cast
using namespace std;
////////////////////////////////////////////////////////////////
class Base
   {
   protected:
      int ba;
   public:
      Base() : ba(0)
         {  }
      Base(int b) : ba(b)
         {  }
      virtual void vertFunc()  //needed for dynamic_cast
         {  }
      void show()
         { cout << "Base: ba=" << ba << endl; }
   };
////////////////////////////////////////////////////////////////
class Derv : public Base
   {
   private:
      int da;
   public:
      Derv(int b, int d) : da(d)
         { ba = b; }
      void show()
         { cout << "Derv: ba=" << ba << ", da=" << da << endl; }
   };
////////////////////////////////////////////////////////////////
int main()
   {
   Base* pBase = new Base(10);           //pointer to Base
   Derv* pDerv = new Derv(21, 22);       //pointer to Derv

   //derived-to-base: upcast -- points to Base subobject of Derv
   pBase = dynamic_cast<Base*>(pDerv);
   pBase->show();                        //"Base: ba=21"

   pBase = new Derv(31, 32);             //normal
   //base-to-derived: downcast -- (pBase must point to a Derv)
   pDerv = dynamic_cast<Derv*>(pBase);
   pDerv->show();                        //"Derv: ba=31, da=32"
   return 0;
   }
```

Here we have a base and a derived class. We've given each of these classes a data item to better demonstrate the effects of dynamic casts.

In an upcast you attempt to change a derived-class object into a base-class object. What you get is the base part of the derived class object. In the example we make an object of class `Derv`. The base class part of this object holds member data `ba`, which has a value of 21, and the derived part holds data member `da`, which has the value 22. After the cast, `pBase` points to the base-class part of this `Derv` class object, so when called upon to display itself, it prints `Base: ba=21`. Upcasts are fine if all you want is the base part of the object.

In a downcast we put a derived class object, which is pointed to by a base-class pointer, into a derived-class pointer.

The `typeid` Operator

Sometimes you want more information about an object than simple verification that it's of a certain class. You can obtain information about the type of an unknown object, such as its class name, using the `typeid` operator. The TYPEID program demonstrates how it works.

```
// typeid.cpp
// demonstrates typeid() function
// RTTI must be enabled in compiler
#include <iostream>
#include <typeinfo>           //for typeid()
using namespace std;
////////////////////////////////////////////////////////////////
class Base
   {
   virtual void virtFunc()    //needed for typeid
      {  }
   };
class Derv1 : public Base
   { };
class Derv2 : public Base
   { };
////////////////////////////////////////////////////////////////
void displayName(Base* pB)
   {
   cout << "pointer to an object of ";  //display name of class
   cout << typeid(*pB).name() << endl;  //pointed to by pB
   }
//--------------------------------------------------------------
int main()
   {
   Base* pBase = new Derv1;
```

```
displayName(pBase);   //"pointer to an object of class Derv1"

pBase = new Derv2;
displayName(pBase);   //"pointer to an object of class Derv2"
return 0;
}
```

In this example the `displayName()` function displays the name of the class of the object passed to it. To do this, it uses the `name` member of the `type_info` class, along with the `typeid` operator. In `main()` we pass this function two objects of class `Derv1` and `Derv2` respectively, and the program's output is

```
pointer to an object of class Derv1
pointer to an object of class Derv2
```

Besides its name, other information about a class is available using `typeid`. For example, you can check for equality of classes using an overloaded == operator. We'll show an example of this in the EMPL_IO program in Chapter 12, "Streams and Files." Although the examples in this section have used pointers, `dynamic_cast` and `typeid` work equally well with references.

Summary

Virtual functions provide a way for a program to decide while it is running what function to call. Ordinarily such decisions are made at compile time. Virtual functions make possible greater flexibility in performing the same kind of action on different kinds of objects. In particular, they allow the use of functions called from an array of type pointer-to-base that actually holds pointers (or references) to a variety of derived types. This is an example of *polymorphism*. Typically a function is declared virtual in the base class, and other functions with the same name are declared in derived classes.

The use of one or more pure virtual functions in a class makes the class *abstract*, which means that no objects can be instantiated from it.

A `friend` function can access a class's private data, even though it is not a member function of the class. This is useful when one function must have access to two or more unrelated classes and when an overloaded operator must use, on its left side, a value of a class other than the one of which it is a member. `friends` are also used to facilitate functional notation.

A `static` function is one that operates on the class in general, rather than on objects of the class. In particular it can operate on static variables. It can be called with the class name and scope-resolution operator.

The assignment operator = can be overloaded. This is necessary when it must do more than merely copy one object's contents into another. The copy constructor, which creates copies during initialization, and also when arguments are passed and returned by value, can also be overloaded. This is necessary when the copy constructor must do more than simply copy an object.

The `this` pointer is predefined in member functions to point to the object of which the function is a member. The `this` pointer is useful in returning the object of which the function is a member.

The `dynamic_cast` operator plays several roles. It can be used to determine what type of object a pointer points to, and, in certain situations, it can change the type of a pointer. The `typeid` operator can discover certain information about an object's class, such as its name.

The UML object diagram shows the relationship of a group of objects at a specific point in a program's operation.

Questions

Answers to these questions can be found in Appendix G.

1. Virtual functions allow you to
   a. create an array of type pointer-to-base class that can hold pointers to derived classes.
   b. create functions that can never be accessed.
   c. group objects of different classes so they can all be accessed by the same function code.
   d. use the same function call to execute member functions of objects from different classes.
2. True or false: A pointer to a base class can point to objects of a derived class.
3. If there is a pointer `p` to objects of a base class, and it contains the address of an object of a derived class, and both classes contain a nonvirtual member function, `ding()`, then the statement `p->ding();` will cause the version of `ding()` in the ________ class to be executed.
4. Write a declarator for a virtual function called `dang()` that returns type `void` and takes one argument of type `int`.
5. Deciding—after a program starts to execute—what function will be executed by a particular function call statement is called ________.
6. If there is a pointer, `p`, to objects of a base class, and it contains the address of an object of a derived class, and both classes contain a virtual member function, `ding()`, the statement `p->ding();` will cause the version of `ding()` in the ________ class to be executed.

7. Write the declaration for a pure virtual function called `aragorn` that returns no value and takes no arguments.
8. A pure virtual function is a virtual function that
   a. causes its class to be abstract.
   b. returns nothing.
   c. is used in a base class.
   d. takes no arguments.
9. Write the definition of an array called `parr` of 10 pointers to objects of class `dong`.
10. An abstract class is useful when
    a. no classes should be derived from it.
    b. there are multiple paths from one derived class to another.
    c. no objects should be instantiated from it.
    d. you want to defer the declaration of the class.
11. True or false: A `friend` function can access a class's private data without being a member of the class.
12. A `friend` function can be used to
    a. mediate arguments between classes.
    b. allow access to classes whose source code is unavailable.
    c. allow access to an unrelated class.
    d. increase the versatility of an overloaded operator.
13. Write the declaration for a `friend` function called `harry()` that returns type `void` and takes one argument of class `george`.
14. The keyword `friend` appears in
    a. the class allowing access to another class.
    b. the class desiring access to another class.
    c. the private section of a class.
    d. the public section of a class.
15. Write a declaration that, in the class in which it appears, will make every member of the class `harry` a `friend` function.
16. A static function
    a. should be called when an object is destroyed.
    b. is closely connected to an individual object of a class.
    c. can be called using the class name and function name.
    d. is used when a dummy object must be created.

17. Explain what the default assignment operator = does when applied to objects.
18. Write a declaration for an overloaded assignment operator in class `zeta`.
19. An assignment operator might be overloaded to
    a. help keep track of the number of identical objects.
    b. assign a separate ID number to each object.
    c. ensure that all member data is copied exactly.
    d. signal when assignment takes place.
20. True or false: The user must always define the operation of the copy constructor.
21. The operation of the assignment operator and that of the copy constructor are
    a. similar, except that the copy constructor creates a new object.
    b. similar, except that the assignment operator copies member data.
    c. different, except that they both create a new object.
    d. different, except that they both copy member data.
22. Write the declaration of a copy constructor for a class called `Bertha`.
23. True or false: A copy constructor could be defined to copy only part of an object's data.
24. The lifetime of a variable that is
    a. local to a member function coincides with the lifetime of the function.
    b. global coincides with the lifetime of a class.
    c. nonstatic member data of an object coincides with the lifetime of the object.
    d. static in a member function coincides with the lifetime of the function.
25. True or false: There is no problem with returning the value of a variable defined as local within a member function so long as it is returned by value.
26. Explain the difference in operation between these two statements.
    ```
    person p1(p0);
    person p1 = p0;
    ```
27. A copy constructor is invoked when
    a. a function returns by value.
    b. an argument is passed by value.
    c. a function returns by reference.
    d. an argument is passed by reference.
28. What does the `this` pointer point to?
29. If, within a class, `da` is a member variable, will the statement `this.da=37;` assign 37 to `da`?

30. Write a statement that a member function can use to return the entire object of which it is a member, without creating any temporary objects.

31. An object rectangle in an object diagram represents

    a. a general group of objects.

    b. a class.

    c. an instance of a class.

    d. all the objects of a class.

32. The lines between objects in a UML object diagram are called _________.

33. True or false: object A may relate to object B at one time but not at another.

34. Object diagrams show

    a. which objects exist at a point in time.

    b. which objects are communicating at a point in time.

    c. which objects participate in a particular behavior of the program.

    d. which objects have operations (member functions) that call objects of other classes.

Exercises

Answers to starred exercises can be found in Appendix G.

*1. Imagine the same publishing company described in Exercise 1 in Chapter 9 that markets both book and audiocassette versions of its works. As in that exercise, create a class called `publication` that stores the title (a string) and price (type `float`) of a publication. From this class derive two classes: `book`, which adds a page count (type `int`); and `tape`, which adds a playing time in minutes (type `float`). Each of the three classes should have a `getdata()` function to get its data from the user at the keyboard, and a `putdata()` function to display the data.

    Write a `main()` program that creates an array of pointers to `publication`. This is similar to the VIRTPERS example in this chapter. In a loop, ask the user for data about a particular book or tape, and use `new` to create an object of type `book` or `tape` to hold the data. Put the pointer to the object in the array. When the user has finished entering the data for all books and tapes, display the resulting data for all the books and tapes entered, using a `for` loop and a single statement such as

    ```
    pubarr[j]->putdata();
    ```

    to display the data from each object in the array.

*2. In the `Distance` class, as shown in the FRENGL and FRISQ examples in this chapter, create an overloaded `*` operator so that two distances can be multiplied together. Make it a `friend` function so that you can use such expressions as

```
Wdist1 = 7.5 * dist2;
```

You'll need a one-argument constructor to convert floating-point values into `Distance` values. Write a `main()` program to test this operator in several ways.

*3. As we saw earlier, it's possible to make a class that acts like an array. The CLARRAY example shown here is a complete program that shows one way to create your own array class:

```
// clarray.cpp
// creates array class
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class Array                         //models a normal C++ array
   {
   private:
      int* ptr;                     //pointer to Array contents
      int size;                     //size of Array
   public:
      Array(int s)                  //one-argument constructor
         {
         size = s;                  //argument is size of Array
         ptr = new int[s];          //make space for Array
         }
      ~Array()                      //destructor
         { delete[] ptr; }
      int& operator [] (int j)      //overloaded subscript operator
         { return *(ptr+j); }
   };
////////////////////////////////////////////////////////////////
int main()
   {
   const int ASIZE = 10;            //size of array
   Array arr(ASIZE);                //make an array

   for(int j=0; j<ASIZE; j++)       //fill it with squares
      arr[j] = j*j;

   for(j=0; j<ASIZE; j++)           //display its contents
      cout << arr[j] << ' ';
   cout << endl;
   return 0;
   }
```

The output of this program is

```
0  1  4  9  16  25  36  49  64  81
```

Starting with CLARRAY, add an overloaded assignment operator and an overloaded copy constructor to the `Array` class. Then add statements such as

```
Array arr2(arr1);
```

and

```
arr3 = arr1;
```

to the `main()` program to test whether these overloaded operators work. The copy constructor should create an entirely new `Array` object with its own memory for storing array elements. Both the copy constructor and the assignment operator should copy the contents of the old `Array` object to the new one. What happens if you assign an `Array` of one size to an `Array` of a different size?

4. Start with the program of Exercise 1 in this chapter, and add a member function of type `bool` called `isOversize()` to the `book` and `tape` classes. Let's say that a book with more than 800 pages, or a tape with a playing time longer than 90 minutes (which would require two cassettes), is considered oversize. You can access this function from `main()` and display the string "Oversize" for oversize books and tapes when you display their other data. If `book` and `tape` objects are to be accessed using pointers to them that are stored in an array of type `publication`, what do you need to add to the `publication` base class? Can you instantiate members of this base class?

5. Start with the program of Exercise 8 in Chapter 8, which overloaded five arithmetic operators for money strings. Add the two operators that couldn't be overloaded in that exercise. These operations

   ```
   long double * bMoney  // number times money
   long double / bMoney  // number divided by money
   ```

   require `friend` functions, since an object appears on the right side of the operator while a numerical constant appears on the left. Make sure that the `main()` program allows the user to enter two money strings and a floating-point value, and then carries out all seven arithmetic operations on appropriate pairs of these values.

6. As in the previous exercise, start with the program of Exercise 8 in Chapter 9. This time, add a function that rounds a `bMoney` value to the nearest dollar. It should be used like this:

   ```
   mo2 = round(mo1);
   ```

   As you know, amounts of $0.49 and less are rounded down, while those $0.50 and above are rounded up. A library function called `modfl()` is useful here. It separates a type `long double` variable into a fractional part and an integer part. If the fractional part is less than 0.50, return the integer part as is; otherwise add 1.0. In `main()`, test the function by sending it a sequence of `bMoney` amounts that go from less than $0.49 to more than $0.50.

7. Remember the PARSE program from Chapter 10? It would be nice to improve this program so it could evaluate expressions with real numbers, say type `float`, instead of single-digit numbers. For example

```
3.14159 / 2.0 + 75.25 * 3.333 + 6.02
```

As a first step toward this goal, you need to develop a stack that can hold both operators (type `char`) and numbers (type `float`). But how can you store two different types on a stack, which is basically an array? After all, type `char` and type `float` aren't even the same size. Could you store pointers to different types? They're the same size, but the compiler still won't allow you to store type `char*` and type `float*` in the same array. The only way two different types of pointers can be stored in the same array is if they are derived from the same base class. So we can encapsulate a `char` in one class and a `float` in another, and arrange for both classes to be derived from a base class. Then we can store both kinds of pointers in an array of pointers to the base class. The base class doesn't need to have any data of its own; it can be an abstract class from which no objects will be instantiated.

Constructors can store the values in the derived classes in the usual way, but you'll need to use pure virtual functions to get the values back out again. Here's a possible scenario:

```
class Token                    // abstract base class
   {
   public:
      virtual float getNumber()=0;    // pure virtual functions
      virtual char getOperator()=0;
   };
class Operator : public Token
   {
   private:
      char oper;               // operators +, -, *, /
   public:
      Operator(char);          // constructor sets value
      char getOperator();      // gets value
      float getNumber();       // dummy function
   };
class Number : public Token
   {
   private:
      float fnum;              // the number
   public:
      Number(float);           // constructor sets value
      float getNumber();       // gets value
      char getOperator();      // dummy function
   };

Token* atoken[100];            // holds types Operator* and Number*
```

Base-class virtual functions need to be instantiated in all derived classes, or the classes themselves become abstract. Thus the `Operand` class needs a `getNumber()` function, even though it doesn't store a number, and the `Number` class needs `getOperand()`, even though it doesn't store an operand.

Expand this framework into a working program by adding a `Stack` class that holds `Token` objects, and a `main()` that pushes and pops various operators (such as `+` and `*`) and floating-point numbers (1.123) on and off the stack.

8. Let's put a little twist into the HORSE example of Chapter 10 by making a class of extra-competitive horses. We'll assume that any horse that's ahead by the halfway point in the race starts to feel its oats and becomes almost unbeatable. From the horse class, derive a class called `comhorse` (for competitive horse). Overload the `horse_tick()` function in this class so that each horse can check if it's the front-runner and if there's another horse close behind it (say 0.1 furlong). If there is, it should speed up a bit. Perhaps not enough to win every time, but enough to give it a decided advantage.

   How does each horse know where the other horses are? It must access the memory that holds them, which in the HORSE program is `hArray`. Be careful, however. You want to create comhorses, not horses. So the `comhorse` class will need to overload `hArray`. You may need to derive a new track class, `comtrack`, to create the comhorses.

   You can continuously check if your horse is ahead of the (otherwise) leading horse, and if it's by a small margin, accelerate your horse a bit.

9. Exercise 4 in Chapter 10 involved adding an overloaded destructor to the `linklist` class. Suppose we fill an object of such a destructor-enhanced class with data, and then assign the entire class with a statement such as

   ```
   list2 = list1;
   ```

   using the default assignment operator. Now, suppose we later delete the `list1` object. Can we still use `list2` to access the same data? No, because when `list1` was deleted, its destructor deleted all its links. The only data actually contained in a `linklist` object is a pointer to the first link. Once the links are gone, the pointer in `list2` becomes invalid, and attempts to access the list lead to meaningless values or a program crash.

   One way to fix this is to overload the assignment operator so that it copies all the data links, as well as the `linklist` object itself. You'll need to follow along the chain, copying each link in turn. As we noted earlier, you should overload the copy constructor as well. To make it possible to delete `linklist` objects in `main()`, you may want to create them using pointers and `new`. That makes it easier to test the new routines. Don't worry if the copy process reverses the order of the data.

   Notice that copying all the data is not very efficient in terms of memory usage. Contrast this approach with that used in the STRIMEM example in Chapter 10, which used only one set of data for all objects, and kept track of how many objects pointed to this data.

10. Carry out the modification, discussed in Exercise 7, to the PARSE program of Chapter 10. That is, make it possible to parse expressions containing floating-point numbers. Combine the classes from Exercise 7 with the algorithms from PARSE. You'll need to operate on pointers to tokens instead of characters. This involves statements of the kind

```
Number* ptrN = new Number(ans);
s.push(ptrN);
```

and

```
Operator* ptrO = new Operator(ch);
s.push(ptrO);
```