INDIAN INSTITUTE OF TECHNOLOGY ROORKEE

EEC-101
Programming with C++

Module-5:
Object Oriented Design:

About Subject

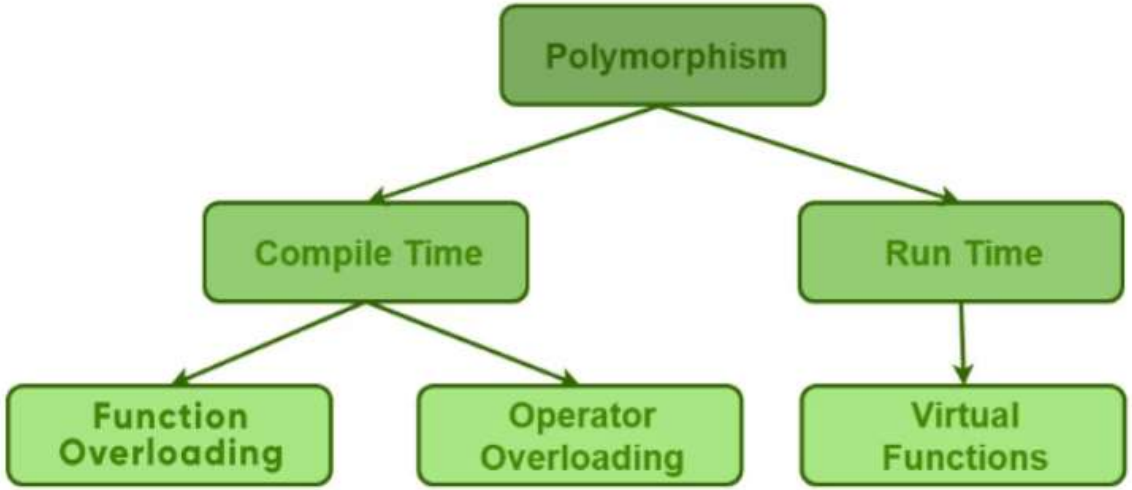
- Object Oriented Programming Concepts
  - Inheritance and composition;
  - Dynamic binding and virtual functions;
  - Polymorphism;
  - Dynamic data in classes.

- 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.

- A real-life example of polymorphism is a person who at the same time can have different characteristics. A man, at the same time, is a father, a husband, and an employee. So the same person exhibits different behavior in different situations. This is called polymorphism.

- Polymorphism is considered one of the important features of Object-Oriented Programming.

- Types of Polymorphism
  - Compile-time Polymorphism
  - Runtime Polymorphism



Polymorphism

- poly' originated from the greek word meaning **many**
- 'morphism' from a greek word meaning **form**

- 'polymorphism' means **many forms**

- In OOP, polymorphism refers to identically named methods (member functions) that have different behavior depending on the type of object they refer.

- polymorphism refers to the possibility of different objects (related via inheritance) to respond differently to same member function call (the member function must have been declared virtual in base class)

- Polymorphism is the process of defining a number of objects of different classes into a group and call the methods to carry out operations of the objects using different function calls.
- In other words, polymorphism means ‘to carry out different processing steps by functions having same messages.
- The keyword virtual is used to perform polymorphism concepts in C++

- polymorphism refers to the possibility of different objects (related via inheritance) to respond differently to same member function call (the member function must have been declared virtual in base class)

- Choosing a function in normal way, during compilation time is called **early binding or static binding or static linkage**

- By default , C++ follows early binding.

Polymorphism with pointers

- Pointers are also central to polymorphism in C++.

- To enable polymorphism, C++ allows **pointer in a base class to point to** either **a base class object or to any derived class object.**

- An example to illustrate how a **pointer is assigned to point to object of the derived class:**

```
class Base_A{
…….
…….
};
class Derived_D : public base_A {
……
……
};
int main()
{
Base_A *ptr; // pointer to Base_A
Derived_D  objd;
ptr= &objd; // indirect reference objd to the pointer
……
……
}
```

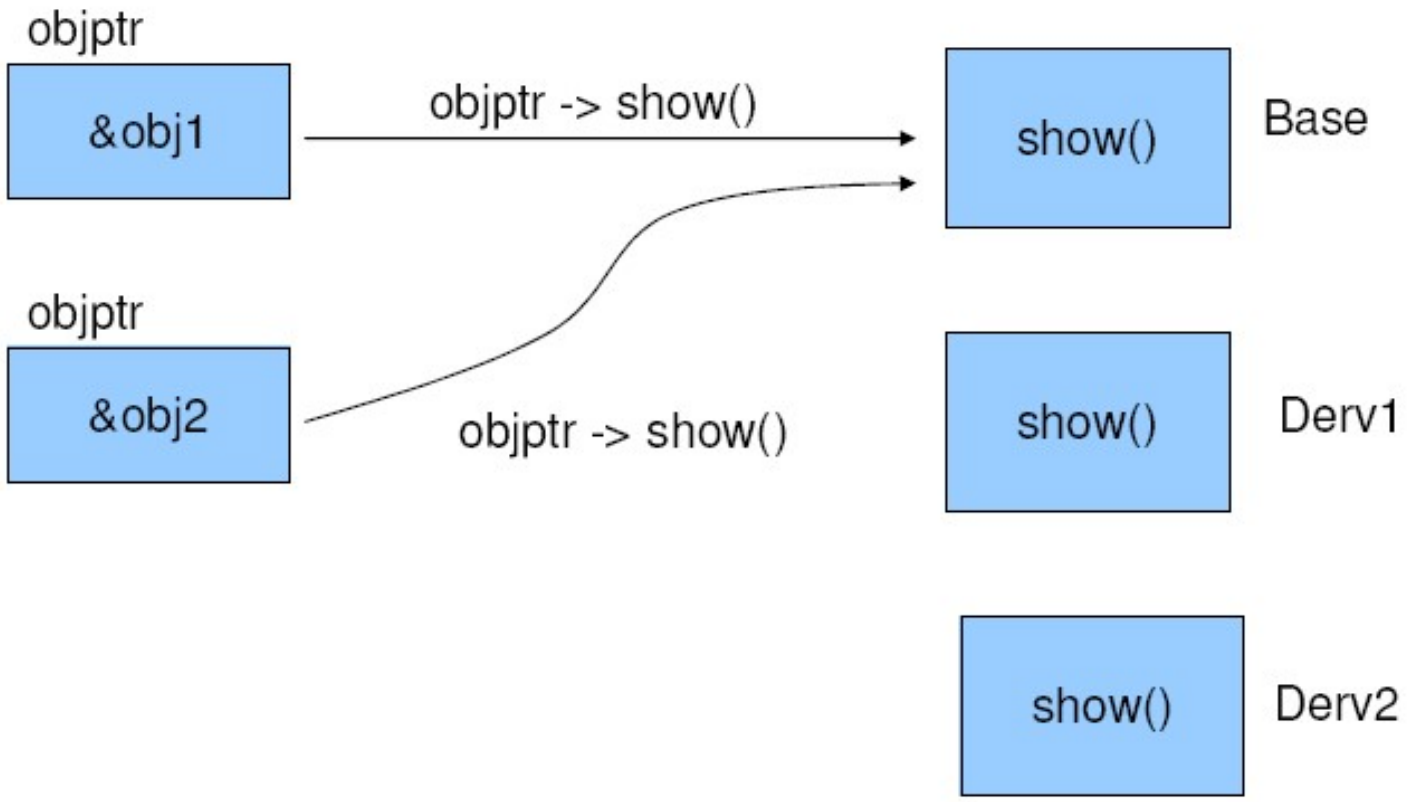
The pointer ptr points to the object of the derived class objd

```
//Static binding, without virtual function
//though pointers pointing to derived class objects
#include <iostream>
using namespace std;
class CPolygon // Base class
{
protected:
int width, height;
public:
void setup (int first, int second){
width= first;
height=second;}
};

class CRectangle: public CPolygon // derived class
{
public:
int area()
{ return (width*height); }
};

class CTriangle: public CPolygon //derived class
{
public:
int area()
{return (width*height / 2); }
};
```

```
int main()
{
CRectangle rectangle;
CTriangle triangle;
CPolygon *ptr_polygon1 = &rectangle; /* base class (Cpolygon) pointer
points to object (rectangle) of the derived class (CRectangle)*/

CPolygon *ptr_polygon2 = &triangle; /* base class (Cpolygon) pointer
points to object (triangle) of the derived class (CTriangle)*/

ptr_polygon1->setup (2,2); // static or compile time binding
ptr_polygon2->setup (2,2); // static or compile time binding

cout << rectangle. area () << endl; //static binding
cout << triangle. area () << endl; //static binding

cin.get(); return 0;}
```

```
4
2
```

Non-Virtual Pointer Access



```cpp
#include<iostream>
using namespace std;
class Base
{ public:
void show() { cout << "Base" << endl; }
};
//*********************************************************
class Derv1 : public Base
{public:
void show() { cout << "Derv1" << endl; }
};
//*********************************************************
class Derv2 : public Base
{ public:
void show() { cout << "Derv2" << endl; }
};
int main()
{
Derv1 obj1; //constructor 1
Derv2 obj2; //constructor 2
Base *objptr; //access with pointers
objptr = &obj1;
objptr -> show(); //print "Base", base class function called
objptr = &obj2;
objptr -> show(); //print "Base", base class function called
return 0; }
```

```
Base
Base

--------------------------------
```

Virtual Functions

Virtual function is one that does not really exist but it appears real in some parts of a program.

A virtual function (also known as virtual methods) is a member function that is declared within a base class and is re-defined (overridden) by a derived class. When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class's version of the method.

- Virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for the function call.
- They are mainly used to achieve Runtime polymorphism.
- Functions are declared with a virtual keyword in a base class.
- The resolving of a function call is done at runtime.

Syntax

```
class user_defined_name{
private:
----------
----------
public:
virtual return_type function_name1(arguments);//function-body outside
virtual return_type function_name2(arguments);
virtual return_type function_name3(arguments);
------------
------------
};
```

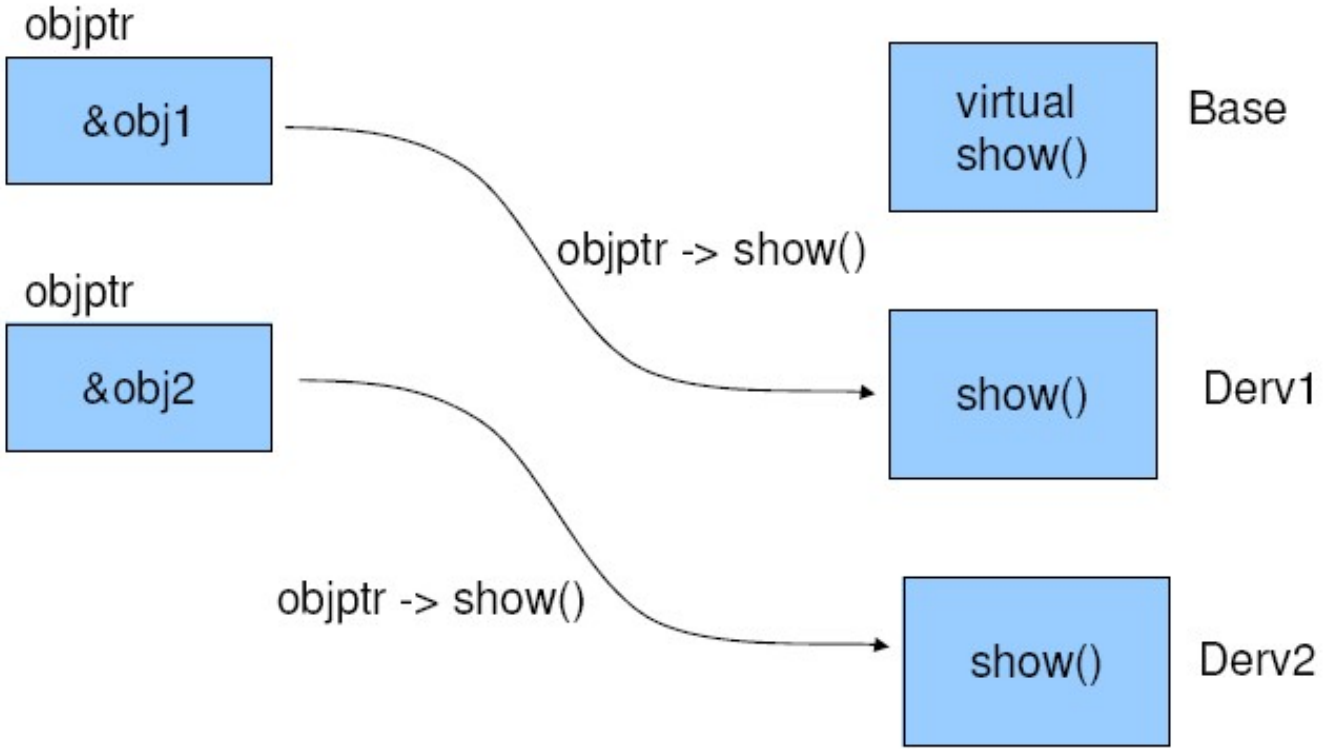
- To make a member function **virtual, the** keyword **virtual** is used in the methods while it is declared in the class definition but not in the member function definition.

```
class Base
{
public:
virtual void show() { cout << ”Base” << endl; }
};
//*************************************************
class Derv1 : public Base
{
public:
void show() { cout << ”Derv1” << endl; }
};
//*************************************************
class Derv2 : public Base
{
public:
```

```cpp
#include<iostream>
using namespace std;
class Base
{
public:
virtual void show() { cout << "Base"<< endl; }
};
//*****************************************************
class Derv1 : public Base
{
public:
void show() { cout << "Derv1" << endl; }
};
//*****************************************************
class Derv2 : public Base
{
public:
void show() { cout << "Derv2" << endl; }
};
int main()
{
Derv1 obj1; //constructor 1
Derv2 obj2; //constructor 2

Base *objptr; //access with pointers

objptr = &obj1;
objptr -> show(); //print „Derv1”, derived class function called

objptr = &obj2;
objptr -> show(); //print „Derv2”, derived class function called

return 0;
}
```

```
Derv1
Derv2
```

Virtual Pointer Access



```cpp
/*Using virtual function Dynamic binding and
pointers pointing to derived class objects */
#include <iostream>
using namespace std;
class CPolygon // base class
{
    protected:
    int width, height;
    public:
    void setup (int first, int second){
    width= first; height= second; }
    virtual int area() // Virtual function declared using
    //int area() // with out Virtual output will be 0 0 0
    { return (0); }
};
class CRectangle: public CPolygon
{
    public:
    int area() // no virtual keyword used
    {return (width*height); }
};
class CTriangle: public CPolygon
{
    public:
    int area() // no virtual keyword used
    {return (width*height/ 2); }
};
```

```cpp
int main()
{
CRectangle rectangle;
CTriangle triangle;
CPolygon polygon;
CPolygon *ptr_polygon1 = &rectangle;
CPolygon *ptr_polygon2 = &triangle;
CPolygon *ptr_polygon3 = &polygon;

ptr_polygon1->setup (2,2); // compile time binding
ptr_polygon2->setup (2,2); // compile time binding
ptr_polygon3->setup (2,2); // compile time binding

cout << ptr_polygon1->area () << endl; // dynamic binding
cout << ptr_polygon2->area () << endl; // dymamic binding
cout << ptr_polygon3->area () << endl; // dymamic binding

cin.get();
return 0;}
```

```
4
2
0
```

Late Binding

- Choosing functions during execution time is called **late binding or dynamic binding or dynamic linkage**
- It provides increased power and flexibility
- Late binding is implemented through virtual functions
- An object of a class must be declared either as a pointer to a class or as a reference to a class

Abstract Class
and
Pure Virtual Function

- Sometimes implementation of all functions cannot be provided in a base class because we don't know the implementation. Such a class is called an abstract class.
- For example, let Shape be a base class. We cannot provide the implementation of function draw() in Shape, but we know every derived class must have an implementation of draw().
- Similarly, an Animal class doesn't have the implementation of move() (assuming that all animals move), but all animals must know how to move. We cannot create objects of abstract classes.
- A pure virtual function (or abstract function) in C++ is a virtual function for which we can have an implementation, But we must override that function in the derived class, otherwise, the derived class will also become an abstract class.
- A pure virtual function is declared by assigning 0 in the declaration.

Example:

```
class Base //abstract base class
{   public:
virtual void show() = 0; //pure virtual function
};
//*******************************************************
class Derv1 : public Base
{   public:
void show() { cout << ”Derv1” << endl; }
};
//*******************************************************
class Derv2 : public Base
{   public:
void show() { cout << ”Derv2” << endl; }
};
```

```
int main()
{
Derv1 obj0;        //constructor 1
Derv2 obj1;        //constructor 2

Base *arr[2]; //access with pointers, object is not created!

arr[0] = &obj0;
arr[1] = &obj1;

arr[0] -> show();
arr[1] -> show();

cin.get()
return 0;
```

```
//Polymorphism Abstract Class with pure virtual function
#include<iostream>
using namespace std;

class CPolygon{
// Abstract Class having pure virtual function
    protected:
    int width, height;

    public:
    void setup (int first, int second)
    {width= first; height= second; }

    virtual int area () = 0;//Pure virtual function
};

class CRectangle: public CPolygon
{
public:
int area() { return (width *height); }
};

class CTriangle: public CPolygon
{
public:
int area () { return (width*height / 2); }
};
```

```
int main()
{
CRectangle rectangle;
CTriangle triangle;

CPolygon *ptr_polygon1 = & rectangle;
CPolygon *ptr_polygon2 = &triangle;

//CPolygon *ptr_polygon3;
// error: declaring object of Abstract class

ptr_polygon1->setup (2,2);
ptr_polygon2->setup (2,2);

cout << ptr_polygon1->area () << endl;
cout << ptr_polygon2->area () << endl;

//cout << ptr_polygon3->area () << endl;
// error: ptr_polygon3 undeclared
cin.get();
return 0;}
```

```
4
2
```

Summary of Virtual functions

- used to allow polymorphism
- works for inherited classes
- declare a function as virtual in base class
- then override this function in each derived class
- while invoking a virtual function through a base-class pointer, the response depends on the actual object pointed by the pointer

Example

- Consider an inheritance hierarchy of shapes.
- base class is shape
- it has a virtual function as
- virtual void draw() const;
- the function draw exists in each of the derived classes such as circle, square, pentagon, triangle etc.
- the function is defined differently in each of the derived class but has same signature
- now consider a declaration

Virtual functions

shape *sptr;

- any call of the type, sptr-> draw() will be decided according to the object pointed
- it is a case of dynamic binding or late binding
- if draw() is called via an object then it is a case of static binding e.g.

square sobj;

sobj.draw();

- polymorphism refers to the possibility of different objects (related via inheritance) to respond differently to same member function call (the member function must have been declared virtual in base class)
- overriding a non-virtual function of base class does not lead to polymorphic behaviour

Abstract classes

Abstract classes

- a class from which we do not plan to make any objects
- meant for inheritance only
- may be called abstract base class
- classes from which objects are to be created - concrete classes
- abstract classes are too general to define real objects
- they only provide a common root for making an inheritance family of some concrete classes
- a class is made abstract by declaring one or more of its virtual functions as pure virtual such as

  virtual float area() const = 0;

Example

```
employee e, *eptr;// base class
eptr = &e;
prof p, *pptr;// derived class
pptr = &p;
eptr->print(); //base class print used;
pptr->print(); //derived class print used
eptr = &p;
eptr->print(); //derived class print used if declared virtual in base class
```

Thanks