2.14 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.

2.15 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.

2.16 Copy Constructors—Is There a Simpler Way?

As has been stated several times in this book, C++ is a very powerful language. It is also a very large, and at times, complex language. Copy constructors are a feature that many programmers point to as a prime example of this complexity because it

is a non-intuitive feature. Newcomers often do not immediately understand why the copy constructor is important, nor is it always obvious to the novice when a copy constructor is needed and when one isn't. This situation often gives rise to the question "Isn't there a better way?" The answer is both Yes and No!

Languages such as Java and C# do not have copy constructors because neither language makes bitwise copies of an object. This is because both Java and C# dynamically allocate all objects and you operate on those objects exclusively through references. Thus, no copies of an object are made when passing one as a parameter or returning one from a function.

The fact that neither Java nor C# require copy constructors streamlines those languages, but it comes at a price. Operating on objects exclusively through references, rather directly as you can in C++, imposes limitations on the type of operations you can perform. Furthermore, because of their exclusive use of object references, in Java and C# you cannot precisely specify when an object will be destroyed. In C++, an objectis always destroyed when it goes out of scope.Because C++ gives you, the programmer, complete control, it is a bit more complicated language than are Java and C#. This is the price of programming power.

2.17 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.
```

Mahmoud HilalFarhan Al Anbar University
College of Computer Sciences and Information Technology