2.10 Returning Objects

Just as objects can be passed to functions, so functions can return objects. To return an object, first declare the function as returning a class type. Second, return an object of that type by using the normal return statement. Here is an example of a function that returns an object:

```
// Returning an object.
#include <iostream.h>
#include <string.h>
class sample
{
     char s[80];
  public:
     void show() { cout<< s << "\n"; }
     void set(char *str) { strcpy(s, str); }
};
// Return an object of type sample.
sample input()
{
    char instr[80];
    sample str;

    cout<< "Enter a string: ";
    cin>>instr;

    str.set(instr);
    return str;
}

int main()
{
  sample ob;
  // assign returned object to ob
  ob = input();
  ob.show();
  return 0;
}
```

In this example, input( ) creates a local object called str and

then reads a string from the keyboard. This string is copied into str.s, and then str is returned by the function. This object is then assigned to ob inside main( ) after it is returned by input( ).

2.11 A Potential Problem When Returning Objects

There is one important point to understand about returning objects from functions: When an object is returned by a function, a temporary object is automatically created, which holds the return value. It is this object that is actually returned by the function. After the value has been returned, this object is destroyed. The destruction of this temporary object may cause unexpected side effects in some situations. For example,if the object returned by the function has a destructor that frees dynamically allocated memory, that memory will be freed even though the object that receives the return value is still using it. Consider the following incorrect version of the preceding program:

```
// An error generated by returning an object.
#include <iostream.h>
#include <string.h>
#include <stdlib.h>
class sample
{
    char *s;
public:
     sample() { s = 0; }
     ~sample()
     { if(s) delete [] s; cout<< "Freeing
  s\n"; }
     void show() { cout<< s << "\n"; }
     void set(char *str);
};
// Load a string.
void sample::set(char *str)
{
  s = new char[strlen(str)+1];
  strcpy(s, str);
}
```

```
// Return an object of type sample.
sample input()
{
   char instr[80];
   sample str;
   cout<< "Enter a string: ";
   cin>>instr;
   str.set(instr);
   return str;
}
int main()
{
   sample ob;
   // assign returned object to ob
   ob = input();  // This causes an error!!!!
   ob.show(); // displays garbage
   return 0;
}
```

The output from this program is shown here:

```
Enter a string: Hello
Freeing s Freeing s garbage here Freeing s
```

Notice that sample's destructor is called three times! First, it is called when the local object str goes out of scope upon the return of input( ). The second time ~sample( ) is called is when the temporary object returned by input( ) is destroyed. When an object is returned from a function, an invisible (to you) temporary object is automatically generated, which holds the return value. In this case, the object is simply a bitwise copy of str, which is the return value of the function. Therefore, after the function has returned, the temporary object's destructor is executed.

Because the memory holding the string entered by the user has already been freed (twice!), garbage is displayed when show( ) is called. (Depending upon how your compiler implements dynamic allocation, you may not see garbage output, but the error is still present.) Finally, the destructor for object ob, inside main( ), is called when the program terminates.

The trouble is that, in this situation, the first time the destructor executes, the memory allocated to hold the string obtained by input( ) is freed. Thus, not only do the othertwo calls to sample's destructor try to free an already released piece of dynamic memory, but they may also damage the dynamic allocation system in the process.

The key point to understand from this example is that when an object is returned froma function, the temporary object holding the return value will have its destructor called. Thus, you should avoid returning objects in which this situation can be harmful. One solution is to return either a pointer or a reference. However, this is not always feasible. Another way to solve this problem involves the use of a copy constructor, which is described next.

2.12 Creating and Using a Copy Constructor

One of the more important forms of an overloaded constructor is the copy constructor. As earlier examples have shown, problems can occur when an object is passed to, or returned from, a function. As you will learn in this section, one way to avoid these problems is to define a copy constructor, which is a special type of overloaded constructor.

To begin, let's restate the problems that a copy constructor is designed to solve. When an object is passed to a function, a bitwise (i.e., exact) copy of that object is made and given to the function parameter that receives the object. However, there are cases in which this identical copy is not desirable. For example, if the object contains a pointer to allocated memory, then the copy will point to the same memory as does the original object.

Therefore, if the copy makes a change to the contents of this memory, it willbe changed for the original object, too! Furthermore, when the function terminates, the copy will be destroyed, thus causing its destructor to be called. This may also have undesired effects on the original object.

A similar situation occurs when an object is returned by a function. The compiler will generate a temporary object that holds a copy of the value returned by the function. (This is done automatically, and is beyond your control.) This temporary object goes out of scope once the value is returned to the calling routine, causing the temporary object's destructor to be called. However, if the destructor destroys something needed by the calling routine, trouble will follow.

At the core of these problems is the creation of a bitwise copy of the object. To prevent them, you need to define precisely what occurs when a copy of an object is made so that you can avoid undesired side effects. The way you accomplish this is by creatinga copy constructor. Before we explore the use of the copy constructor, it is important for you to understandthat C++ defines two distinct types of situations in which the value of one object is given to another. The first situation is assignment. The second situation is initialization,

which can occur three ways:

- ***When one object explicitly initializes another, such as in a declaration***
- ***When a copy of an object is passed as a parameter to a function***
- ***When a temporary object is generated (most commonly, as a return value)***

The copy constructor applies only to initializations. It does not apply to assignments.The most common form of copy constructor is shown here:

```
classname (constclassname&obj) {
// body of constructor
}
```

Here, obj is a reference to an object that is being used to initialize another object. For example, assuming a class called myclass, and y as an object of type myclass, then the following statements would invoke the myclass copy constructor:

```
myclass x = y; // y explicitly initializing x
func1(y);    // y passed as a parameter
y = func2(); // y receiving a returned object
```

In the first two cases, a reference to y would be passed to the copy constructor. In the third, a reference to the object returned by func2( ) would be passed to the copy constructor.To fully explore the value of copy constructors, let’s see how they impact each of the three situations to which they apply.

2.13 Copy Constructors and Parameters

When an object is passed to a function as an argument, a copy of that object is made. If a copy constructor exists, the copy constructor is called to make the copy. Here is a program that uses a copy constructor to properly handle objects of type myclass when they are passed to a function. (This is a corrected version of the incorrect program shown earlier in this chapter.)

```
// Use a copy constructor to construct a parameter.
#include <iostream.h>
#include <stdlib.h>
class myclass
{
     int *p;
  public:
     myclass(int i); // normal constructor
     myclass(constmyclass&ob);     //     copy
    constructor
     ~myclass();
     intgetval() { return *p; }
};
// Copy constructor.
myclass::myclass(constmyclass&obj)
{
  p = new int;
  *p = *obj.p; // copy value
  cout<< "Copy constructor called.\n";
}
// Normal Constructor.
myclass::myclass(int i)
{
  cout<< "Allocating p\n";
  p = new int;
  *p = i;
}
myclass::~myclass()
{
cout<< "Freeing p\n";
```

```
delete p;
}
// This function takes one object parameter.
void display(myclassob)
{
    cout<<ob.getval() << '\n';
}
int main()
{
   myclass a(10);
   display(a);
   return 0;
}
```

This program displays the following output:

```
Allocating p
Copy constructor called.
10
Freeing p
Freeing p
```

Here is what occurs when the program is run: When a is created inside main( ), the normal constructor allocates memory and assigns the address of that memory to a.p. Next, a is passed to ob of display( ). When this occurs, the copy constructor is called, and a copy of a is created. The copy constructor allocates memory for the copy, and a pointer to that memory is assigned to the copy's p member. Next, the value stored at the original object's p is assigned to the memory pointed to by the copy's p. Thus, the areas of memory pointed to by a.p and ob.p are separate and distinct, but the valuesthat they point to are the same. If the copy constructor had not been created, then the default bitwise copy would have caused a.p and ob.p to point to the same memory.

When display( ) returns, ob goes out of scope. This causes its destructor to be called, which frees the memory pointed to by ob.p. Finally, when main( ) returns, a goes out of scope, causing its destructor to free a.p. As you can see, the use of the copy constructor has eliminated the destructive side effects associated with passing an object to a function.