OOPS

Constructors, destructors, and inheritance

It is possible for the base class, the derived class, or both to have constructor and/or destructor functions. When a base class and a derived class both have constructor and destructor functions, the constructor functions are executed in order of derivation. **The destructor functions are executed in reverse order.**

That is, the base constructor is executed before the constructor in the derived class.

The reverse is true for destructor functions: **the destructor in the derived class is executed before the base class destructor**.

So far, you have passed arguments to either the derived class or base class constructor. When only the derived class takes an initialization, arguments are passed to the derived class constructor in the normal fashion.

However, if you need to pass an argument to the constructor of the base class, a little more effort is needed:

All necessary arguments to both the base class and derived class are passed to the derived class constructor.

Using an expanded form of the derived class' constructor declaration, you then pass the appropriate arguments along to the base class

The syntax for passing an argument from the derived class to the base class is as

```
derived-constructor(arg-list) : base(arg-list) {
     // body of the derived class constructor
}
```

Here base is the name of the base class. It is permissible for both the derived class and the base class to use the same argument. It is also possible for the derived class to ignore all arguments and just pass them along to the base.

```
// Illustrates when base class and derived class
// constructor and destructor functions are executed
#include <iostream>
using namespace std;
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"; }
};
int main() {
   derived obj;
   return 0;
}
```

This program displays

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

In the following example both the derived class and the base class take arguments:

```
#include <iostream>
using namespace std;
class base {
   int i;

  public:
   base(int n) {
```

```
      cout << "Constructing base\n";
      i = n;
   }
   ~base() { cout << "Destructing base\n"; }
   void showi() { cout << i << "\n"; }
};
class derived : public base {
   int j;

  public:
   derived(int n) : base(n) {
      // pass argument to the base class
      cout << "Constructing derived\n";
      j = n;
   }
   ~derived() { cout << "Destructing derived\n"; }
   void showj() { cout << j << "\n"; }
};
int main() {
   derived o(10);
   o.showi();
   o.showj();
   return 0;
}
```

Pay special attention to the declaration of the derived class constructor. Notice how the parameter n (which receives the initialization argument) is both used by derived() and base().