Copy construct

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.

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.

Copy Constructors and Initializations

The copy constructor is also invoked when one object is used to initialize another. Examine this sample program:

```
// The copy constructor is called for initialization.
   #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&ob)
   {
      p = new int;
      *p = *ob.p; // copy value
      cout<< "Copy constructor allocating p.\n";
   }

   // Normal constructor.
   myclass::myclass(int i)
   {
      cout<< "Normal constructor allocating p.\n";
      p = new int;
      *p = i;
   }
   myclass::~myclass()
   {
      cout<< "Freeing p\n";
      delete p;
   }
```

```
int main()
{
myclass a(10); // calls normal constructor
myclass b = a; // calls copy constructor
return 0;
}
```

This program displays the following output:

```
Normal constructor allocating p.
Copy constructor allocating p.
Freeing p
Freeing p
```

As the output confirms, the normal constructor is called for object a. However, when a is used to initialize b, the copy constructor is invoked. The use of the copy constructor ensures that b will allocate its own memory. Without the copy constructor, b would simply be an exact copy of a, and a.p would point to the same memory as b.p.

Keep in mind that the copy constructor is called only for initializations. For example, the following sequence does not call the copy constructor defined in the preceding program:

```
myclass a(2), b(3);
// ... b = a;
```

In this case, b = a performs the assignment operation, not a copy operation.

Using Copy Constructors When an Object Is Returned

The copy constructor is also invoked when a temporary object is created as the result of a function returning an object. Consider this short program:

```
#include <iostream>
class myclass {
public:
   myclass() { cout<< "Normal constructor.\n"; }
   myclass(constmyclass&obj)
   { cout<< "Copy constructor.\n"; }
};

myclass f()
{
   myclassob; // invoke normal constructor
   return ob; // implicitly invoke copy constructor
}
int main()
{
myclass a; // invoke normal constructor
   a = f(); // invoke copy constructor
   return 0;
}
```

This program displays the following output:

```
Normal constructor.
Normal constructor.
Copy constructor.
```

Here, the normal constructor is called twice: once when a is created inside main( ), and once when ob is created inside f( ). The copy constructor is called when the temporary object is generated as a return value from f( ).Although copy constructors may seem a bit esoteric at this point, virtually every real-world class will require one, due to the side effects that often result from the default bitwise copy.

The this Keyword

Each time a member function is invoked, it is automatically passed a pointer, called ***this***, to the object on which it is called. The this pointer is an implicit parameter toall member functions. Therefore, inside a member function, this may be used to refer to the invoking object.As you know, a member function can directly access the private data of its class. For example, given this class,

```
class cl
{int i;
void f() { ... };
// ...
};
```

inside f( ), the following statement can be used to assign i the value10:

```
i = 10;
```

In actuality, the preceding statement is shorthand for this one:

```
this->i = 10;
```

To see how the this pointer works, examine the following short program:

```
#include <iostream.h>
class cl
{int i;
  public:
     void load_i(intval) { this->i = val; }
     // same as i = val
     intget_i() { return this->i; }// same as return i
} ;
int main()
{cl o;
  o.load_i(100);
  cout<<o.get_i();
return 0;
}
```

This program displays the number

```
100.
```