Lecture 17

Object-Oriented Programming XI

Multiple Inheritance

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

Contents

- Multiple Inheritance (17.3.1~17.3.4)
- Diamond Problem (17.3.5)

Multiple Inheritance

- In some cases, single inheritance is inadequate, either because it fails to model the situation properly or the model it imposes is unnecessarily complex.
- In these cases, **multiple inheritance** may model the application more directly.
- A multiply derived class inherits the properties of all its parents.

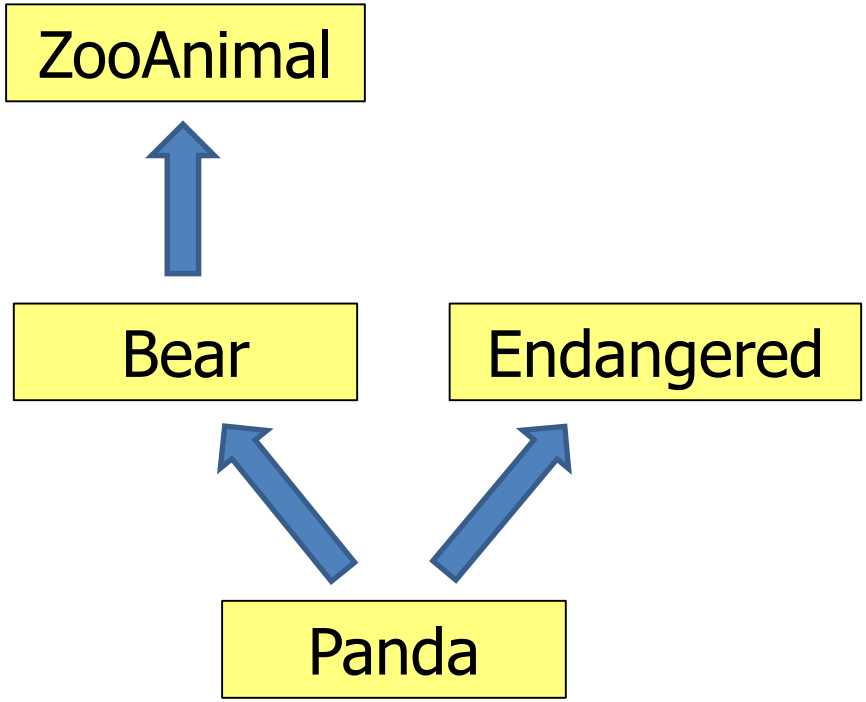
An Example of Multiple Inheritance

```
class ZooAnimal {};

class Endangered {};

class Bear : public ZooAnimal {};

class Panda : public Bear, public Endangered {};
```

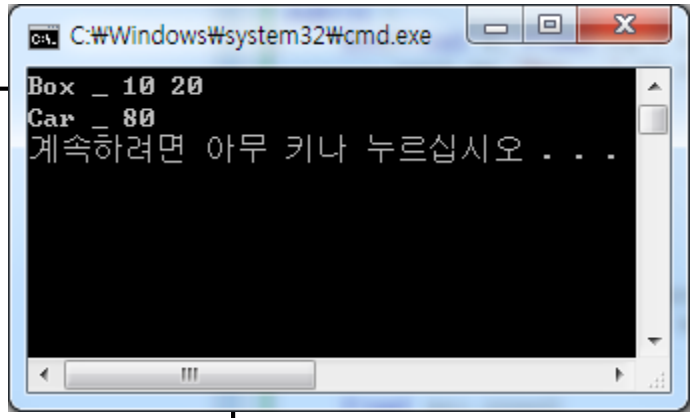


Construction of Multiply Derived Classes

- As is the case for inheriting from a single base class, a constructor of multiply derived class may pass values to its base class constructors in the constructor initializer.
- The base class constructors are invoked in the order in which they appear in the class derivation list.

```
class Box {
public :
   Box(float w, float h) : width(w), height(h) {
      cout << "Box _ " << width << " " << height << endl;
   }
   float width, height;
};
class Car {
public :
   Car(float msp) : max_speed(msp) {
      cout << "Car _ " << max_speed << endl;
   }
   float max_speed;
};
class BoxCar : public Box, public Car {
public :
   BoxCar(float w, float h, float msp) : Car(msp), Box(w,h) {}
};

void main() {
   BoxCar boxcar(10,20,80);
}
```



Constructor is invoked in this order

Can pass values in the constructor initializer

Destruction of Multiply Derived Classes

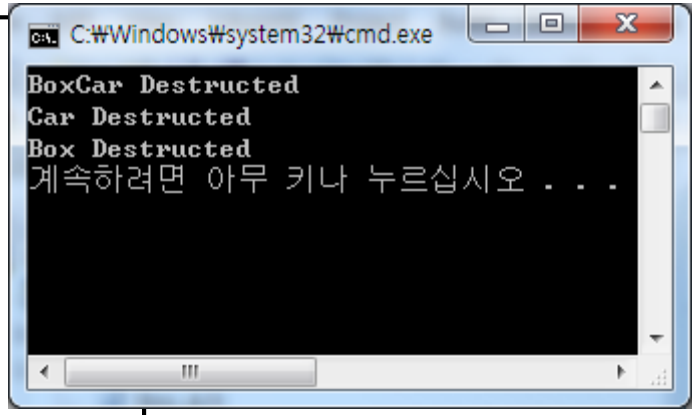
- Destructors are invoked in the reverse order from the order the constructors are invoked.

```
class Box {
public :
   Box() {}
   ~Box() { cout << "Box Destructed" << endl; }
};

class Car {
public :
   Car() {}
   ~Car() { cout << "Car Destructed" << endl; }
};

class BoxCar : public Box, public Car {
public :
   BoxCar() {}
   ~BoxCar() { cout << "BoxCar Destructed" << endl; }
};

void main() {
   BoxCar boxcar;
}
```



Use of Pointers or References of a Multiply Derived Classes

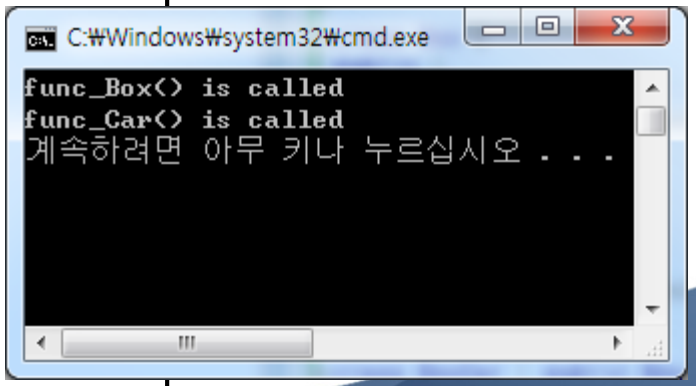
- An object of a multiply derived class can be pointed by a pointer or a reference variable of its base class.

```
class Box {
public :
   Box() {}
   void func_Box() { cout << "func_Box() is called" << endl; }
};
class Car {
public :
   Car() {}
   void func_Car() { cout << "func_Car() is called" << endl; }
};
class BoxCar : public Box, public Car {
public :
   BoxCar() {}
};

void main() {
   Box* box_ptr = new BoxCar();
   Car* car_ptr = new BoxCar();
   box_ptr->func_Box();
   car_ptr->func_Car();

   //box_ptr->func_Car();  // Compilation Error !

   Car car1; Car& car2 = car1;
   BoxCar car3; Car& car4 = car3;
}
```



Name Collisions

- Multiple inheritance can lead to ambiguities.

```
class Box {
public :
   Box() {}
   void func() { cout << "func() in Box is called" << endl; }
};

class Car {
public :
   Car() {}
   void func() { cout << "func() in Car is called" << endl; }
};

class BoxCar : public Box, public Car {
public :
   BoxCar() {}
   void func_2() {
      func();     // Compilation Error !
                  // Ambiguous Access of 'func'
                  // could be the 'func' in base 'Box‘
                  // or could be the 'func' in base 'Car'
   }
};
```

User-Level Resolution of Ambiguities

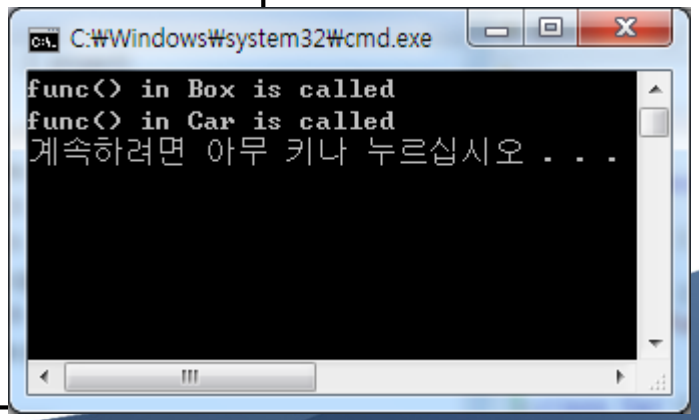
- We can resolve the ambiguity by explicitly specifying the class

```
class Box {
public :
   Box() {}
   void func() { cout << "func() in Box is called" << endl; }
};

class Car {
public :
   Car() {}
   void func() { cout << "func() in Car is called" << endl; }
};

class BoxCar : public Box, public Car {
public :
   BoxCar() {}
   void func_2() {
      Box::func();      // resolving ambiguity
      Car::func();
   }
};

void main() {
   BoxCar boxcar;
   boxcar.func_2();
}
```



Problems in Multiple Inheritance

- Although simple in concept, multiple inheritance can cause design-level and implementation level problems.
  - For example, the **Diamond Problem**
- Diamond situation should be avoided.
  - If a diamond situation is inevitable, the problematic class should be defined abstract.

```
class Animal {
public:
   int getWeight() { return weight; }

private :
   int weight;
};

class Tiger : public Animal {};
class Lion : public Animal {};
class Liger : public Tiger, public Lion {};

void main() {
   Liger lg ;
   int weight = lg.getWeight();  // Compilation Error !
                                 // ambiguous access of getWeight()
}
```