Module 6:
Polymorphism

Prof. Tran Minh Triet

Acknowledgement

- Slides
  - Course CS202: Programming Systems

    Instructor: MSc. Karla Fant,

    Portland State University
  - Course CS202: Programming Systems

    Instructor: Dr. Dinh Ba Tien,

    University of Science, VNU-HCMC
  - Course DEV275: Essentials of Visual Modeling with UML 2.0

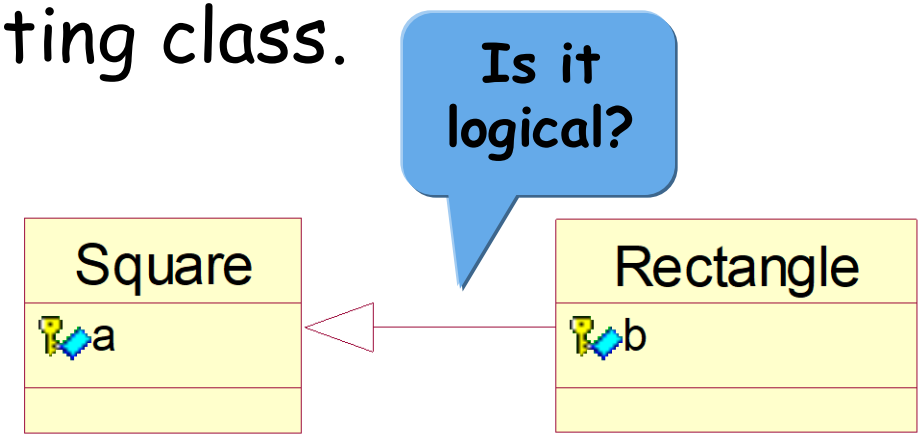
    IBM Software Group

Outline

- IS-A relationship
- Initialization of a pointer to base class
- Static binding
- Typecasting of a pointer to an object
- Virtual functions and polymorphism
- Pure virtual functions
- Abstract class

IS-A relationship

- Do not implement inheritance when you only want to use some attributes or behaviors of an existing class.



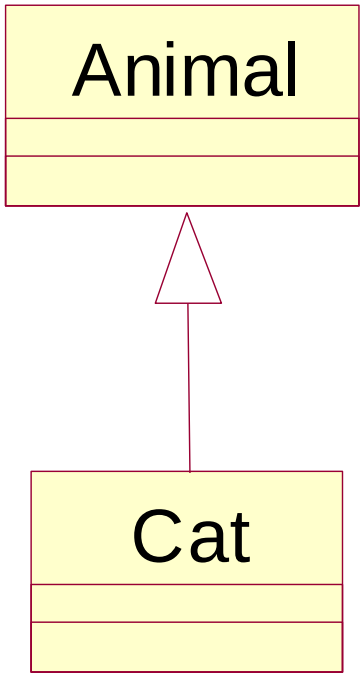
- The inheritance is only applied when there is a "**IS-A**" relationship between classes
  - E.g. Dog is an animal. Or, employer is an employee.

An example

- Cat "is an" animal.

```
class Animal
{
   …
};

class Cat: public Animal
{
   …
};
```



A pointer to base class

- A pointer to base class can be assigned with the address of an object of the derived class.
- For example:

```
Animal      *pAni;
Cat         c;
pAni = &c; //OK
```

Implicit type conversion in inheritance

- It is normal to pass a derived class variable to a function with an argument of base class data type.
- The compiler will do an implicit conversion.

```
Animal::Process(const Animal &a);
int main()
{
   Cat c;
   Animal::Process(c); //OK
}
```

Initialization of a pointer to base class

```
A*  pA;
```

pA can be instantiated:

```
pA = new X(…);
```

where X is A or X is a class derived from A

Initialization of a pointer to base class

```
A*    pA;
```

```
pA = new A(...);
```

```
pA = new B(...);
```

```
pA = new C(...);
```

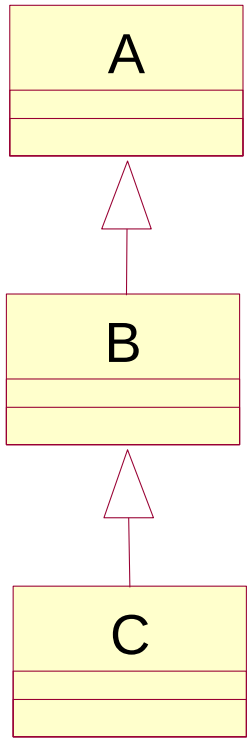
Initialization of a pointer to base class

```
B*    pB;
```

```
pB = new A(...);
```

```
pB = new B(...);
```

```
pB = new C(...);
```



Initialization of a pointer to base class

You are requested to create a polygon

```
CPolygon*  pShape;
```

You can draw a triangle

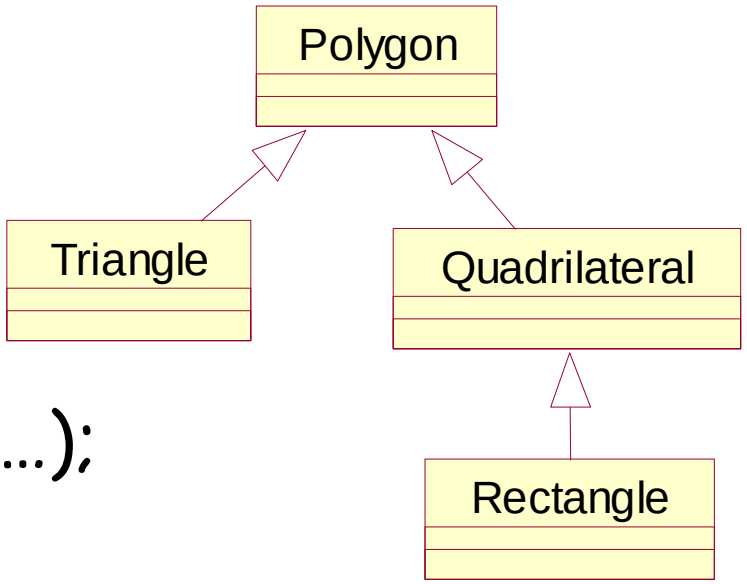
✓ `pShape = new Triangle(...);`

or a quadrilateral

✓ `pShape = new Quadrilateral(...);`

or a rectangle

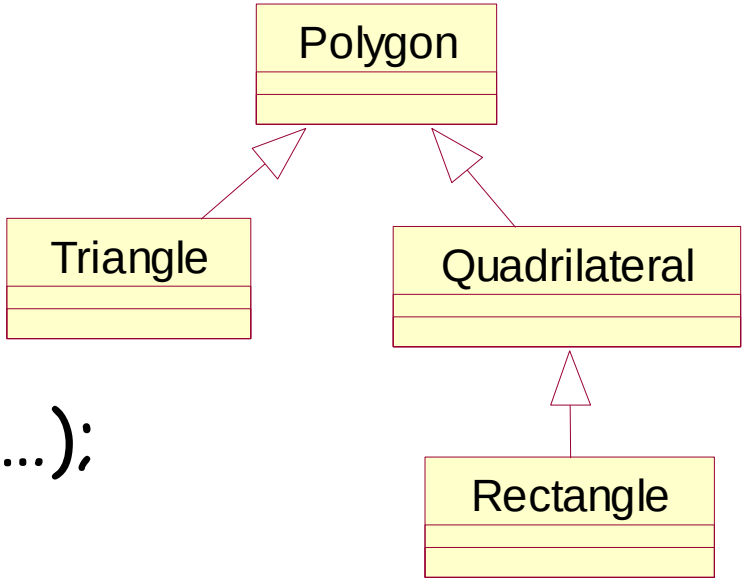
✓ `pShape = new Rectangle(...);`



Initialization of a pointer to base class

You are requested to create a quadrilateral

```
CQuadrilateral* pShape;
```



You can draw a rectangle

✓ `pShape = new Rectangle(...);`

or a quadrilateral

✓ `pShape = new Quadrilateral(...);`

You cannot draw a triangle

~~`pShape = new Triangle(...);`~~

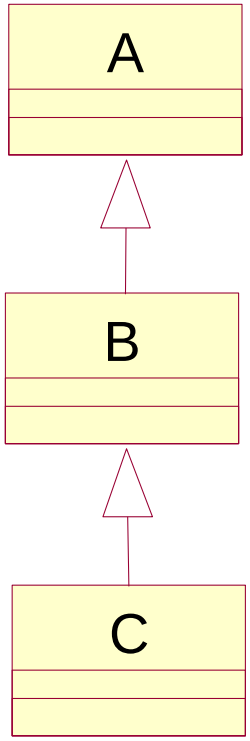
You cannot draw a general polygon (e.g. pentagon)

~~`pShape = new Polygon(...);`~~

Static binding

Consider the following situation:

- **class** *A* has **void Print()**
- **class** *B* also has **void Print()**
- **class** *C* has **void Print()** too




```
int main() {
   C     varC;
   B     varB;
   varB.Print();          // Print() of B
   varC.Print();          // Print() of C
   varC.B::Print();       // Print() of B
}
```

Static binding

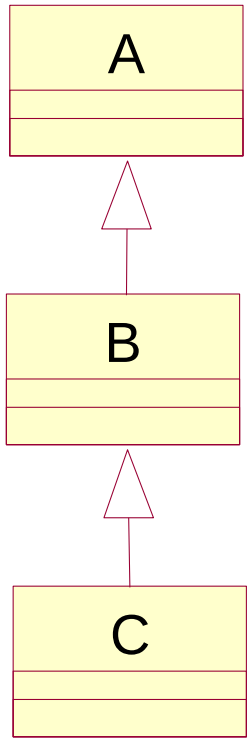
- Another example:

```
int main()
{
   A    varA;
   B    varB;
   C    varC;
   A    *var1, *var2;

   var1 = &varC;
   var2 = &varB;
   var1->Print();     // Print() of A
   var2->Print();     // Print() of A
}
```

var1 is formally a pointer to class A

var1 is actually pointed to an object of class C



Typecasting of pointer to an object

Formally, pA is a pointer to an object of class A

```
A*  pA;
```

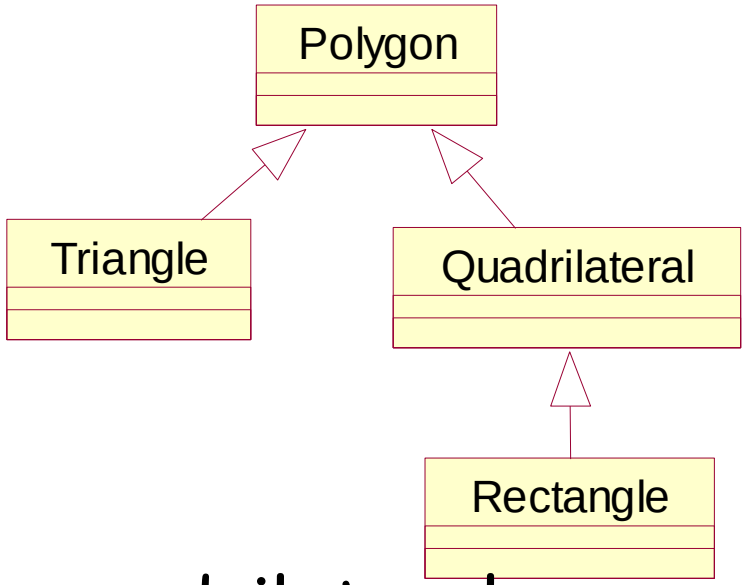
Actually, pA is currently pointer to an object of class X

```
pA = new X(…);
```

pA can be typecasted to a pointer to an object of class Y where Y is X or Y is a base class of X

```
Y* pY;
pY = (Y*)pA;
```

Typecasting of pointer to an object



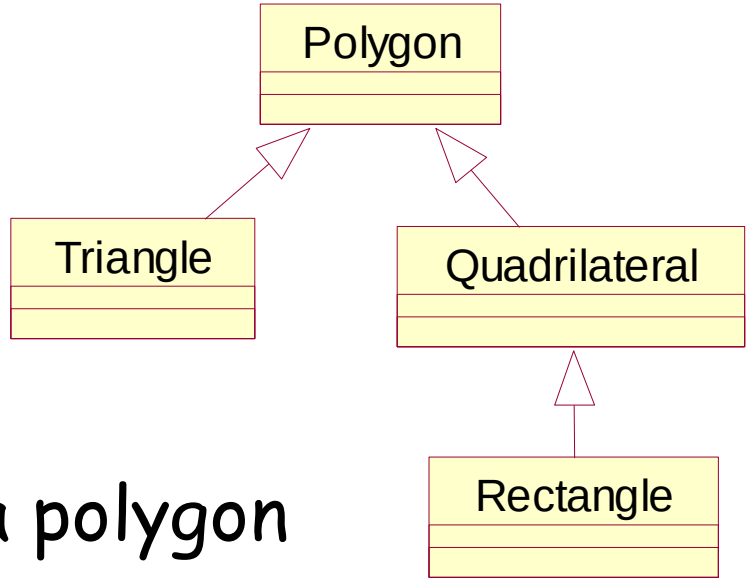
pShape is formally a pointer to a quadrilateral

```
CQuadrilateral* pShape;
```

Actually, pShape is a pointer to a rectangle

```
pShape = new Rectangle(…);
```

Typecasting of pointer to an object



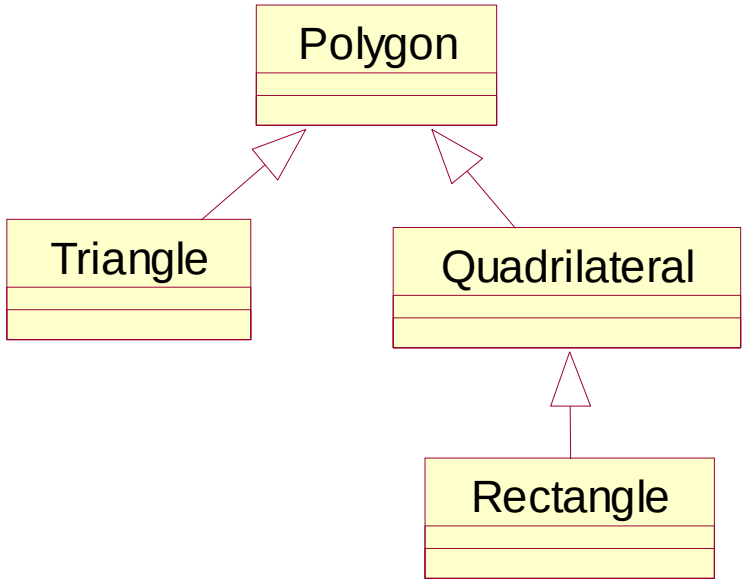
pShape is currently pointed to a polygon

✓ Polygon* p = (Polygon*)pShape;

pShape is currently pointed to a rectangle

✓ Rectangle* p = (Rectangle *)pShape;

Typecasting of pointer to an object



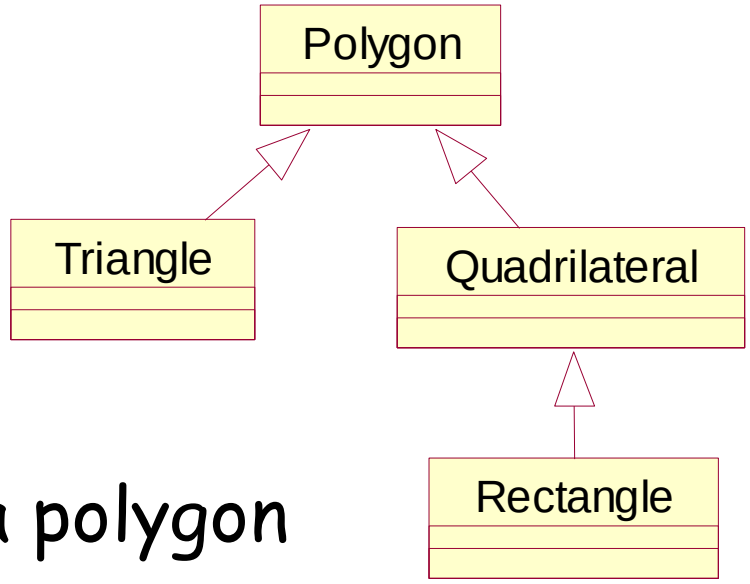
We no longer need this object

```
delete pShape;
```

Now, pShape is re-instantiated as a pointer to a quadrilateral

```
pShape = new Quadrilateral(…);
```

Typecasting of pointer to an object



pShape is currently pointed to a polygon

✓ Polygon* p = (Polygon*)pShape;

We are not sure if pShape is currently pointed to a rectangle

~~Rectangle* p = (Rectangle *)pShape;~~

Method invocation

pA -> F(…);

F(…) must be a member function of the *current* formal type of pA

The actual behavior of F(…) should correspond to the *actual* type of pA

Method Invocation

```
class  Polygon
{
public:
   double      Surface();
   double      Draw();
}

class  Triangle:
      public Polygon
{
public:
   double      Surface();
   double      Draw();
}
```

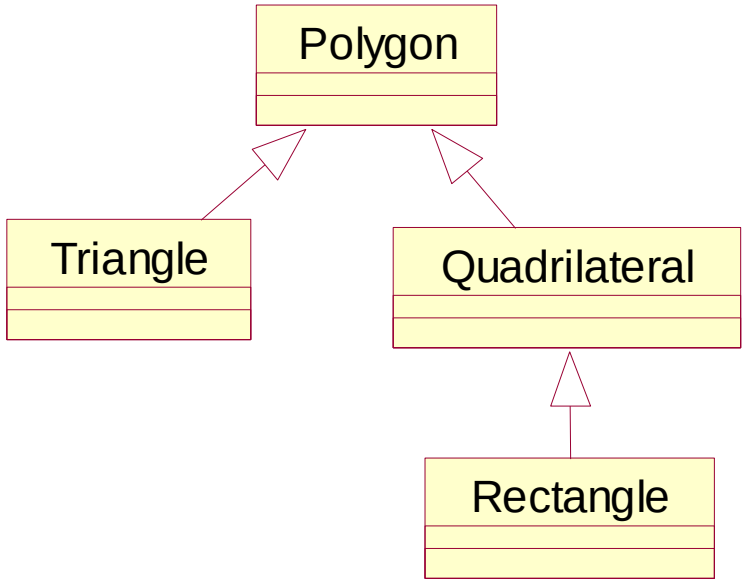
```
class  Quadrilateral :
      public Polygon
{
public:
   double      Surface();
   double      Draw();
}
class  Rectangle:
      public Quadrilateral
{
public:
   double      Surface();
   double      Draw();
}
```

Method Invocation

```
Polygon         myPolygon;
myPolygon.Draw();
```

```
Triangle        myTriangle;
myTriangle.Draw();
```

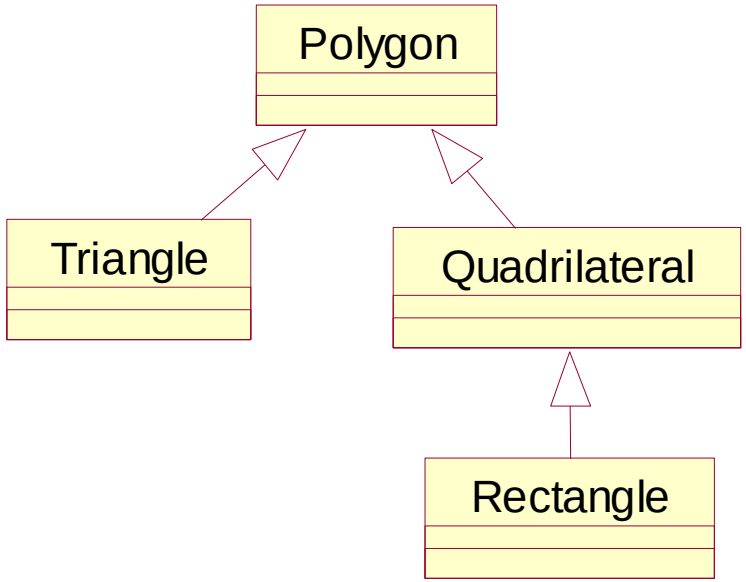
```
Polygon* pPolygon;
pPolygon = new Triangle(...);
pPolygon->Draw();
```



Method Invocation

```
Quadrilateral*      pPolygon;
pPolygon = new Rectangle(…);
pPolygon->Draw();

…
// IN CLASS DEMO
```



Virtual function

- Static binding problem:

```
class Animal
{
public:
        void talk()  { cout << "Don't talk"; }
};

class Cat: public Animal
{
public:
        void talk()  { cout << "Meo meo"; }
};

class Dog: public Animal
{
public:
        void talk()  { cout << "Gau gau"; }
};
```

```
void doSomething(Animal
p)
{
        p.talk();
}
void main()
{
        Cat       c;
        Dog       d;
        doSomething(c);
        doSomething(d);

        Animal  *p;
        p = &c;
        p->talk();
        p = &d;
        p->talk();
}
```

Bind to Animal implementation when compile

Bind to Animal implementation when compile

Virtual function

- Virtual function concept
  - Normal function
    - Function call binds to implementation at compile-time.
      - Static binding.
  - Virtual function
    - Function call binds to implementation at run-time .
      - Dynamic binding.
      - Implementation depends on run-time object.
  - C++ usage
    - Declaration: virtual <Function signature>;
    - Called through object pointer.

Virtual Method Invocation

```
class  Polygon
{
public:
virtual double Surface();
virtual double Draw();
}

class  Triangle:
      public Polygon
{
public:
virtual double Surface();
virtual double Draw();
}
```

```
class  Quadrilateral :
      public Polygon
{
public:
virtual double Surface();
virtual double Draw();
}
class  Rectangle:
      public Quadrilateral
{
public:
virtual double Surface();
virtual double Draw();
}
```

Virtual function

Dynamic binding

```
class Animal
{
public:
        virtual void talk()  { cout << "Don't talk"; }
};

class Cat: public Animal
{
public:
        void talk()  { cout << "Meo meo"; }
};

class Dog: public Animal
{
public:
        void talk()  { cout << "Gau gau"; }
};
```

```
void doSomething(Animal  *p)
{
        p->talk();
}
void main()
{
        Cat      c;
        Dog      d;
        doSomething(&c);
        doSomething(&d);

        Animal  *p;
        p = &c;
        p->talk();
        p = &d;
        p->talk();
}
```

implementation depends on run-time object

implementation depends on run-time object

Virtual function

- Pure virtual function
  - Has declaration only, no implementation
  - virtual <Function signature> = 0
  - Used for dynamic binding
  - Derived class provides implementation

```
class Animal
{
public:
        virtual void talk() = 0;
};
```

Pure virtual function, has no implementation!!

Virtual function/operation

- We sometimes declare a function, but we do not implement it

```
class Shape {
protected:
    int m_Width;
        int m_Height;
public:
    virtual int getArea() = 0;
};
```

For a general shape, we do not know how to calculate area.

A pure virtual function has "= 0"

Virtual function

Example:

```
class Animal
{
public:
        virtual void talk() = 0;
};

class Cat: public Animal
{
public:
        void talk()  {  cout << "Meo meo";  }
};

class Dog: public Animal
{
public:
        void talk()  {  cout << "Gau gau";  }
};
```

Abstract class

```
void doSomething(Animal   *p)
{
        p->talk();
}
void main()
{
        Cat       c;
        Dog       d;
        doSomething(&c);
        doSomething(&d);

        Animal  *p;
        p = new Animal; // Wrong
        p = new Cat;    // Right
        p->talk();
}
```

implementation depends on run-time object

Abstract class

- An abstract class is a class having at least one pure virtual function
  - Pure virtual operation does not have implementation

- An abstract class is called interface (in C++)
- We cannot instantiate an object from an abstract class
- A concrete class is a class that can be instantiated
- A derived concrete class must implement virtual functions from the base class

Abstract class example

```
class Shape {
protected:
        int m_Width;
        int m_Height;
public:
        virtual int getArea() = 0;
};

class Rectangle : public Shape {
public: int getArea() {
        return (m_Width * m_Height);   }
}
};
class Triangle: public Shape {
public: int getArea() {
        return (m_Width * m_Height)/2;
}
};
```

```
main() {
Rectangle rect;
Triangle tri;
rect.setWidth(5);
rect.setHeight(7);

tri.setWidth(5);
tri.setHeight(7);

Shape shape; // wrong!
```

Why do we need abstract classes?

- An abstract class provides a base class for inheritance
- Detailed implementation of one or many operations is yet to know
- Support polymorphism

Virtual destructor: problem

```
class Employee
{
private:
        char        *m_Name;
public:
        ~Employee()  {
                    delete m_Name;
        }
};
class Doctor : public Employee
{
private:
        char        *m_Specialty;
public:
        ~Doctor()  {
                    delete m_Specialty;
        }
};
```

```
void main()
{
        Doctor  *doc = new Doctor;
        delete  doc;

        Employee  *e = new Doctor;
        delete  e;
}
```

Call order:
~Doctor()
~Employee()

Call order:
~Employee()

Is there any problem with this?

Virtual destructor: solution

```
class Employee
{
private:
        char        *m_Name;
public:
        ~virtual Employee()  {
                    delete m_Name;
        }
};
class Doctor : public Employee
{
private:
        char        *m_Specialty;
public:
        ~Doctor()  {
                    delete m_Specialty;
        }
};
```

```
void main()
{
        Doctor  *doc = new Doctor;
        delete  doc;

        Employee  *e = new Doctor;
        delete  e;
}
```

Call order:
~Doctor()
~Employee()

Call order:
~Doctor()
~Employee()