Lecture 16

Object-Oriented Programming XII

Reusing the copy-control members of the base class
for the definition of the derived class. Copy control members =
{copy constructor, assignment operator, destructor}

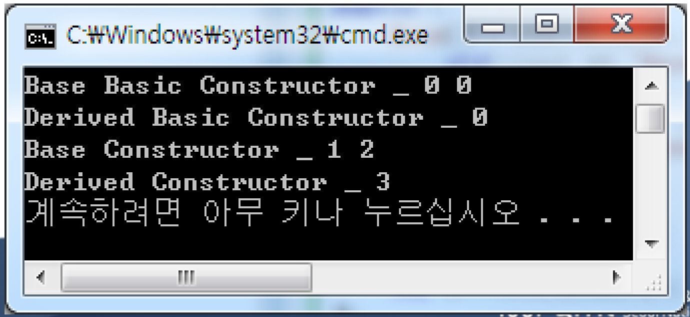
Prof. Hyeong-Seok Ko
Seoul National University
Graphics & Media Lab

Contents

How Calls to Base Class Constructor are Made?

- If there isn't any explicit call in the constructor initializer, a call to the no-argument constructor of the base class is automatically made.
- An explicit call with arguments can be made in the constructor initializer.

```cpp
class Base {
public :
   Base() : public_base_var(0), private_base_var(0) {
      std::cout << "Base Basic Constructor _ " << public_base_var << " " << private_base_var << std::endl;
   }
   Base(int a, int b) : public_base_var(a), private_base_var(b) {
      std::cout << "Base Constructor _ " << public_base_var << " " << private_base_var << std::endl;
   }

   int public_base_var;
private :
   int private_base_var;
};

class Derived : public Base {
public :
   Derived() : public_derived_var(0) {
      std::cout << "Derived Basic Constructor _ " << public_derived_var << std::endl;
   }
   Derived(int a, int b, int c) : Base(a,b), public_derived_var(c) {
      std::cout << "Derived Constructor _ " << public_derived_var << std::endl;
   }

   int public_derived_var;
};

void main() {
   Derived derived_0;
   Derived derived_1(1,2,3);
}
```

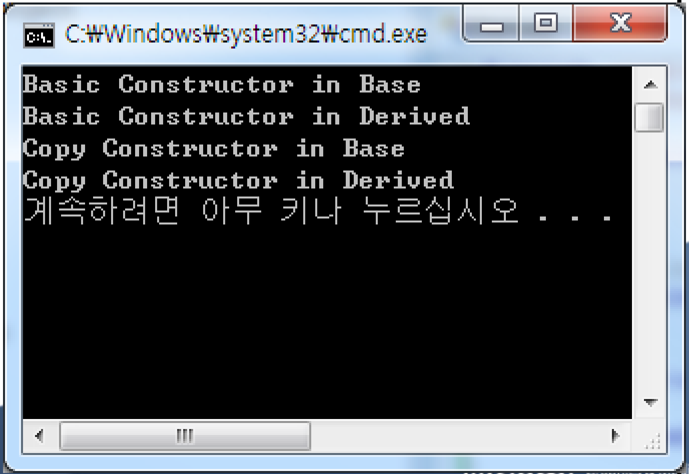


```
Base Basic Constructor _ 0 0
Derived Basic Constructor _ 0
Base Constructor _ 1 2
Derived Constructor _ 3
계속하려면 아무 키나 누르십시오 . . .
```

Defining Copy Constructor of the Derived Class

- You can copy the base part of the object by making a call `Base(derived)` to the base-class copy constructor.
  - Although the call is made with the derived class object argument, in this case, only the Base class portion is used.
- If such a call is not explicitly made, a call to the base-class constructor (no-parameter version) is automatically made. Note that the call is made to the constructor not the copy constructor.

```cpp
class Base {
public :
   Base() { cout << “Basic Constructor in Base" << endl; }
   Base(const Base& base) { cout << "Copy Constructor in Base" << endl; a = base.a; }

protected :
   int a;
};

class Derived : public Base {
public :
   Derived() { size = 10; ptr = new int[size]; cout << “Basic Constructor in Derived" << endl; }

   // Correct definition of copy constructor in derived class
   Derived(const Derived& derived) : Base(derived) {
      cout << "Copy Constructor in Derived" << endl;

      size = derived.size;
      ptr = new int[size];
      for(size_t i=0;i<size;++i)
         ptr[i] = derived.ptr[i];
   }

   size_t    size;
   int *     ptr;
};

void main() {
   Derived derived_0;
   Derived derived_1(derived_0);
}
```



C:\Windows\system32\cmd.exe

```
Basic Constructor in Base
Basic Constructor in Derived
Copy Constructor in Base
Copy Constructor in Derived
계속하려면 아무 키나 누르십시오 . . .
```

Defining Assignment Operator of the Derived Class

```
Derived& Derived::operator=(const Derived& rhs) {
   if(this != &rhs) {


      if(ptr != NULL) delete[] ptr;

      size = rhs.size;
      ptr = new int[size];
      for(size_t i=0;i<size;++i)
          ptr[i] = rhs.ptr[i];
   }
}
```

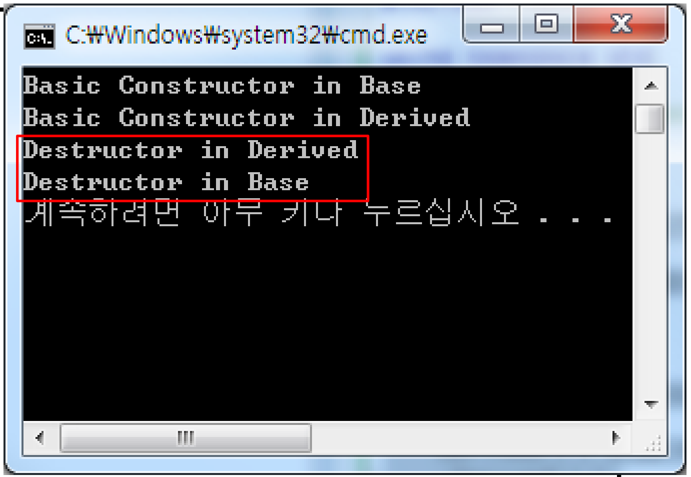
Defining Assignment Operator of the Derived Class

- Similarly to the case of the copy constructor, you can assign the base part of the object by making a call `Base::operator=(rhs);` to the base-class assignment operator.
- If such a call is not explicitly made, assignment of the base part does not occur.
- Therefore, in the implementation of the assignment, you should not forget to make the base part assignment explicitly.

```
Derived& Derived::operator=(const Derived& rhs) {
   if(this != &rhs) {
      Base::operator=(rhs);   // assignment of the base part

      if(ptr != NULL) delete[] ptr;

      size = rhs.size;
      ptr = new int[size];
      for(size_t i=0;i<size;++i)
          ptr[i] = rhs.ptr[i];
   }
}
```

Automatic Call of Base Class Destructor

```
class Base {
public :
   Base() { cout << "Basic Constructor in Base" << endl; }
   ~Base() {
      cout << "Destructor in Base" << endl;
   }
};

class Derived : public Base {
public :
   Derived() {
      size = 10; ptr = new int[size];
      cout << "Basic Constructor in Derived" << endl;
   }
   ~Derived() {
      // Each destructor does only what is necessary to clean up its own members
      delete[] ptr;
      cout << "Destructor in Derived" << endl;
    }

    size_t size;
    int *  ptr;
};

void main() {
    Derived derived_0;
}
```

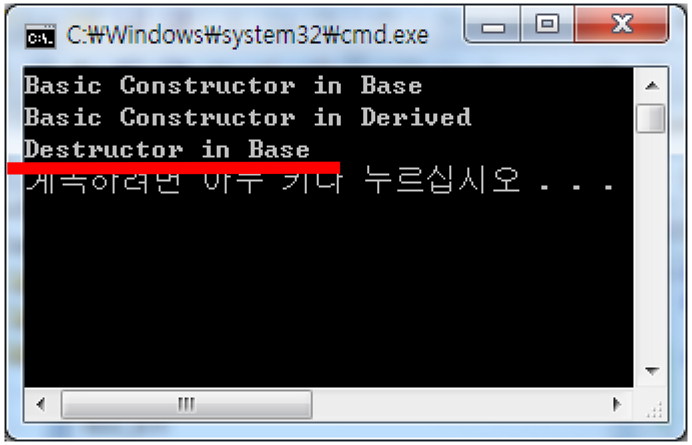


```
Basic Constructor in Base
Basic Constructor in Derived
Destructor in Derived
Destructor in Base
계속하려면 아무 키나 누르십시오 . . .
```

Potential Problems Associated with Destructors when Using Pointers

- If we delete a base pointer, then the members of the derived class may not be cleaned up, leading to a memory leak.

```cpp
class Base {
public :
   Base() { cout << "Basic Constructor in Base" << endl; }
   ~Base() { cout << "Destructor in Base" << endl; }
};

class Derived : public Base {
public :
   Derived() {
      size = 10; ptr = new int[size];
      cout << "Basic Constructor in Derived" << endl;
   }
   ~Derived() {
      delete[] ptr;
      cout << "Destructor in Derived" << endl;
   }

   size_t  size;
   int *   ptr;
};

void main() {
   Base * p = new Derived();
   delete p;
}
```



```
Basic Constructor in Base
Basic Constructor in Derived
Destructor in Base
계속하려면 아무 키나 누르십시오 . . .
```

Only the destructor of Base class is called.
Therefore ptr of derived class is not deleted!

Virtual Destructor

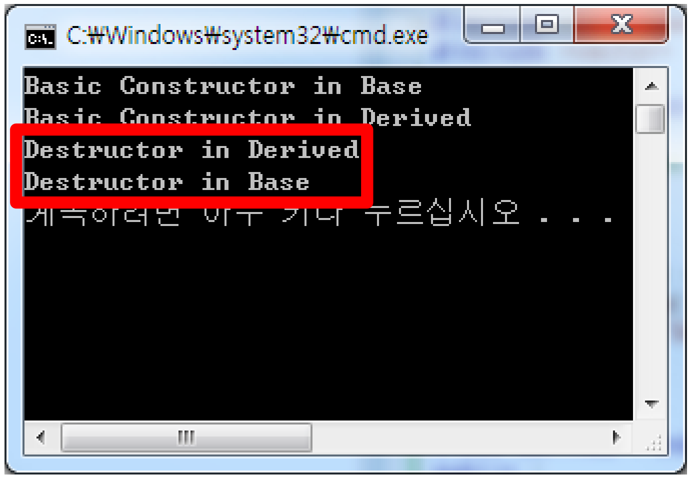
- To ensure that the proper destruction is done in the previous example, the destructor must be defined virtual.

```
class Base {
public :
   Base() { cout << "Basic Constructor in Base" << endl; }
   ~Base() { cout << "Destructor in Base" << endl; }
};

class Derived : public Base {
public :
   Derived() {
      size = 10; ptr = new int[size];
      cout << "Basic Constructor in Derived" << endl;
   }
   ~Derived() {
      delete[] ptr;
      cout << "Destructor in Derived" << endl;
   }

   size_t  size;
   int *   ptr;
};

void main() {
   Base * p = new Derived();
   delete p;
}
```

Virtual Destructor

- To ensure that the proper destruction is done in the previous example, the destructor must be defined virtual.

```
class Base {
public :
   Base() { cout << "Basic Constructor in Base" << endl; }
   virtual ~Base() { cout << "Destructor in Base" << endl; }
};

class Derived : public Base {
public :
   Derived() {
      size = 10; ptr = new int[size];
      cout << "Basic Constructor in Derived" << endl;
   }
   ~Derived() {
      delete[] ptr;
      cout << "Destructor in Derived" << endl;
   }

   size_t  size;
   int *   ptr;
};

void main() {
   Base * p = new Derived();
   delete p;
}
```


C:\Windows\system32\cmd.exe

```
Basic Constructor in Base
Basic Constructor in Derived
Destructor in Derived
Destructor in Base
계속하려면 아무 키나 누르십시오 . . .
```

The destructors of both Derived and Base classes are called, in that order.
Therefore ptr of Derived class is now successfully deleted!

Destructor vs (Copy Constructor & Assignment Op)

- Note that implementation of the destructor is different from that of the copy constructor or assignment operator.
  - The part destroying the members of its base objects does not need to be included in the definition, but are called automatically.
  - In this aspect, it is similar to the constructor.

Rule of Three and Inheritance

- Whether a class needs to define the copy-control members (i.e., copy constructor, assignment operator, and destructor) depends entirely on the class's own direct members; that decision does not need to consider the class's super classes.
  - It is because that issue in regard to the super class is already taken care of. (If it is not taken care of yet, it should be taken care of in defining the super class.)

```
class Base {
public :
   size_t  n;
   int *   ptr_Base;
};

class Derived : public Base {
public :
   float   a;
   int     b;
};

void main() {
   Base * p1 = new Base();
   Base * p2 = new Derived();
   Derived * p3 = new Derived();
   Derived p4;
   Derived p5(p4);
   delete p;
}
```

Copy-control members need to be defined explicitly.

Implicit copy-control members are all right.

Copy-control members need to be defined explicitly.

```
class Base {
public :
   size_t  n;
   int *   ptr_Base;
};

class Derived : public Base {
public :
   size_t  m;
   int *   ptr_Derived;
};

void main() {
   Base * p1 = new Base();
   Base * p2 = new Derived();
   Derived * p3 = new Derived();
   Derived p4;
   Derived p5(p4);
   delete p;
}
```