CHAPTERFOUR

Constructors, Destructors, and Inheritance

Constructors, Destructors, and Inheritance

There are two important questions that arise relative to constructors and destructors when inheritance is involved. First, when are base class and derived class constructors and destructors called? Second, how can parameters be passed to a base class constructor? This section answers these questions.

When Constructors and Destructors Are Executed

It is possible for a base class, a derived class, or both, to contain a constructor and/or destructor. It is important to understand the order in which these are executed when an object of a derived class comes into existence and when it goes out of existence.Examine this short program:

```
#include <iostream.h>
class base
{
   public:
      base() { cout<< "Constructing base\n"; }
      ~base() { cout<< "Destructing base\n"; }
};
class derived: public base
{
   public:
      derived() { cout<< "Constructing derived\n"; }
      ~derived() { cout<< "Destructing derived\n"; }
};
intmain()
{
   derived ob;
   // do nothing but construct and destruct ob
   return 0;
}
```

As the comment in main( ) indicates, this program simply constructs and then destroys an object called ob, which is of class derived. When executed, this program displays:

```
Constructing base
Constructing derived
Destructing derived
Destructing base
```

As you can see, the constructor of base is executed, followed by the constructor of derived. Next (since ob is immediately destroyed in this program), the destructor of derived is called, followed by that of base.

The results of the foregoing experiment can be generalized as follows: When an object of a derived class is created, the base class constructor is called first, followed by the constructor for the derived class. When a derived object is destroyed, its destructor is called first, followed by the destructor for the base class. Put differently, constructors are executed in the order of their derivation. Destructors are executed in reverse order of derivation.

If you think about it, it makes sense that constructor functions are executed in the order of their derivation. Because a base class has no knowledge of any derived class, any initialization it needs to perform is separate from, and possibly prerequisite to, any initialization performed by the derived class. Therefore, it must be executed first.

Likewise, it is quite sensible that destructors be executed in reverse order of derivation. Since the base class underlies a derived class, the destruction of the base class implies the destruction of the derived class. Therefore, the derived destructor must be called before the object is fully destroyed.In the case of a large class hierarchy (i.e., where a derived class becomes the base class for another derived class), the general rule applies: Constructors are called in order of derivation, destructors in reverse order. For example, this program

```
#include <iostream.h>
class base
{
   public:
      base() { cout<< "Constructing base\n"; }
      ~base() { cout<< "Destructing base\n"; }
};
class derived1 : public base
{
   public:
      derived1() { cout<< "Constructing derived1\n"; }
      ~derived1() { cout<< "Destructing derived1\n"; }
};
class derived2: public derived1
{
   public:
      derived2() { cout<< "Constructing derived2\n"; }
      ~derived2() { cout<< "Destructing derived2\n"; }
};
intmain()
{
  derived2 ob;// construct and destruct ob
  return 0;
}
```

displays this output:

```
Constructing base
Constructing derived1
Constructing derived2
Destructing derived2
Destructing derived1
Destructing base
```

The same general rule applies in situations involving multiple base classes. For example, this program

```
#include <iostream.h>
class base1
{
   public:
```

```
      base1() { cout<< "Constructing base1\n"; }
      ~base1() { cout<< "Destructing base1\n"; }
};
class base2
{
   public:
      base2() { cout<< "Constructing base2\n"; }
      ~base2() { cout<< "Destructing base2\n"; }
};
class derived: public base1, public base2
{
   public:
      derived() { cout<< "Constructing derived\n"; }
      ~derived() { cout<< "Destructing derived\n"; }
};
intmain()
{
   derived ob;// construct and destruct ob
   return 0;
}
```

produces this output:

```
Constructing base1
Constructing base2
Constructing derived
Destructing derived
Destructing base2
Destructing base1
```

As you can see, constructors are called in order of derivation, left to right, as specified in derived's inheritance list. Destructors are called in reverse order, right to left. This means that if base2 were specified before base1 in derived's list, as shown here:

```
class derived: public base2, public base1 {
```

then the output of the preceding program would look like this:

```
Constructing base2
Constructing base1
Constructing derived
```

```
Destructing derived
Destructing base1
Destructing base2
```