

CS217 Object Oriented Programming

Polymorphism and Virtual functions

Overriding a functions in the subclass

When a subclass **declares a function of the name previously known in its superclass, the original method is overridden**. This means that the subclass hides the previous meaning of the function identifier and any invocation encoded within the **subclass** will refer to the **newer** method.

```

class A
{
public:
    int x;
    void print() { cout<<" Class A    x= "<<x<<endl;}
};

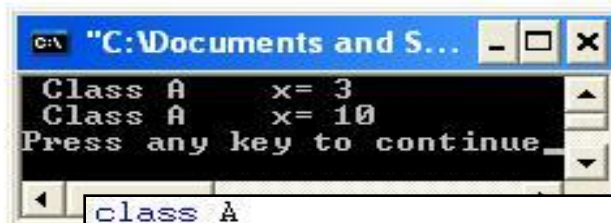
class B: public A
{
public:
    int a;
    void print() {  cout<<" Class B    a= "<<a; A::print();}
};

void printAll(A &obj)
{
    obj.print();
}

void main()
{
    A obj1;
    obj1.x=3;
    B obj2;
    obj2.x=10;
    obj2.a=100;

    printAll(obj1);
    printAll(obj2);
}

```



```

C:\Documents and S...
Class A    x= 3
Class A    x= 10
Press any key to continue

```

```

class A
{
public:
    int x;
    virtual void print() { cout<<" Class A    x= "<<x<<endl;}
};

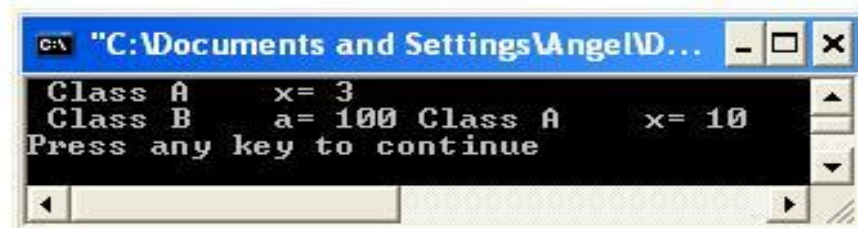
class B: public A
{
public:
    int a;
    void print() {  cout<<" Class B    a= "<<a; A::print();}
};

void printAll(A &obj)
{
    obj.print();
}

void main()
{
    A obj1;
    obj1.x=3;
    B obj2;
    obj2.x=10;
    obj2.a=100;

    printAll(obj1);
    printAll(obj2);
}

```



```

C:\Documents and Settings\Angel\W...
Class A    x= 3
Class B    a= 100 Class A    x= 10
Press any key to continue

```

Overriding a function in the subclass

```
#include <iostream>
using namespace std;
class Pet {
protected:
    string Name;
public:
    Pet(string n) { Name = n; }
    void MakeSound(void) { cout << Name << " the Pet says: Shh! Shh!" << endl; }
};
class Cat : public Pet {
public:
    Cat(string n) : Pet(n) {}
    void MakeSound(void) { cout << Name << " the Cat says: Meow! Meow!" << endl; }
};
class Dog : public Pet {
public:
    Dog(string n) : Pet(n) {}
    void MakeSound(void) { cout << Name << " the Dog says: Woof! Woof!" << endl; }
};
int main(void) {
    Pet *a_pet1, *a_pet2;
    Cat *a_cat;
    Dog *a_dog;

    a_pet1 = a_cat = new Cat("Kitty");
    a_pet2 = a_dog = new Dog("Doggie");
    a_pet1 -> MakeSound();
    a_cat -> MakeSound();
    a_pet2 -> MakeSound();
    a_dog -> MakeSound();
    return 0;
}
```

Overriding a method in the subclass

I want to ask you now for your full attention as we are going to introduce one of the most important objective notions:

Polymorphism.

This is a method to redefine the behavior of a superclass without touching its implementation.

Overriding a method in the subclass

- The word “**polymorphism**” means that the one and same class may show many (“poly”) forms (“*morphs*”) not defined by the class itself, but **by its subclasses**.
- Another definition says that polymorphism is **the ability to realize class behaviour in multiple ways**.
- Both definitions are ambiguous and could not actually be used to explain the true nature of the phenomenon. I believe that an example will work better.

- Take a look at the code below. It's almost the same as the previous one. The *main* function may look different but the class definitions are nearly identical. There's only one difference – one keyword has been added to the code.
- Can you find it?

```
#include <iostream>
using namespace std;
class Pet {
protected:
    string Name;
public:
    Pet(string n) { Name = n; }
    virtual void MakeSound(void) { cout << Name << " the Pet says: Shh! Shh!" << endl; }
};
class Cat : public Pet {
public:
    Cat(string n) : Pet(n) {}
    void MakeSound(void) { cout << Name << " the Cat says: Meow! Meow!" << endl; }
};
class Dog : public Pet {
public:
    Dog(string n) : Pet(n) {}
    void MakeSound(void) { cout << Name << " the Dog says: Woof! Woof!" << endl; }
};
```

```
int main(void) {
    Pet *a_pet1, *a_pet2;
    Cat *a_cat;
    Dog *a_dog;

    a_pet1 = a_cat = new Cat("Kitty");
    a_pet2 = a_dog = new Dog("Doggie");
    a_pet1 -> MakeSound();
    a_cat -> MakeSound();
    static_cast<Pet *>(a_cat) -> MakeSound();
    a_pet2 -> MakeSound();
    a_dog -> MakeSound();
    static_cast<Pet *>(a_dog) -> MakeSound();
    return 0;
}
```

Overriding a method in the subclass

Yes, you're right – it's the word “**virtual**”, placed in front of the *MakeSound* member function, inside the *Pet* class body.

This word means that the method won't be overridden within any of the possible subclasses. It also means that the method will be **redefined**(replaced) at the level of the original class.

To make a long story short – the example program will output a somewhat surprising piece of text. This is how it goes:

Kitty the Cat says: Meow! Meow!

Kitty the Cat says: Meow! Meow!

Kitty the Cat says: Meow! Meow!

Doggie the Dog says: Woof! Woof!

Doggie the Dog says: Woof! Woof!

Doggie the Dog says: Woof! Woof!

6.3.4 Overriding a method in the subclass (4)

The next example shows that the binding between the origin of the virtual function (inside the superclass) and its replacement (defined within the subclass) is created dynamically, during the execution of the program.

Take a look at the example →

```
#include <iostream>
using namespace std;
class Pet {
protected:
    string Name;
public:
    Pet(string n) { Name = n; MakeSound(); }
    virtual void MakeSound(void) { cout << Name << " the Pet says: Shh! Shh!" << endl; }
};
class Cat : public Pet {
public:
    Cat(string n) : Pet(n) {}
    void MakeSound(void) { cout << Name << " the Cat says: Meow! Meow!" << endl; }
};
class Dog : public Pet {
public:
    Dog(string n) : Pet(n) {}
    void MakeSound(void) { cout << Name << " the Dog says: Woof! Woof!" << endl; }
};
```

```
int main(void) {
    Cat *a_cat;
    Dog *a_dog;

    a_cat = new Cat("Kitty");
    a_dog = new Dog("Doggie");

    return 0;
}
```

6.3.4 Overriding a method in the subclass (4)

We invoke the *MakeSound* method as part of the *Pet* constructor. We already know that the method is **polymorphically** replaced by the new implementations presented by the *Cat* and *Dog* subclasses. We don't know yet **when** the replacement occurs.

The program will output the following lines:

Kitty the Pet says: Shh! Shh!

Doggie the Pet says: Shh! Shh!

This means that the binding between the original functions and their polymorphic implementations is established when the subclass object is created, not sooner.

6.3.5 Overriding a method in the subclass (5)

- *Overriding a method in the subclass – continued*
- The experiment we're going to perform using our software guinea pig refers to the fact that the virtual method may be invoked not only from outside the class but also from within.
- Check out this example, please →

```
#include <iostream>
using namespace std;
class Pet {
protected:
    string Name;
public:
    Pet(string n) { Name = n; }
    virtual void MakeSound(void) { cout << Name << " the Pet says: Shh! Shh!" << endl; }
    void WakeUp(void) { MakeSound(); }
};
class Cat : public Pet {
public:
    Cat(string n) : Pet(n) {}
    void MakeSound(void) { cout << Name << " the Cat says: Meow! Meow!" << endl; }
};
class Dog : public Pet {
public:
    Dog(string n) : Pet(n) {}
    void MakeSound(void) { cout << Name << " the Dog says: Woof! Woof!" << endl; }
};
```

```
int main(void) {
    Cat *a_cat;
    Dog *a_dog;

    a_cat = new Cat("Kitty");
    a_cat -> WakeUp();
    a_dog = new Dog("Doggie");
    a_dog -> WakeUp();

    return 0;
}
```

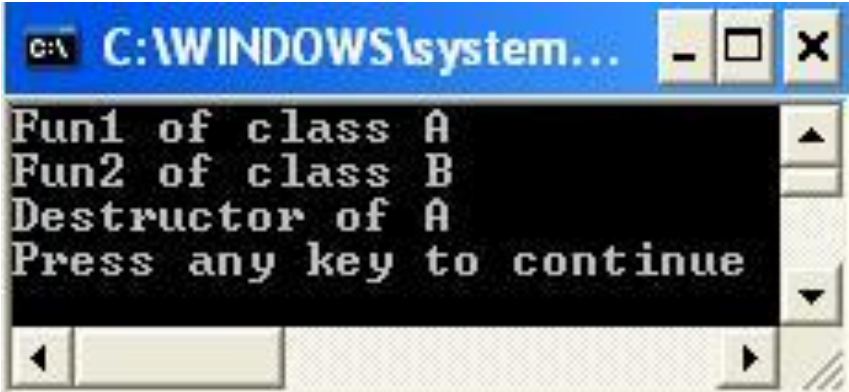
Classes and Virtual Destructors

- Classes with pointer member variables should have the destructor
 - Destructor can be designed to deallocate storage for dynamic objects
- If a derived class object is passed to a formal parameter of the base class type, destructor of the base class executes
 - Regardless of whether object is passed by reference or by value
- Solution: use a virtual destructor (base class)

```
class A
{
public:
    void fun1(){cout<<"Fun1 of class A\n";}
    virtual void fun2(){cout<<"Fun2 of class A\n";}
    ~A(){cout<<"Destructor of A\n";}
};

class B:public A
{
public:
    virtual void fun1(){cout<<"Fun1 of class B\n";}
    void fun2(){cout<<"Fun2 of class B\n";}
    ~B(){cout<<"Destructor of B\n";}
};

void main()
{
    A *p;
    p=new B;
    p->fun1();
    p->fun2();
    delete p;
}
```



The screenshot shows a Windows command prompt window with the title bar "C:\WINDOWS\system...". The window contains the following text output from the program:

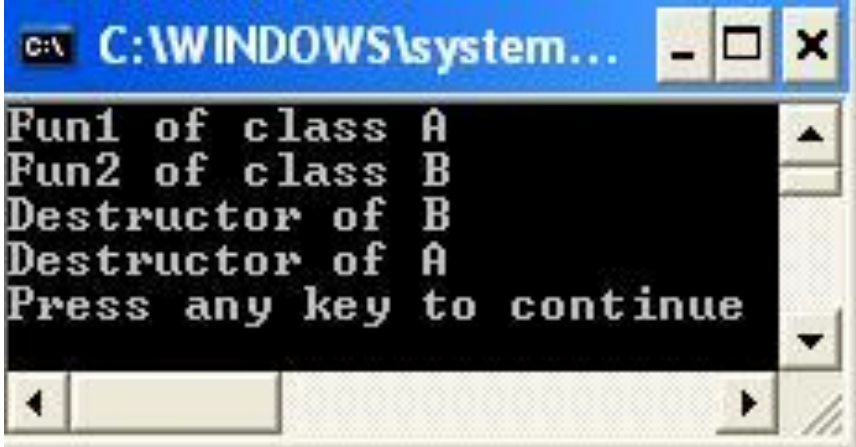
```
Fun1 of class A
Fun2 of class B
Destructor of A
Press any key to continue
```

The text is displayed in a monospaced font on a black background. The window has standard Windows controls (minimize, maximize, close) in the title bar and a scroll bar on the right side.

```
class A
{
public:
    void fun1(){cout<<"Fun1 of class A\n";}
    virtual void fun2(){cout<<"Fun2 of class A\n";}
    virtual ~A(){cout<<"Destructor of A\n";}
};

class B:public A
{
public:
    virtual void fun1(){cout<<"Fun1 of class B\n";}
    void fun2(){cout<<"Fun2 of class B\n";}
    ~B(){cout<<"Destructor of B\n";}
};

void main()
{
    A *p;
    p=new B;
    p->fun1();
    p->fun2();
    delete p;
}
```



A screenshot of a Windows command prompt window. The title bar shows the path 'C:\WINDOWS\system...' and standard window controls. The black command prompt area displays the following output in white text: 'Fun1 of class A', 'Fun2 of class B', 'Destructor of B', 'Destructor of A', and 'Press any key to continue'. The window has a scrollbar on the right side.

Virtual functions for class A:

fun2

Virtual functions for class B:

fun1 and fun2

Virtual functions for class C:

fun1 and fun2 and fun3

```
class A
{
public:
    void fun1(){cout<<"Fun1 of class A\n";}
    virtual void fun2(){cout<<"Fun2 of class A\n";}
    void fun3(){cout<<"Fun3 of class A\n";}
    ~A(){cout<<"Destructor of A\n";}
};

class B:public A
{
public:
    virtual void fun1(){cout<<"Fun1 of class B\n";}
    void fun2(){cout<<"Fun2 of class B\n";}
    void fun3(){cout<<"Fun3 of class B\n";}
    ~B(){cout<<"Destructor of B\n";}
};

class C:public B
{
public:
    void fun1(){cout<<"Fun1 of class C\n";}
    void fun2(){cout<<"Fun2 of class C\n";}
    virtual void fun3(){cout<<"Fun3 of class C\n";}
    ~C(){cout<<"Destructor of C\n";}
};

void OutFun(B & objParameter)
{
    objParameter.fun1();
    objParameter.fun2();
    objParameter.fun3();
}

void main()
{
    C objC;
    OutFun(objC);
    B *p=new C;
    delete p;
}
```

```
C:\WINDOWS\system...
Fun1 of class C
Fun2 of class C
Fun3 of class B
Destructor of B
Destructor of A
Destructor of C
Destructor of B
Destructor of A
Press any key to continue
```

Classes and Virtual Destructors (continued)

- The **virtual destructor** of a base class automatically makes the destructor of a derived class virtual
 - After executing the destructor of the derived class, the destructor of the base class executes
- If a base class contains virtual functions, make the destructor of the base class virtual


```

class A
{
public:
    void fun1(){cout<<"Fun1 of class A\n";}
    virtual void fun2(){cout<<"Fun2 of class A\n";}
    void fun3(){cout<<"Fun3 of class A\n";}
    virtual ~A(){cout<<"Destructor of A\n";}
};

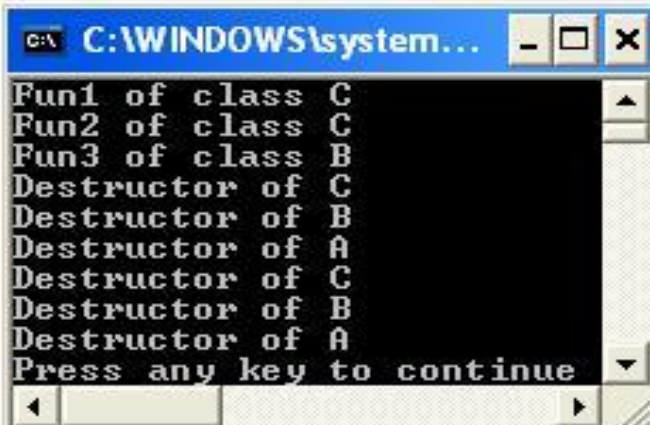
class B:public A
{
public:
    virtual void fun1(){cout<<"Fun1 of class B\n";}
    void fun2(){cout<<"Fun2 of class B\n";}
    void fun3(){cout<<"Fun3 of class B\n";}
    ~B(){cout<<"Destructor of B\n";}
};

class C:public B
{
public:
    void fun1(){cout<<"Fun1 of class C\n";}
    void fun2(){cout<<"Fun2 of class C\n";}
    virtual void fun3(){cout<<"Fun3 of class C\n";}
    ~C(){cout<<"Destructor of C\n";}
};

void OutFun(B & objParameter)
{
    objParameter.fun1();
    objParameter.fun2();
    objParameter.fun3();
}

void main()
{
    C objC;
    OutFun(objC);
    B *p=new C;
    delete p;
}

```



```

C:\WINDOWS\system...
Fun1 of class C
Fun2 of class C
Fun3 of class B
Destructor of C
Destructor of B
Destructor of A
Destructor of C
Destructor of B
Destructor of A
Press any key to continue

```

Abstract Classes and Pure Virtual Functions

- Through inheritance, we can derive new classes without designing them from scratch
 - Derived classes inherit existing members of base class, can add their own members, and also redefine or override public and protected member functions
 - Base class can contain functions that you would want each derived class to implement
 - Base class may contain functions that may not have meaningful definitions in the base class

Abstract Classes and Pure Virtual Functions (continued)

```
class shape
{
public:
    virtual void draw();
        //Function to draw the shape.

    virtual void move(double x, double y);
        //Function to move the shape at the position
        //(x, y).

    .
    .
    .
};
```

```
virtual void draw() = 0;
virtual void move(double x, double y) = 0;
```

- To make them **pure virtual functions**:

```

class A
{
public:
    virtual void print()=0;
};

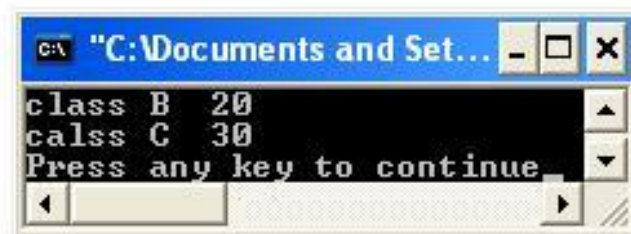
class B:public A
{
    int y;
public:
    B(int a){y=a;}
    void print() {cout<<"class B  "<<y<<endl;}
};

class C:public A
{
    int z;
public:
    C(int a){z=a;}
    void print() {cout<<"calss C  "<<z<<endl;}
};

void fun(A & x)
{
    x.print();
}

void main()
{
    B obj2(20);
    C obj3(30);
    fun(obj2);
    fun(obj3);
}

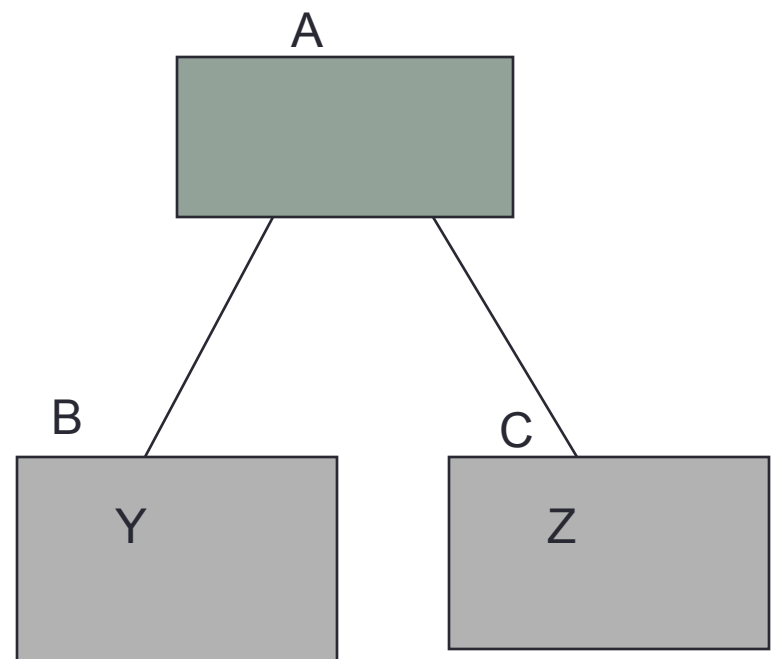
```



```

C:\Documents and Set...
class B  20
calss C  30
Press any key to continue

```



Virtual functions

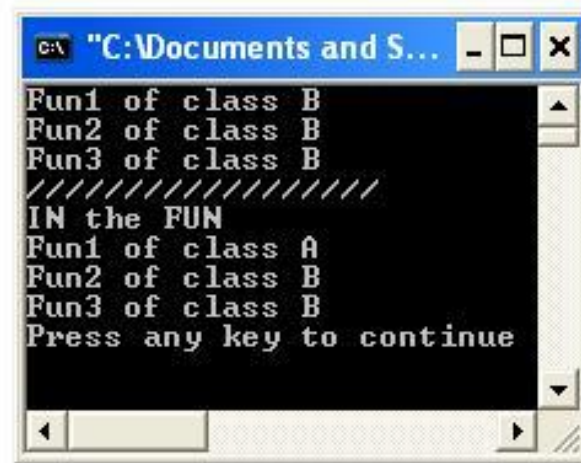
```
class A
{
public:
    void fun1() {cout<<"Fun1 of class A\n";}
    virtual void fun2(){cout<<"Fun2 of class A\n";}
    virtual void fun3()=0;
};
```

Class A is Abstract class

```
class B:public A
{
public:
    void fun1() {cout<<"Fun1 of class B\n";}
    void fun2() {cout<<"Fun2 of class B\n";}
    void fun3() {cout<<"Fun3 of class B\n";}
};
```

Class B is non abstract class

```
//void fun(A obj)  illigal
void fun(A & obj)
{
    cout<< "IN the FUN \n";
    obj.fun1();
    obj.fun2();
    obj.fun3();
}
void main()
{
    // A obj1;          illigal
    B obj2;
    obj2.fun1();
    obj2.fun2();
    obj2.fun3();
    cout<<"////////////////\n";
    fun(obj2);
}
```



```
C:\Documents and S...
Fun1 of class B
Fun2 of class B
Fun3 of class B
////////////////
IN the FUN
Fun1 of class A
Fun2 of class B
Fun3 of class B
Press any key to continue
```

Abstract Classes and Pure Virtual Functions (continued)

- Abstract class: contains one or more pure virtual functions

```
class shape ← You cannot create objects of an abstract class
{
public:
    virtual void draw() = 0;
        //Function to draw the shape. Note that this is a
        //pure virtual function.
    virtual void move(double x, double y) = 0;
        //Function to move the shape at the position
        //(x, y). Note that this is a pure virtual
        //function.
    .
    .
    .
};
```

Abstract Classes and Pure Virtual Functions (continued)

- If we derive `rectangle` from `shape`, and want to make it a nonabstract class:
 - We must provide the definitions of the pure virtual functions of its base class
- Note that an abstract class can contain instance variables, constructors, and functions that are not pure virtual
 - The class must provide the definitions of constructor/functions that are not pure virtual


```

class A //Abstract
{
public:
    void fun1(){cout<<"Fun1 of class A\n";}
    virtual void fun2(){cout<<"Fun2 of class A\n";}
    virtual void fun3()=0;
};

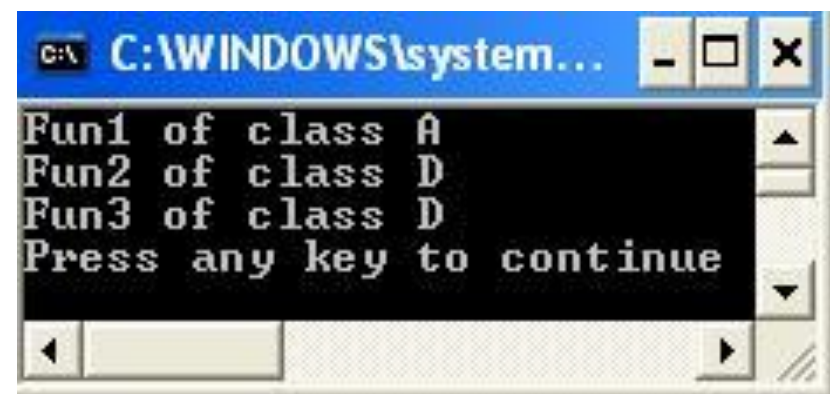
class B:public A //Abstract...why????
{
public:
    void fun1(){cout<<"Fun1 of class B\n";}
    void fun2(){cout<<"Fun2 of class B\n";}
};

class D:public B //Non Abstract
{
public:
    void fun1(){cout<<"Fun1 of class D\n";}
    void fun2(){cout<<"Fun2 of class D\n";}
    void fun3(){cout<<"Fun3 of class D\n";}
};

//void OutFun(A obj) illegal
// A OutFun () illegal
void OutFun(A & objParameter)
{
    objParameter.fun1();
    objParameter.fun2();
    objParameter.fun3();
}

void main()
{
    //A obj B obj illegal
    D obj;
    OutFun(obj);
}

```



```

C:\WINDOWS\system...
Fun1 of class A
Fun2 of class D
Fun3 of class D
Press any key to continue

```