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.

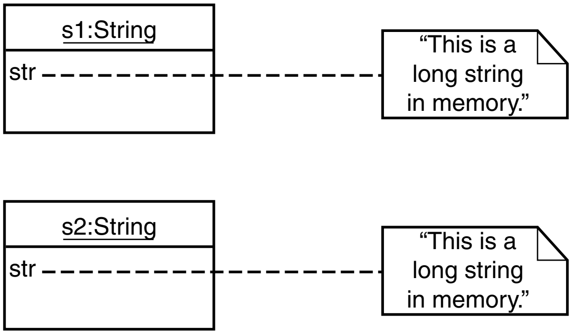

FIGURE 11.4
UML object diagram: replicating strings.

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.

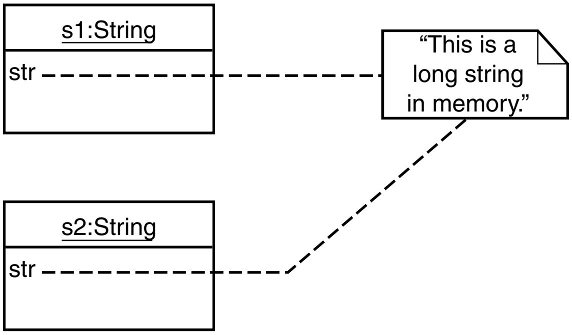

FIGURE 11.5
UML object diagram: replicating pointers to strings.

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.

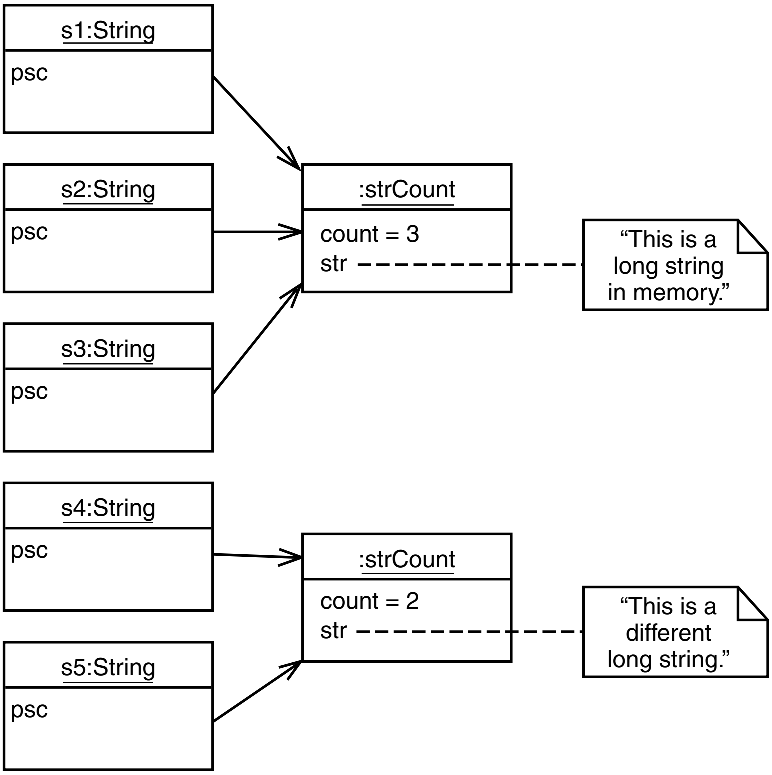

FIGURE 11.6

`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.

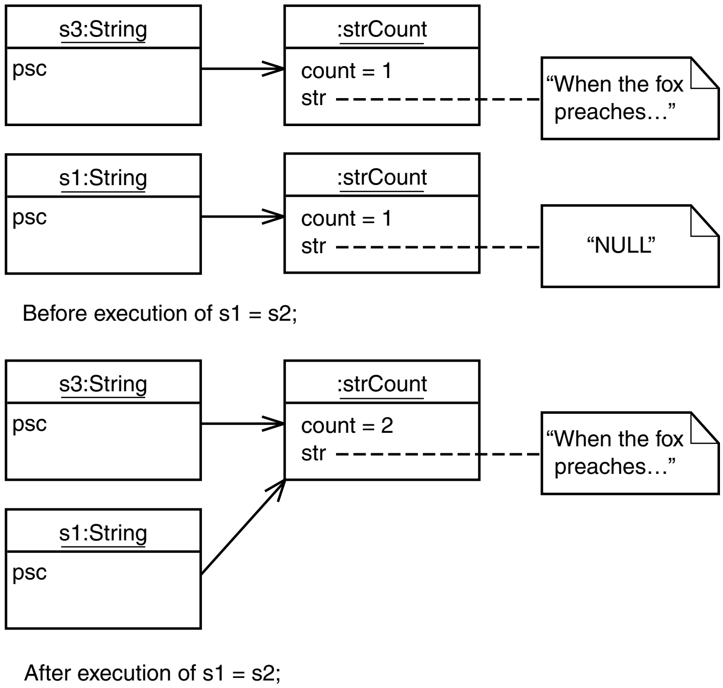

FIGURE 11.7

Assignment operator in STRIMEM.

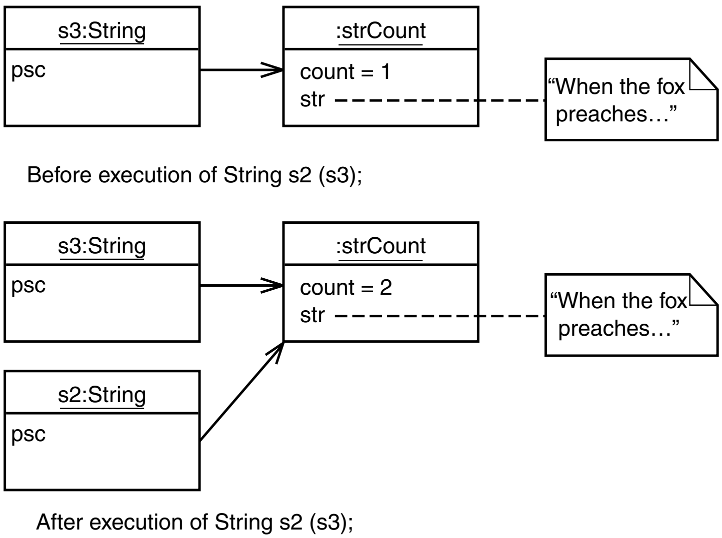

FIGURE 11.8

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 `deleted`). 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
      {  }
   };
```