Polymorphism

Polymorphism: one object can show more than one form or different form.

Concepts:

Static Polymorphism (Early binding): The function call will be linked with function definition during Compile time.
Example: Function Overloading

Dynamic Polymorphism (Late binding): The function call will be linked with function definition during Run time.
Example: Virtual Function

Virtual Function:
Virtual function is a member function that is declared within the base class and redefined by the derived class both have the same function name with same return type and number of arguments then declare the function in base class as virtual.
By doing so, C++ determines which function to use at run time based on the **type of content** (or) objected pointed to by the base pointer instead of type of the pointer.

```
Class Base
{
Public:
Virtual  Void print()
{
Cout<<"base class function\n";
}
};
Class Derived: public Base
{
Public:
Void print()
{
Cout<<"derived class function\n";
}
};
```

```
Void main()
{
Base *b p; // base pointer
Base b1; // base object
Derived d1;// derived object

Bp=&b1;//base pointer points to base object
Bp->print();//base calss print() is called

Bp=&d1;//  base pointer points to derived object
Bp->print();//derived  calss print() is called
}
```

Output:
Base class function
Derived class function

Note : base pointer can point to any type of derived object. The reverse is not possible.

PURE VIRTUAL FUNCTION

- **Pure virtual function is a function with no definition.**

Or

Pure virtual function is a function that are only declared but not defined in the base class.

- A pure virtual function will always be equals to zero in its declaration
  Syntax:
  Virtual returntype function name(argument list) = 0 ;

- In case of pure virtual function the compiler requires each derived class to define the function.

A class containing at least one pure virtual function is said to be an Abstract Base Class. We cannot create any instance (object) for an abstract class.

PURE VIRTUAL FUNCTION

```
Class Base
{
Public:
Virtual void print()=0;  // not defining
};
Class Derived : public Base
{
Public:
Void print()
{
Cout<<"hello"
}
};
Void main()
{
Derived d1;
D1.print();
}
```

Output:
Hello

- Abstract base class (consists of atleast one pure virtual function)

VIRTUAL DESTRUCTOR

- Destructor is executed whenever an object is destroyed or terminated.
- Using **virtua**l for destructor in the base class we get correct invocation
- Order of execution of destructors in inheritance is:
  - Derived class destructor
  - Base class destructor

```
Class Base
{
Public:
Base()
{
Cout<<”base class constructor\n”
}
Virtual ~Base()
{
Cout<<”base class destructor\n”;
}
};
Class Derived
{
Public:
Derived()
{
Cout<<”derived class constructor\n”
}
~Derived()
{
Cout<<”Derived class destructor\n”;
}
};

Void main()
{
Base *bp=new Derived(); //base pointer points to derived object
Delete bp; //  deallocates memory
}
```

Output:
Base class constructor
Derived class constructor
Derived class destructor
Base class destructor