Program : **B.Tech**

Subject Name: **Object Oriented Programming & Methodology**

Subject Code: **CS-305**

Semester: **3rd**

Unit-IV

Polymorphism Introduction:

Polymorphism is another building block of object oriented programming. The philosophy that underlies is "one interface, multiple implementations."
Real life example of polymorphism, a person at a same time can have different characteristic. Like a man at a same time is a father, a husband, a employee. So same person posses have different behavior in different situations. This is called polymorphism. Polymorphism is considered as one of the important features of Object Oriented Programming.
Polymorphism is derived from 2 greek words: **poly** and morphs. The word "poly" means many and **morphs** means forms. So polymorphism means many forms.
In programming languages, polymorphism means that some code or operations or objects behave differently in different contexts.

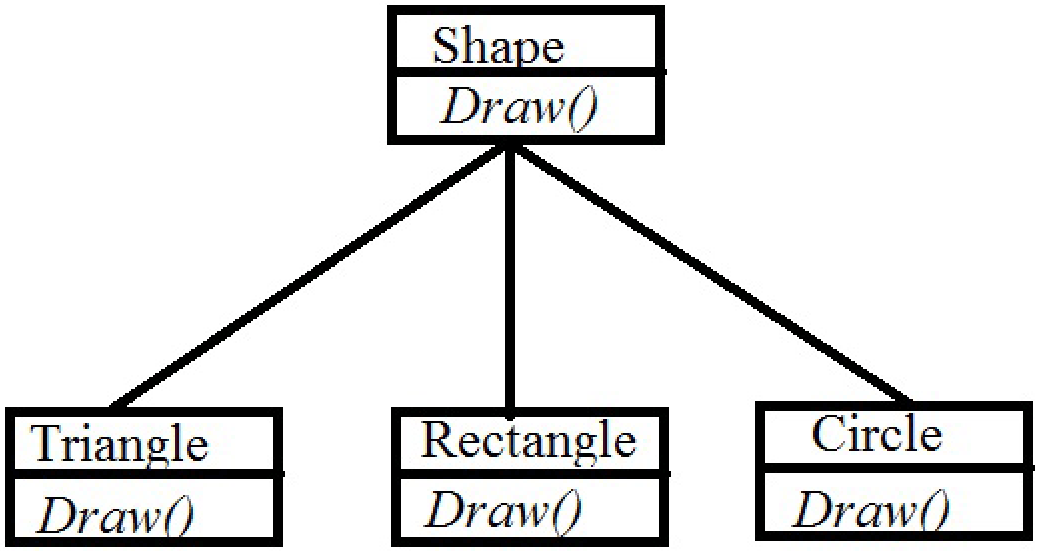


Fig 4.1 polymorphism

For example, the + (plus) operator in C++:

4 + 5 <-- integer addition
3.14 + 2.0 <-- floating point addition
s1 + "bar" <-- string concatenation!

Real life example of Polymorphism in C++

Suppose if you are in class room that time you behave like a student, when you are in market at that time you behave like a customer, when you at your home at that time you behave like a son or daughter, Here one person have different-different behaviors.

Usages and Advantages of Polymorphism

1. Method overloading allows methods that perform similar or closely related functions to be accessed through a common name. For example, a program performs operations on an array of

numbers which can be int, float, or double type. Method overloading allows you to define three methods with the same name and different types of parameters to handle the array of operations.

2. Method overloading can be implemented on constructors allowing different ways to initialize objects of a class. This enables you to define multiple constructors for handling different types of initializations.
3. Method overriding allows a sub class to use all the general definitions that a super class provides and add specialized definitions through overridden methods.
4. Method overriding works together with inheritance to enable code reuse of existing classes without the need for re-compilation.

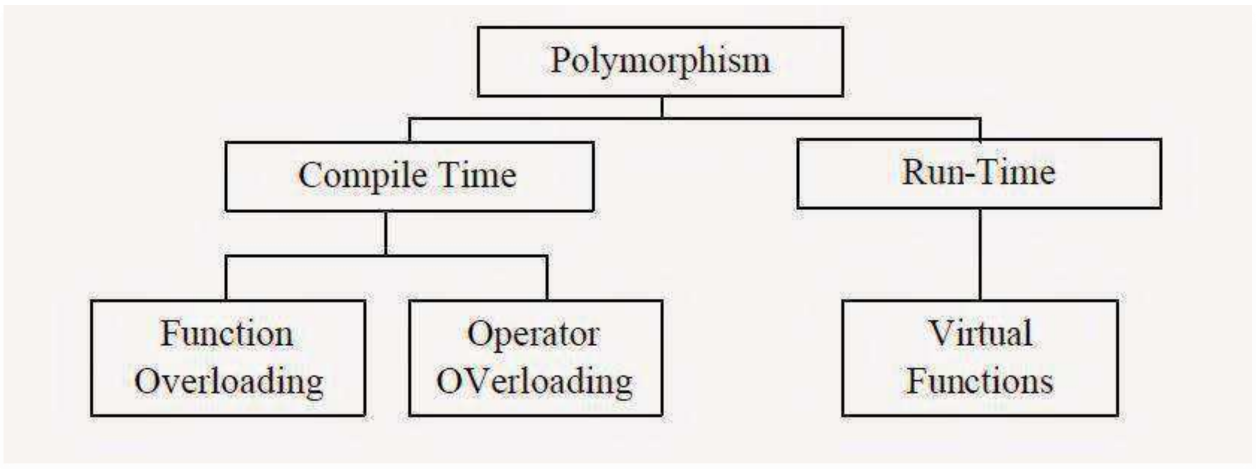


Fig 4.2 Polymorphism Type

Method Overloading and Overriding:

Whenever same method name is exiting multiple times in the same class with different number of parameter or different order of parameters or different types of parameters is known as **method overloading**.

Why method Overloading

Suppose we have to perform addition of given number but there can be any number of arguments, if we write method such as a(int, int)for two arguments, b(int, int, int) for three arguments then it is very difficult for you and other programmer to understand purpose or behaviors of method they can not identify purpose of method. So we use method overloading to easily figure out the program. For example above two methods we can write sum(int, int) and sum(int, int, int) using method overloading concept.

Syntax

class class_Name

```
{
        Returntype  method()
        {
        ..........
        ..........
        }
        Returntype  method(datatype1 variable1)
        {
        ..........
        ..........
        }
        Returntype  method(datatype1 variable1, datatype2 variable2)
        {
        ..........
        ..........
        }

};
```

Different ways to overload the method

- By changing number of arguments or parameters
- By changing the data type

By changing number of arguments

In this example, we have created two overloaded methods, first sum method performs addition of two numbers and second sum method performs addition of three numbers.
Program Function Overloading in C++

```
#include<iostream.h>
#include<conio.h>

class Addition
{
public:
void sum(int a, int b)
{
        cout<<a+b;
}
void sum(int a, int b, int c)
{
        cout<<a+b+c;
}
};
void main()
{
      clrscr();
      Addition obj;
      obj.sum(10, 20);
```

```
        cout<<endl;
        obj.sum(10, 20, 30);
  }
```

Output

```
    30
    60
```

By changing the data type

In this example, we have created two overloaded methods that differs in data type. The first sum method receives two integer arguments and second sum method receives two float arguments.

Method Overloading Program in C++

```
#include<iostream.h>
#include<conio.h>

class Addition
{
        public:
        void sum(int a, int b)
        {
        cout<<a+b;
        }
        void sum(float a, float b)
        {
                cout<<a+b+c;
        }
};
void main()
{
        clrscr();
        Addition obj;
        obj.sum(10, 20);
        cout<<endl;
        obj.sum(10, 20, 30);
}
```

Method Overriding

If derived class defines same method as defined in its base class, it is known as method overriding in C++. It is used to achieve runtime polymorphism. It enables you to provide specific implementation of the method which is already provided by its base class.

C++ Method Overriding Example

Let's see a simple example of method overriding in C++. In this example, we are overriding the eat() function.

```
#include <iostream>
using namespace std;
class Animal
{
            public:
                  void eat()
                  {
                           cout<<"Eating...";
                  }
};
class Dog: public Animal
{
         public:
                  void eat()
                  {
                           cout<<"Eating bread...";
                  }
};
int main()
{

  Dog d = Dog();
  d.eat();
  return 0;
}
```

Output:

```
Eating bread...
```

Difference between Method Overloading and Overriding

Context	**Method Overloading**	**Method Overriding**
Definition	In Method Overloading, Methods of the same class shares the same name but each method must have different number of parameters or parameters having different types and order.	In Method Overriding, sub classes have the same method with same name and exactly the same number and type of parameters and same return type as a super class.
Meaning	Method Overloading means more than one method shares the same name in the class but having different signature.	Method Overriding means method of base class is re-defined in the derived class having same signature.
Behavior	Method Overloading is to “add” or “extend” more to method’s behavior.	Method Overriding is to “Change” existing behavior of method.

Polymorphism	It is a compile time polymorphism.	It is a run time polymorphism.
Inheritance	It may or may not need inheritance in Method Overloading.	It always requires inheritance in Method Overriding.
Signature	In Method Overloading, methods must have different signature.	In Method Overriding, methods must have same signature.
Relationship of Methods	In Method Overloading, relationship is there between methods of same class.	In Method Overriding, relationship is there between methods of super class and sub class.
Criteria	In Method Overloading, methods have same name different signatures but in the same class.	In Method Overriding, methods have same name and same signature but in the different class.
No. of Classes	Method Overloading does not require more than one class for overloading.	Method Overriding requires at least two classes for overriding.

Static and Run Time Polymorphism

Static polymorphism: Static polymorphism refers to an entity existing in different physical forms simultaneously. Static polymorphism involves binding of functions based on the number, type, and sequence of arguments. The various types of parameters are specified in the function declaration, and therefore the function can be bound to calls at compile time. This form of association is called early binding. The term early binding stems from the fact that when the program is executed, the calls are already bound to the appropriate functions.
The resolution of a function call is based on number, type, and sequence of arguments declared for each form of the function. Consider the following function declaration:

```
void add(int , int);
void add(float, float);
```

When the add() function is invoked, the parameters passed to it will determine which version of the function will be executed. This resolution is done at compile time.

Example of Static Polymorphism

1. Method Overloading
2. Operator Overloading

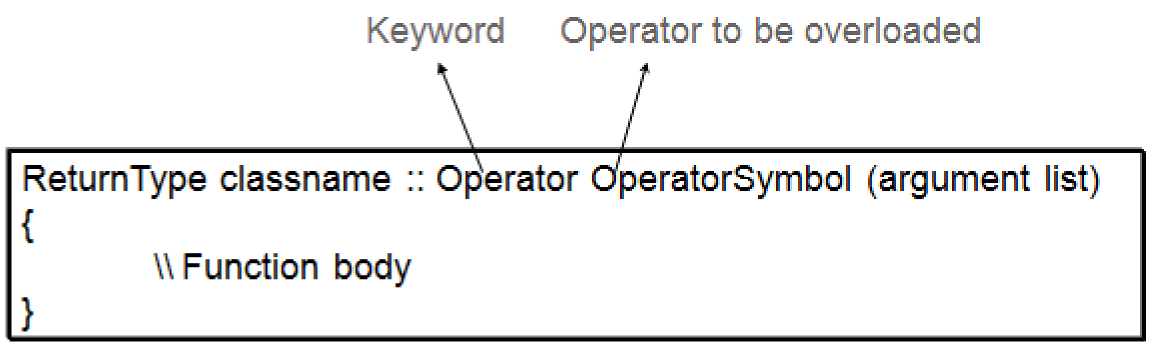
Operator Overloading

Operator overloading is an important concept in C++. It is a type of polymorphism in which an operator is overloaded to give user defined meaning to it. Overloaded operator is used to perform operation on user-defined data type. For example '+' operator can be overloaded to perform addition on various data types, like for Integer, String (concatenation) etc.

Operators that are not overloaded are follows

- scope operator - ::
- sizeof
- member selector - .
- member pointer selector - *
- ternary operator - ?:

Operator Overloading Syntax



```
ReturnType classname :: Operator OperatorSymbol (argument list)
{
        \\ Function body
}
```

Implementing Operator Overloading

Operator overloading can be done by implementing a function which can be :

1. Member Function
2. Non-Member Function
3. Friend Function

Operator overloading function can be a member function if the Left operand is an Object of that class, but if the Left operand is different, then Operator overloading function must be a non-member function.

Operator overloading function can be made friend function if it needs access to the private and protected members of class.

Restrictions on Operator Overloading

Following are some restrictions to be kept in mind while implementing operator overloading.

1. Precedence and associativity of an operator cannot be changed.
2. Arity (numbers of Operands) cannot be changed. Unary operator remains unary, binary remains binary etc.
3. No new operators can be created, only existing operators can be overloaded.
4. Cannot redefine the meaning of a procedure. You cannot change how integers are added

There are two types of operator overloading in C++

- Binary Operator Overloading
- Unary Operator Overloading

Binary Operator Overloading Example

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

class Box {
  public:
    double getVolume(void) {
      return length * breadth * height;
    }
    void setLength( double len ) {
      length = len;
    }
    void setBreadth( double bre ) {
      breadth = bre;
    }
    void setHeight( double hei ) {
      height = hei;
    }

    // Overload + operator to add two Box objects.
    Box operator+(const Box& b) {
      Box box;
      box.length = this->length + b.length;
      box.breadth = this->breadth + b.breadth;
      box.height = this->height + b.height;
      return box;
    }

  private:
    double length;      // Length of a box
    double breadth;     // Breadth of a box
    double height;      // Height of a box
};

// Main function for the program
int main() {
  Box Box1;                // Declare Box1 of type Box
  Box Box2;                // Declare Box2 of type Box
  Box Box3;                // Declare Box3 of type Box
  double volume = 0.0;     // Store the volume of a box here

  // box 1 specification
  Box1.setLength(6.0);
  Box1.setBreadth(7.0);
  Box1.setHeight(5.0);
```

```
   // box 2 specification
   Box2.setLength(12.0);
   Box2.setBreadth(13.0);
   Box2.setHeight(10.0);

   // volume of box 1
   volume = Box1.getVolume();
   cout << "Volume of Box1 : " << volume <<endl;

   // volume of box 2
   volume = Box2.getVolume();
   cout << "Volume of Box2 : " << volume <<endl;

   // Add two object as follows:
   Box3 = Box1 + Box2;

   // volume of box 3
   volume = Box3.getVolume();
   cout << "Volume of Box3 : " << volume <<endl;

   return 0;
}
```

Run Time Polymorphism

Run Time Polymorphism: Dynamic polymorphism refers to an entity changing its form depending on the circumstances. A function is said to exhibit dynamic polymorphism when it exists in more than one form, and calls to its various forms are resolved dynamically when the program is executed. The term late binding refers to the resolution of the functions at run-time instead of compile time. This feature increases the flexibility of the program by allowing the appropriate method to be invoked, depending on the context.

Example of Run time Polymorphism

1. Method Overriding.
2. Virtual Functions.

Static Vs Dynamic Polymorphism

- Static polymorphism is also known as early binding and compile-time polymorphism.
- Dynamic polymorphism is also known as late binding and run-time polymorphism.
- Static polymorphism is considered more efficient and dynamic polymorphism more flexible.
- Statically bound methods are those methods that are bound to their calls at compile time. Dynamic function calls are bound to the functions during run-time. This involves the additional step of searching the functions during run-time. On the other hand, no run-time search is required for statically bound functions.
- As applications are becoming larger and more complicated, the need for flexibility is increasing rapidly. Most users have to periodically upgrade their software, and this could become a very tedious task if static polymorphism is applied. This is because any change in requirements requires

a major modification in the code. In the case of dynamic binding, the function calls are resolved at run-time, thereby giving the user the flexibility to alter the call without having to modify the code.

- To the programmer, efficiency and performance would probably be a primary concern, but to the user, flexibility or maintainability may be much more important. The decision is thus a trade-off between efficiency and flexibility

Virtual Function

A virtual function a member function which is declared within base class and is re-defined (Override) by 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 function.

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

Example of virtual function

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

     class BaseClass
     {

            public:
             virtual void Display()
             {
                   cout<<"\n\tThis is Display() method of Base Class";
             }
             void Show()
             {
                   cout<<"\n\tThis is Show() method of Base Class";
             }

     };
     class DerivedClass : public BaseClass
     {
            public:
             void Display()
             {
                   cout<<"\n\tThis is Display() method of Derived Class";
             }
             void Show()
             {
                   cout<<"\n\tThis is Show() method of Derived Class";
             }

     };
```

```
int main()
{

      DerivedClass D;
      BaseClass *B;          //Creating Base Class Pointer
      B = new BaseClass;
      B->Display();          //This will invoke Display() method of Base Class
      B->Show();             //This will invoke Show() method of Base Class
      B=&D;
      B->Display();          //This will invoke Display() method of Derived Class
                             //bcoz Display() method is virtual in Base Class

      B->Show();             //This will invoke Show() method of Base Class
                            //bcoz Show() method is not virtual in Base Class
     Return 0;
}
```

Output :

```
This is Display() method of Base Class
This is Show() method of Base Class
This is Display() method of Derived Class
This is Show() method of Base Class
```

Rules for Virtual Functions

1. They must be declared in public section of class.
2. Virtual functions cannot be static and also cannot be a friend function of another class.
3. Virtual functions should be accessed using pointer or reference of base class type to achieve run time polymorphism.
4. The prototype of virtual functions should be same in base as well as derived class.
5. They are always defined in base class and overridden in derived class. It is not mandatory for derived class to override (or re-define the virtual function), in that case base class version of function is used.
6. A class may have virtual destructor but it cannot have a virtual constructor.

Pure Virtual Functions

It is possible that you want to include a virtual function in a base class so that it may be redefined in a derived class to suit the objects of that class, but that there is no meaningful definition you could give for the function in the base class.
We can change the virtual function area() in the base class to the following −

```
class Shape
{
     protected:
             int width, height;

    public:
         Shape(int a = 0, int b = 0)
        {
           width = a;
           height = b;
       }

      // pure virtual function
         virtual int area() = 0;
  };
```

The = 0 tells the compiler that the function has no body and above virtual function will be called pure virtual function.