

CSE 1203

Object Oriented Programming [C++]

Chapter 3: Polymorphism

Learning Objectives

2

To know about:

- Function Overloading
- Operator Overloading
- Function Overriding
- Polymorphism

Polymorphism

3

Polymorphism in C++

Polymorphism-

The word polymorphism means having many forms.

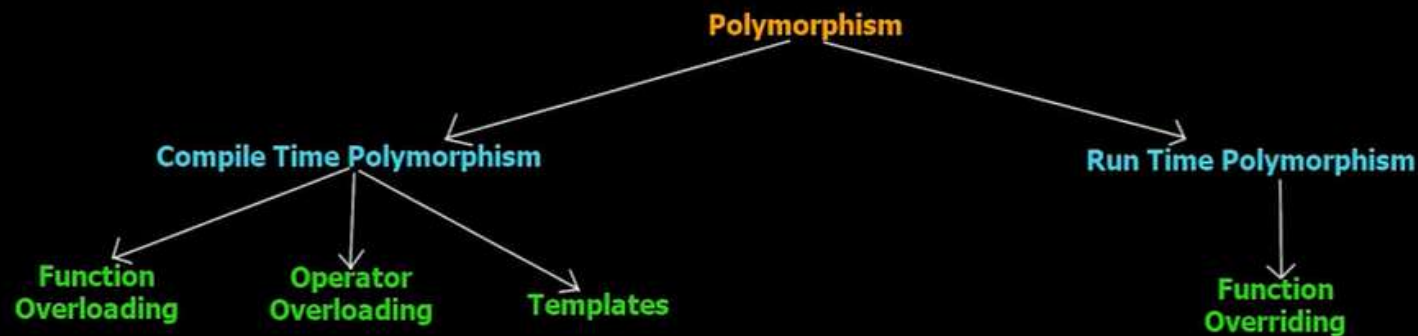
In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.

Polymorphism is an important and basic concept of OOPS.

In C++, An operator or function can be given different meanings or functions.

In C++ polymorphism is mainly divided into two types:

- 1) Compile time Polymorphism (early binding / static polymorphism)
- 2) Runtime Polymorphism (late binding / dynamic polymorphism)



Function Overloading

4

- Function overloading means to have *more than one function* with the *same name* but with *different parameters*.
- Overloaded functions are differentiated by checking
 1. *Number* of arguments.
 2. *Type & sequence* of arguments but *not by return type* of the function.

Function Overloading

- An Overloaded function must have:
 - Different **type** of **parameters**
 - Different **number** of **parameters**
 - Different **sequence** of **parameters**

1. void **print**();
2. void **print**(**int** a);
3. void **print**(**float** a);
4. void **print**(**int** a, **int** b);
5. void **print**(**int** a, **double** b);
6. void **print**(**double** a, **int** b);

```
#include <iostream>
using namespace std;
```

```
class A{
    public:
    int Sum(int a,int b){
        return (a+b);
    }
    double Sum(double a,double b){
        return (a+b);
    }
};
```

```
int main(void) {
    A a;
    cout<<a.Sum(3,4);
    cout<<endl;
    cout<<a.Sum(2.5,4.6);
}
```

Operator Overloading

- C++ **allows** you to specify **more than one definition** for an **operator** in the same scope, which is called **operator overloading**.
- You can **redefine or overload** most of the built-in operators available in C++
- It is a type of **polymorphism** in which an **operator** is overloaded to give user defined meaning to it.
- Almost any operator can be overloaded in C++. However there are few operator which **can not be overloaded**. **Operator that are not overloaded** are follows-
 - scope operator (::)
 - **sizeof**
 - member selector -(.)
 - member pointer selector - (*)
 - ternary operator - (?:)



Binary Operator Overloading

```
#include<iostream>
using namespace std;

class Complex {
private:
    int real, imag;
public:
    Complex(int r = 0, int i = 0){
        real = r;
        imag = i;
    }

    // This is automatically called when '+'
    // is used with between two Complex objects
    Complex operator + (Complex const &obj) {
        Complex res;
        res.real = real + obj.real;
        res.imag = imag + obj.imag;
        return res;
    }

    void print() {
        cout << real << " + i" << imag << "\n";
    }
};
```

```
int main()
{
    Complex c1(10, 5), c2(2, 4);
    Complex c3;
    c3 = c1 + c2; //c3=c1.add(c2)
    c3.print();
}
```

Operator functions are the same as normal functions. The only differences are, that the name of an operator function is always the **operator** keyword followed by the symbol of the operator and operator functions are called when the corresponding operator is used.

Unary Operator Overloading

```
#include <iostream>
using namespace std;

class Counter{
private:
    int count;
public:
    Counter(){count=0; }
    int get_count()
        {return count;}
    void operator++()
        {count++;}
};

int main(void)
{
    Counter c1, c2;
    c1++;
    cout<<"c1="<<c1.get_count();
}
```

vul ase ++c1 hobe

The operator function uses unary operator. Here ++ operator is used to increment the value of private member data count.

Function Overriding

- If we inherit a class into the **derived class** and provide a definition for one of the base class's function again inside the **derived class**, then that function is said to be **overridden**, and this mechanism is called **Function Overriding**
- Inheritance should be there. Function overriding cannot be done within a class. For this we require a derived class and a **base class**
- Function that is redefined must have exactly the same declaration in both **base** and **derived class**, that means same name, same return type and same parameter list
- If you create an object of the derived class and call the member function which exists in both the classes then member function in the **derived class** is invoked and the function in the **base class** is ignored.

Function Overriding

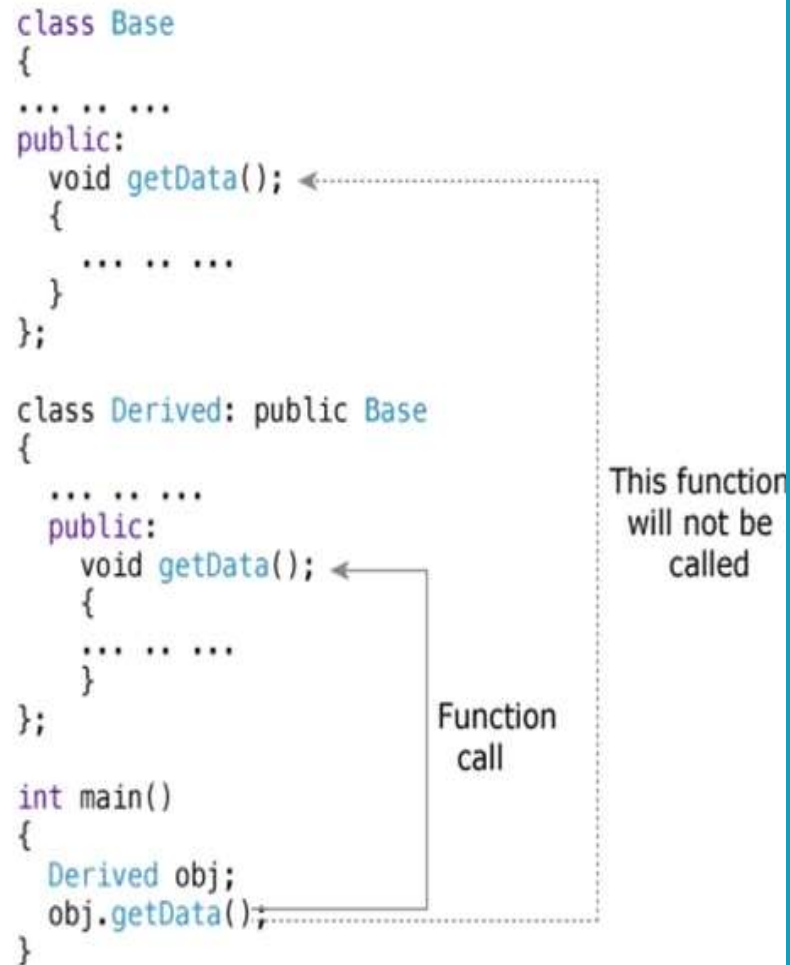
```
class Base
{
    ... ..
public:
    void getData();
    {
        ... ..
    }
};

class Derived: public Base
{
    ... ..
public:
    void getData();
    {
        ... ..
    }
};

int main()
{
    Derived obj;
    obj.getData();
}
```

Function call

This function will not be called



Parent class method is not called

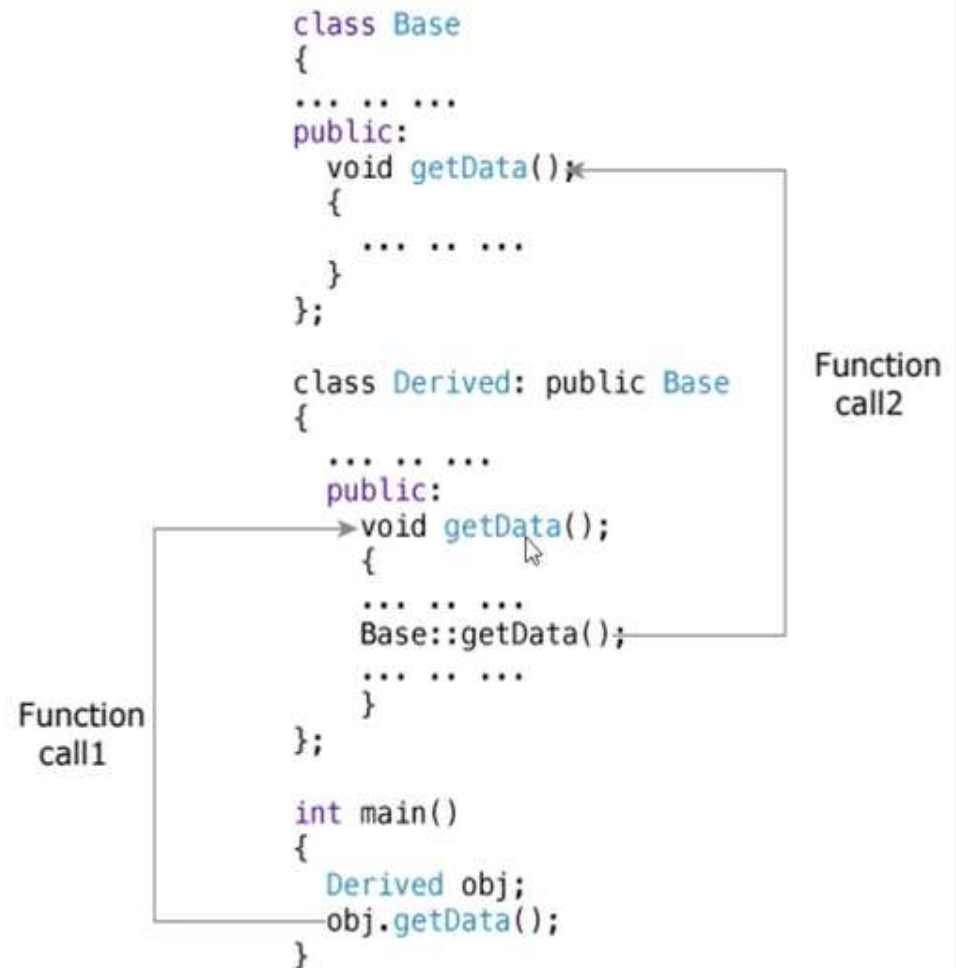
```
class Base
{
    ... ..
public:
    void getData();
    {
        ... ..
    }
};

class Derived: public Base
{
    ... ..
public:
    void getData();
    {
        Base::getData();
        ... ..
    }
};

int main()
{
    Derived obj;
    obj.getData();
}
```

Function call1

Function call2



Parent class method is called

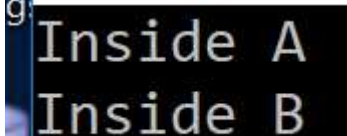
Function Overriding

```
#include <iostream>
using namespace std;

class A{
    public:
        void Print(){
            cout<<"Inside A"<<endl;
        }
};

class B:public A{
    public:
        void Print(){
            cout<<"Inside B"<<endl;
        }
};

int main(void) {
    A a;
    a.Print();
    B b;
    b.Print();
}
```



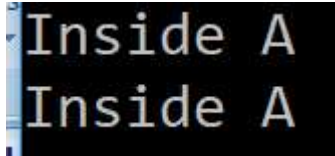
11

```
#include <iostream>
using namespace std;

class A{
    public:
        void Print(){
            cout<<"Inside A"<<endl;
        }
};

class B:public A{
    public:
};

int main(void) {
    A a;
    a.Print();
    B b;
    b.Print();
}
```



If function does not exists in derived class then base class function is called

Virtual Function & Polymorphism

12

- Polymorphism means **same** action but **different** reaction/reply
- In C++, polymorphism refers to the property by which **objects** belonging to **different classes** are able to **respond** to the **same** message, but in **different** forms
- Polymorphism is also known as **late** binding/**dynamic** binding/**run-time** binding
- In C++, **two** things are required to achieve polymorphism
 1. A **virtual** function in the **base class**
 2. A **pointer** of the **base class**

Virtual Function & Polymorphism

13

- The function in the base class is declared as **virtual** by using the keyword virtual preceding its normal declaration
- When a function is made virtual, C++ determines which function to use at runtime **based** on the **type of the object** pointed to by the base pointer.

Virtual Function & Polymorphism

```
#include <iostream>
using namespace std;

class A{
    public:
    virtual void Print(){
        cout<<"Inside A"<<endl;
    }
};

class B:public A{
    public:
    void Print(){
        cout<<"Inside B"<<endl;
    }
};

int main(void) {
    A *pa;
    A a;
    pa=&a;
    pa->Print();
    B b;
    pa=&b;
    pa->Print();
}
```

14

Here pa is the pointer to base class. First it points to base class object a. So pa->Print() calls base class method

After that pa is assigned to B class object b. So pa->Print() calls derived class method

As the address generates at runtime the statement pa=&b will be executed at runtime which ultimate creates run-time calling (**dynamic binding**) So a base class pointer can point to any derived class objects at run-time.

Virtual Function

Rules of Virtual Function

- The virtual functions should not be static.
- It must be member of some class.
- A virtual function can be declared as friend for another class.
- Constructors cannot be declared as virtual, but destructors can be declared as virtual.
- They can be accessed by using pointer object.
- The prototype of the base class version of virtual function and derived class function prototype must be identical.
- Base pointer can point to any type of derived object but derived pointer can not point to base class object.
- If virtual function is defined in base class, it is need not be redefine in derived class.

Virtual Function

```
class A{
    public:
    void Print(){
        cout<<"Inside Print A"<<endl;
    }
    void Show(){
        cout<<"Inside Show A"<<endl;
    }
};
class B:public A{
    public:
    void Print(){
        cout<<"Inside Print B"<<endl;
    }
    void Show(){
        cout<<"Inside Show B"<<endl;
    }
};
int main(void) {
    A *pa;
    B b;
    pa=&b;
    pa->Print();
    pa->Show();
}
```

Inside Print A
Inside Show A

16

```
class A{
    public:
    virtual void Print(){
        cout<<"Inside Print A"<<endl;
    }
    void Show(){
        cout<<"Inside Show A"<<endl;
    }
};
class B:public A{
    public:
    void Print(){
        cout<<"Inside Print B"<<endl;
    }
    void Show(){
        cout<<"Inside Show B"<<endl;
    }
};
int main(void) {
    A *pa;
    B b;
    pa=&b;
    pa->Print();
    pa->Show();
}
```

Inside Print B
Inside Show A

AS Print() declared as virtual so pa->Print()
call derived class method

Pure Virtual Function & Abstract Class

17

- Sometimes **implementation** of all **function** cannot be provided in a **base class** because we **don't know** the **implementation**. Such a class is called **abstract class**.
- A **pure virtual function** (or abstract function) in C++ is a virtual function for which **we don't have implementation**, we only **declare it**. A pure virtual function is declared **by assigning 0 in declaration**.
- Some important facts –
 - A **class** is **abstract** if it has **at least one** **pure virtual function**.
 - We can have **pointers** and **references** of **abstract class type**.
 - If we do not **override** the **pure virtual function** in **derived class**, then **derived class** also becomes **abstract class**.
 - Abstract classes **cannot be instantiated**.

Pure Virtual Function & Abstract Class

18

- A pure virtual function is used to make a class **abstract**
- An abstract class is such a class whose **objects** cannot be created
- A virtual function is made '**pure virtual**' by assigning **zero(0)** to the function name. Such a function is also known as 'do-nothing' function
- virtual void show() = 0;

Pure Virtual Function & Abstract Class

```
// concept of Virtual Functions
#include<iostream>
using namespace std;

class Shape
{
public:
    virtual void getArea()=0; // pure virtual function
};

class Circle:public Shape{
public:
    void getArea()
    {
        cout<<"Enter circle radius"<<endl;
        int r;
        cin>>r;
        cout<<"Area of circle is: "<<(3.14*r*r);
    }
};

class Rectangle: public Shape{
public:
    void getArea()
    {
        cout<<"Enter length and breadth to calculate area of rectangle"<<endl;
        int l,b;
        cin>>l;
        cin>>b;
        cout<<"Area of rectangle is: "<<(l*b);
    }
};

int main()
{
    Circle c1;
    c1.getArea();
    Rectangle r1;
    r1.getArea();
}
```

Here getArea() is pure virtual function makes Shape as abstract class.
The getArea() method needs to be defined in derived class.

Pure Virtual Function & Abstract Class

```
#include <iostream>
using namespace std;
class Animal{
public:
    virtual void eat()=0;
};
class Dog:public Animal{
public:
    void eat(){
        cout<<"Dog food"<<endl;
    }
};
class Cat:public Animal{
public:
    void eat(){
        cout<<"Cat food"<<endl;
    }
};
```

```
void Show(Animal *a){
    a->eat();
}
int main()
{
    Animal *pb;
    Dog d;
    pb=&d;
    Show(pb);
    Cat c;
    pb=&c;
    Show(pb);
}
```

```
Dog food
Cat food
```

21

THANK YOU