Polymorphism (Lecture -2)

(CS 217)

Dr. Muhammad Aleem,

Department of Computer Science,
National University of Computer & Emerging Sciences,
Islamabad Campus

Lecture 2 - Contents

- Re-cap of Pointers to Derived Classes
- Virtual Function
- Redefine Vs. Override
- Virtual Destructors

Summary – Based and Derived Class Pointers

1. **Base-class pointer pointing to base-class object**
   - **Straightforward, Alright**

2. **Derived-class pointer pointing to derived-class object**
   - **Straightforward, Alright**

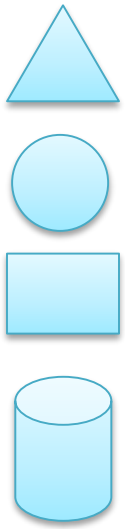
3. **Base-class pointer pointing to derived-class object**
   - **Safe**
   - Can **access non-virtual methods** of **only base-class**
   - Can **access virtual methods** of **derived class**

4. **Derived-class pointer pointing to base-class object**
   - **Compilation error**

Example: Pointers to Derived Classes

- To **access several derived class objects**, **we have created a pointer array of type base class**, and then used this **pointer array** to point to **different derived class objects.**

```
Line *p[4];
p[0] = new Triangle (3, 4, 5, 19 );
p[1] = new Circle (3, 4, 5 );
p[2] = new Rectangle ( 3, 4, 10 , 20 );
p[3] = new Cylinder ( 3, 4, 5, 10 );
for ( int loop = 0; loop < 4; loop ++ )
{
    p[loop]->draw();
    cout << "The area is " << p[loop]->GetArea( );
}
```



Example: Shapes Implementation using Polymorphism

```
class Shape{
public:
   virtual void sayHi() { cout <<‘‘Just hi! \n’’;}
};
class Triangle : public Shape{
public:
   virtual void sayHi() { cout <<‘‘Hi from a triangle! \n’’;}
};
class Rectangle : public Shape{
public:
   virtual void sayHi() { cout <<‘‘Hi from a rectangle! \n; }
};

int main(){
   Shape *p;
   int which;
   cout << ‘‘1 -- shape, 2 -- triangle, 3 -- rectangle\n ’’;
   cin >> which;
   switch ( which ) {
   case 1:  p = new Shape; break;
   case 2:  p = new Triangle; break;
   case 3:  p = new Rectangle; break;
   }
   p -> sayHi();     // dynamic binding of sayHi()
   delete p;
}
```

DEMO: vShapes.cpp

Virtual function

- Declaring a **function virtual** will **make** the **compiler check the type of each object** to search the **specific version** of the **virtual function**

- **To declare a function virtual, we use the keyword virtual:**

```
class Shape {
  public:
    virtual void sayHi () //A virtual function
    {
          cout <<“This is a virtual function!\n”;
    }
};
```

Example: Classes Using Virtual Functions

Class Hierarchy with Virtual Functions:

```
class Animal{
  public:
     virtual void id(){cout << "animal";}
};

class Cat : public Animal{
  public:
     virtual void id() override {cout << "cat";}
};

class Dog : public Animal{
  public:
     virtual void id() override {cout << "dog";}
};
```

Virtual Functions

- If the **member function definition** is **outside the class**, the **keyword virtual must not be specified again**.

```
class Shape{
public:
  virtual void sayHi ();
};
virtual void Shape::sayHi (){  // error
  cout << ‘‘Just hi! \n’’;
}
```

- **Virtual function can not be a stand-alone or static function.**

- **A virtual function can be inherited from a base class by a derived class.**

Virtual Functions

- The ***virtualness*** of an **operation** is **always inherited**

- If a **function** is **virtual** in the **base class**, it **must be virtual** in the **derived class**

- **Even if** the **keyword** "**virtual**" **is not specified**

(**Best Practice**: always use **virtual** keyword in **derived classes for clarity**.)

- If **no overridden function** is **provided**, then the **virtual function** of **base class** **will be used.**

Redefine Vs. Override

- **redefine (a method)** is **used for static binding** (i.e., ***non-virtual functions***):
  - **Example: both base and derived classes contains a non-virtual function with same signature:**

```
float getArea( ) { … } //A non-virtual function
```

- **override (a method)** is **used for dynamic binding** (i.e., ***virtual functions***):
  - **Example: both base and derived classes contains a virtual function with the same signature:**

```
virtual float getArea( ) { … } //A virtual function
```

Virtual Functions and override Keyword

- The **override keyword** serves **two purposes**:

  1. **For Programmer**: **to indicate** a **virtual function** that is **overridden**

  2. **For Compiler**: to **make sure same signature function exists** in **Base Class** too (***otherwise a Compilation Error)***.

Polymorphism Example
(using Base Class's Pointers, References, Override keywords)

```
class Shape{
public:
   virtual void sayHi() { cout <<‘‘Just hi! \n’’;}
};
class Triangle : public Shape{
public:
   // overrides Shape::sayHi(), automatically virtual
   void sayHi() { cout <<‘‘Hi from a triangle! \n’’;}
};

void print(Shape obj, Shape *ptr, Shape &ref){
   ptr -> sayHi();    // bound at run time
   ref.sayHi();       // bound at run time
   obj.sayHi();       // bound at compile time
}

int main(){
  Triangle mytri;
  print( mytri, &mytri, mytri );
}
```

DEMO: PolyExample2.cpp

Virtual Constructor/Destructors

- **Constructor cannot be virtual**, because when a **constructor is being executed** there is **no virtual table (for that object)** in the **memory**, (i.e., **no virtual pointer defined yet)**.

- **Destructors can be virtual (should be made virtual)**

- **A virtual Destructor, ensures that** the **derived class destructor** is **called** when a **base class pointer** is **used. (deleting memory allocation in derived class first …)**

```
virtual ~Shape();
```

Example: Non-Virtual Destructors

```
class base {
public:
    ~base() {
   cout<<“destructing base\n”;
    }
};

class derived : public base {
public:
    ~derived() {
cout<<“destructing derived\n”;
    }
};
```

```
int main()
{
    base *p = new derived;
    delete p;
     return 0;
}
```

Output:

```
destructing base
```

Using non-virtual destructor

Example: Virtual Destructors

```
class base {

public:
   virtual ~base() {
    cout<<“destructing base\n”;
   }
};

class derived : public base {
public:
   ~derived() {
    cout<<“destructing derived\n”;
   }
};
```

```
int main()
{
     base *p = new derived;
     delete p;
      return 0;
}
```

Output:

```
destructing derived
destructing base
```

Using virtual destructor

End of Lecture 2