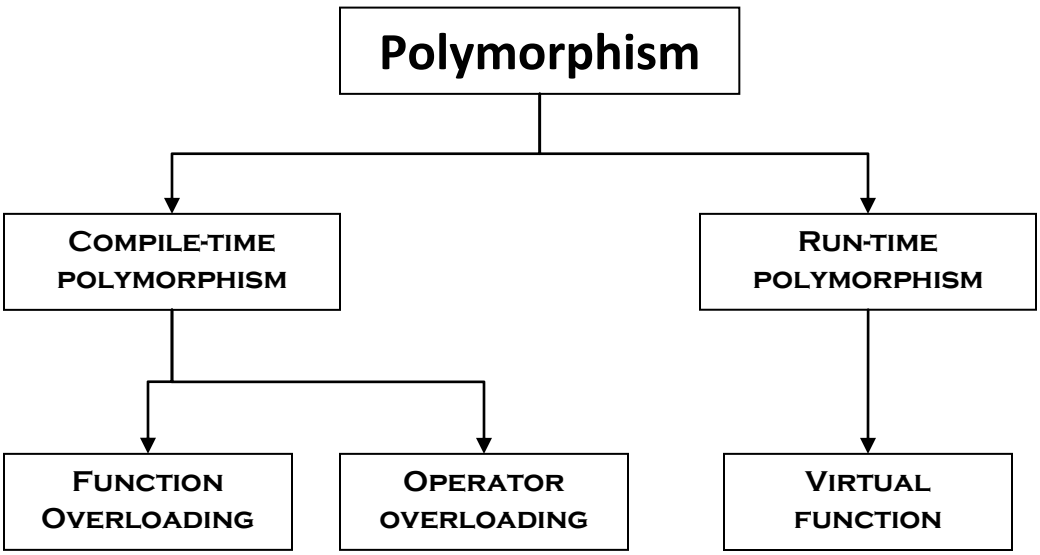
Program Name	**Computer Engineering Program Group**
Semester	**FOURTH**
Course Title	**Object Oriented Programming Using C++**
Course Code	**BTCOE404(1)**

Unit-3 Polymorphism

Polymorphism

Polymorphism -

- The word polymorphism is derived from the greek meaning many forms.
- It allows a single name to be used for more than one related purposes which are technically different. One Thing for Many Purposes/Forms.



Polymorphism is categorized into two types-

1] Compile-time polymorphism

2] Run-time polymorphism

1] Compile-Time Polymorphism-

The concept of polymorphism is implemented by using the overloaded function and operators.

In function or operator overloading the complier decides which function to call and pass an argument to function these all are done at **compile time** so this is compile time polymorphism. It is also called as early/static binding.

1. Function Overloading

- Overloading means uses a one thing for different purposes.
- Function overloading means **more than one function having the same name for different purposes.**
- **In function overloading the function have the same name but signature of the function are must be different.**

The signature of functions are different in following ways:-

1) No Of Parameters :-

void add(int,int);

void add(int,int,int);

void add(int,int,int.int);

In above example all three functions have same name add but the number of parameters for each functions are different.

2) By Type of Parameters: -

void add(int,int);

void add(float,float);

void add(int,float);

In above example numbers of parameters are same but the type of parameter of the functions are different.

3) By Sequence of parameters: -

Void add(int, float);

Void add(float,int);

In above example the number of parameters and also the types of parameters are same but the sequence of parameter is different.

Thus by using these three ways compiler differentiates the functions.

In short more than one function has the same name having different signature and purpose is called as function overloading.

```
#include<iostream>
#include<conio.h>
using namespace std;
//Function Prototype Declaration (Signature)
void add(int,int);
void add(int,int,int);
void add(double,double);
void add(double,int);
int main()
{
cout<<"function Overloading";
//Function call
add(30,40,60);          //no of parameters
add(10,20);
add(10.1,10.2);                 //Type of parameters
add(20.2,20);           //Sequence of Parameters
return(0);
}
//Function Definition
void add(int a, int b)                  //2 parameters of type integer
{
        int Total;
        Total=a+b;
        cout<<"Total="<<Total<<"\n";
}
void add(int a, int b, int c)     //3 parameters of type integer
{
        int Total;
```

```
        Total=a+b+c;
        cout<<"Total="<<Total<<"\n";
}
void add(double a,double b)          //2 parameters of type double
{
        float Total;
        Total= a+b;
        cout<<"Total="<<Total<<"\n";
}
void add(double a,int b)             //2 parameters of type double
{
        double Total;
        Total= a+b;
        cout<<"Total="<<Total<<"\n";
}
```

2] Run-Time Polymorphism / Dynamic Binding-

The second type of polymorphism is runtime polymorphism in which compiler decides which function to call at run time so it is called as run time polymorphism. It is also called as dynamic or late binding.

Run time polymorphism is implemented by using virtual function.

3.1 Virtual function-

Virtual function allows programmer to declare a function in base class and define that function name in both base and derived classes then function in the base class is declared as virtual,

Syntax-

```
class <class_name>
{
 public:
 virtual <return_type><function_name>(arg s)
 {
  _________________;
```

```
        _____;
      }
     };
```

- Virtual function is define starting with keyword virtual.
- Virtual function is accessed through a pointer to the base class, the compiler decides which function to be use at runtime using base class pointer.

```
class Base
{
public:
        virtual void show()
        {
         cout<<"Base class";
        }
};
class Derv1 : public Base
{
public:
        void show()
        {
         cout<<"Derived class 1";
        }
};
class Derv2 : public Base
{
        public:
        void show()
        {
         cout<<"Derived class 2";
        }
};
```

```
void main()
{
        Base *bptr;
        Base B1;
        Derv1 D1;
        Derv2 D2;
        bptr = &B1;
        bptr --> show();
       bptr = &D1;
        bptr --> show();
        bptr = &D2;
        bptr--> show();
        getch();
       }
```

- In above example we create the derived classes derv1 and derv2 and a pointer in Base class.
- Then we put address of a derived class object in the Base class pointer as,

  Bpt = &derv1;

  Bpt = &derv2;

- Then the member functions of the derived classes are executed at the statement Bptr → show( ) executes different functions depending on the contain of pointer Bpt.
- So the compiler selects which function to call depending on the content of base pointed and not on the type of pointer. This is done at runtime.

Rules for virtual function-

- The virtual function must be member of some class.
- They can not be a static member.
- They are accessed by using pointer.
- The virtual function can not be a friend of another class.
- A virtual function must be define in base class.
- The prototype of virtual function in the base class is must be same in the derived class.

- We cannot have virtual constructor but we can have virtual destructor.

3.2 Pure Virtual function

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.

A pure virtual function is implemented by classes which are derived from an Abstract class.

A class is abstract if it has at least one pure virtual function.

Example:

```
#include<iostream>
using namespace std;
class Base                                          //Abstract Base Class
{
public:
   virtual void fun() = 0;        //Pure Virtual Function (=0 no definition)
};
class Derived: public Base              // This class inherits from Base and implements fun()
{
public:
void fun()
{
cout << "Pure Virtual fun() called";
}
};
int main(void)
{
   Derived d;
   d.fun();
   return 0;
}
```

3.3 Abstract Classes-

- Abstract Class is a class which contains at least one Pure Virtual function in it.
- Abstract classes are used to provide an Interface for its sub classes.
- Classes inheriting an Abstract Class must provide definition to the pure virtual function, otherwise they will also become abstract class.

Characteristics of Abstract Class

- Abstract class cannot be instantiated, but pointers and refrences of Abstract class type can be created.
- Abstract class can have normal functions and variables along with a pure virtual function.
- Abstract classes are mainly used for Upcasting, so that its derived classes can use its interface.
- Classes inheriting an Abstract Class must implement all pure virtual functions, or else they will become Abstract too.

Example:

```
//Abstract base class

class Base    //Abstract Class

{

    public:

    virtual void show() = 0;    // Pure Virtual Function

};

class Derived:public Base

{

    public:

    void show()

    {
```

```
      cout << "Implementation of Virtual Function in Derived
class\n";

    }

};

int main()

{

// Base obj;   Compile Time Error we can't create object of
abstract class

Base *b;

    Derived d;

    b = &d;

    b->show();

}
```

Why can't we create Object of an Abstract Class?

When we create a pure virtual function in Abstract class, we reserve a slot for a function in the VTABLE(studied in last topic), but doesn't put any address in that slot. Hence the VTABLE will be incomplete.

As the VTABLE for Abstract class is incomplete, hence the compiler will not let the creation of object for such class and will display an errror message whenever you try to do so.

3.4 Friend Function:

- **A friend function of a class is defined outside that class**' scope but it has the right to access all private and protected members of the class.
- Even though the **prototypes for friend functions appear in the class definition**, friends are not member functions.
- A friend can be a function, function template, or member function, or a class or class template, in which case the entire class and all of its members are friends.
- **A friend function can access the private and protected data of a class**. We declare a friend function using the friend keyword inside the body of the class.

```
class className {
    ... .. ...
    friend returnType functionName(arguments);
    ... .. ...
}
```

Example 2: Add Members of Two Different Classes

```
//Example 2: Add Members of Two Different Classes

// Add members of two different classes using friend functions

#include <iostream>

using namespace std;

// forward declaration

class ClassB;

class ClassA

{

   private:

      int numA;

       // friend function declaration
```

```
        friend int add(ClassA, ClassB);
          public:
       // constructor to initialize numA to 12
       ClassA()
                    {
                            numA=12;
                     }
};
class ClassB
{
   private:
       int numB;
        // friend function declaration
       friend int add(ClassA, ClassB);
   public:
       // constructor to initialize numB to 1
       ClassB()
       {
          numB=1;
                    }
};
// access members of both classes
int add(ClassA objectA, ClassB objectB)
{
   return (objectA.numA + objectB.numB);
```

```
}
int main() {
   ClassA objectA;
   ClassB objectB;
   cout << "Sum: " << add(objectA, objectB);
   return 0;
}
```

3.5 This Pointer

Every object in C++ has access to its own address through an important pointer called this pointer. The this pointer is an implicit parameter to all member functions. Therefore, inside a member function, this may be used to refer to the invoking object.

```
//this pointer points to current object of a class
#include<iostream>
using namespace std;
class Test
{
private:
int x;
public:
void setX (int x)
{
        // 'this' pointer is used to retrieve the object's x hidden by the local variable 'x'
        this->x = x;            //x=x;
}
void print()
{
cout << "x = " << x << endl;
}
```

```
};
int main()
{
Test obj;
int x = 20;
obj.setX(x);
obj.print();
return 0;
}
```