

## **UNIT- IV**

### **Virtual Functions and Polymorphism:**

Polymorphism, Compile time polymorphism: Overloading- Function Overloading, Operator overloading, Run time polymorphism, Virtual Functions.

### **Exception handling:**

Basics of Exception Handling, Types of exceptions, Exception Handling Mechanism, Throwing and Catching Mechanism.

### **Introduction:**

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”**.

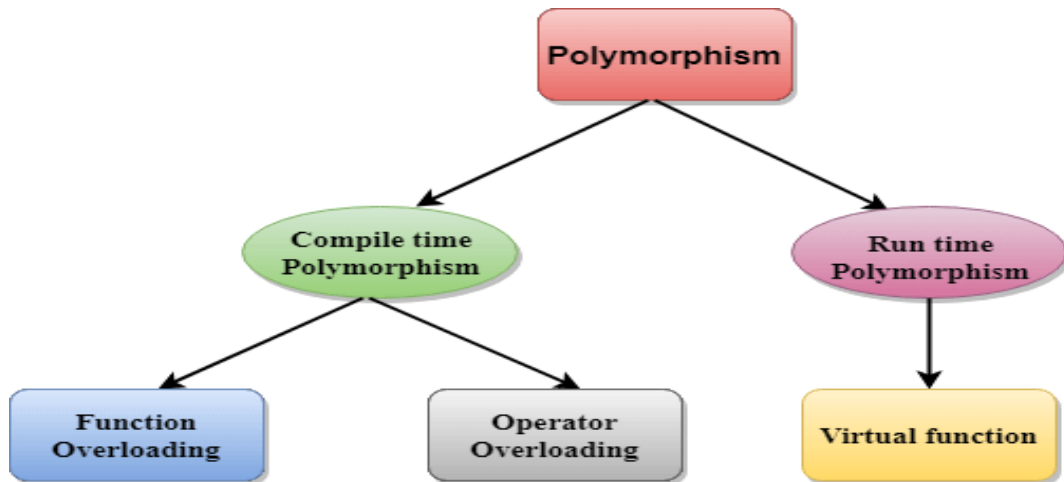
Example:

Real life example of polymorphism, a person at the same time can have different characteristic. Like a man at the same time is a father, a husband, an employee. So the same person possesses different behavior in different situations. This is called polymorphism.

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

### **Type of polymorphism**

1. Compile time polymorphism
2. Run time polymorphism



**1.Compile time polymorphism:**The ability to take one form into many forms at compile time is called “Compile-time Polymorphism”

- In C++ programming you can achieve compile time polymorphism in two ways, which is given below as;
  - i. Function Overloading
  - ii. Operator Overloading’

**(i)Function Overloading:** The process of defining the same function to perform different operations by passing different parameters is called **”Function Overloading”**.

- The overloaded functions are invoked by matching the type and number of arguments.
- This information is available at the compile time and, therefore, compiler selects the appropriate function at the compile time.
- It is achieved by function overloading and operator overloading which is also known as static binding or early binding.

**Example:**

- 1) Void Addition(int a,int b);
- 2) Int Addition(int a,int b,int c)

3) Float Addition(int a,float b)

**//Write a C++ program to demonstrate Function Overloading**

```
#include<iostream>
using namespace std;
class FunctionOverloading
{
    public:
        void  Addition(int a,int b);
        int   Addition(inta,int b,int c);
        float Addition(int a,float b);
};
void FunctionOverloading::Addition(int a,int b)
{
    Cout<<"The Sum of 2 integers :"<<a+b<<"\n";
}
int FunctionOverloading::Addition(int a,int b,int c)
{
    return(a+b+c);
}
float FunctionOverloading::Addition(int a,float b)
{
    return(a+b);
}
int main()
{
    FunctionOverloading f;
```

```

f.Addition(10,20);
int Sum=f.Addition(100,200,300);
cout<<"The Sum of 3 integers:"<<Sum<<"\n";
float result=f.Addition(10,20.75f);
cout<<"The Sum of integer,float :"<<result<<"\n";
return 0;
}

```

### Output:

```

C:\TURBOC3\BIN>TC
The Sum of 2 integers :30
The Sum of 3 integers :600
The Sum of integer,float :30.75

```

Activate Windows  
Go to Settings to activate Windows.

### **Example-2:Write a C++ Program to overload a function area() that finds the area of different solids.**

```

#include<iostream.h>
#include<conio.h>
class AreaofSolids
{
    public:
    int area(int length,int breadth);//Area for Rectangle
    int area(int side);           //Area of Square
    float area(float radius);     //Area of Circle
    float area(int length,float height);//Area of Triangle
};
int AreaofSolids::area(int l,int b)
{
    return(l*b);
}

```

```

int AreaofSolids::area(int s)
{
    return(s*s);
}
float AreaofSolids::area(float r)
{
    return(3.14*r*r);
}
float AreaofSolids::area(int l,float h)
{
    return(0.5*l*h);
}
void main()
{
    clrscr();
    AreaofSolids a;
    cout<<"The Area of Rectangle="<<a.area(10,20);
    cout<<"\nThe Area of Square="<<a.area(10);
    cout<<"\nThe Area of Circle="<<a.area(5.5f);
    cout<<"\nThe Area of Triangle="<<a.area(10,2.5f);
    getch();
}

```

### **Output:**

```

The Area of Rectangle=200
The Area of Square=100
The Area of Circle=94.985001
The Area of Triangle=12.5_

```

## **Operator Overloading in C++**

In C++, we can make operators to work for user defined classes. This means C++ has the ability to provide the operators with a special meaning for a data type,

this ability is known as operator overloading.

For example, we can overload an operator '+' in a class like String so that we can concatenate two strings by just using +.

Other example classes where arithmetic operators may be overloaded are Complex Number, Fractional Number, Big Integer, etc.

The process of overloading involves the following steps:

1. Create a class that defines the data type that is to be used in the overloading operations.
2. Declare the operator function operator op() in the public part of the class. It may be either a member function or a friend function.
3. Define the operator function to implement the required operations.

### **Invoking Operator Functions:**

#### **For Unary Operators:**

- Overloaded operator functions can be invoked by expressions such as:

**“Op x;(or)x Op”;**

Ex: Student s;

-s; (or)s-;

- If the operator function is “friend function” then this statement can be evaluated by the compiler as

**“operator op( x)”**

#### **For Binary Operators:**

- The Binary operator overloading can be represented as

**“X op Y”**

- The above statement can be interpreted by the compiler as

**“X.operator op(Y)”(if operator function is normal function)**

- If the operator function is friend function then it can be evaluated as

**“ operator op(X,Y)”**

```

#include<iostream.h>
#include<conio.h>
//using namespace std;
class Complex
{
    private:
    int real, imag;
    public:
    Complex(int r , int i)
    {
        real = r;
        imag = i;
    }

    // This is automatically called when '+' is used with between two Complex objects
    Complex operator + (Complex &obj)
    {
        Complex res;
        res.real = real + obj.real;
        res.imag = imag + obj.imag;
        return res;
    }
    void print()
    {
        cout << real << " + i" << imag << endl;
    }
};

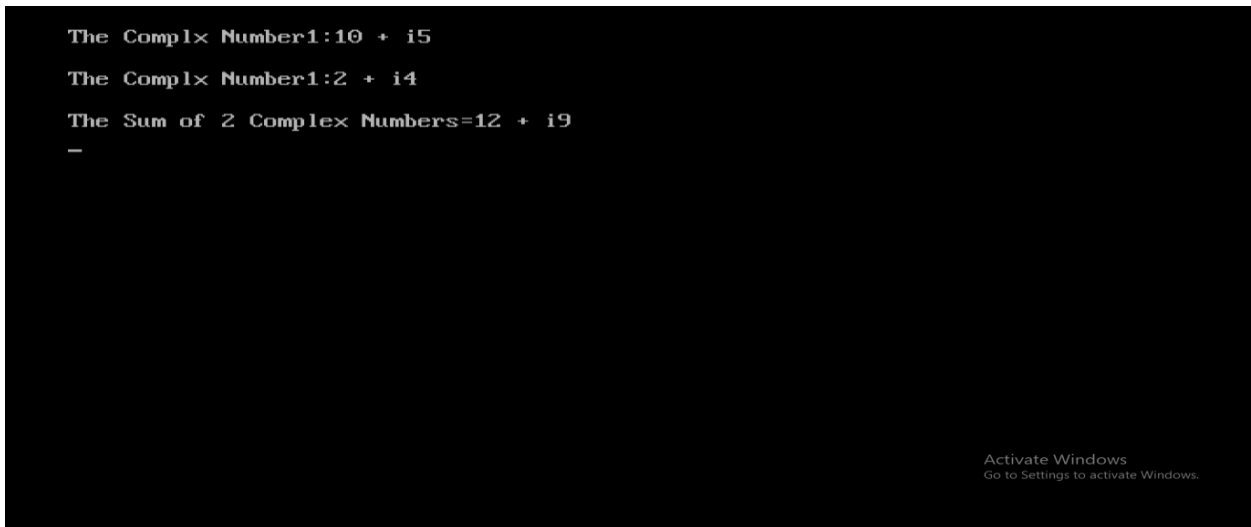
int main()
{
    clrscr();

```

```

Complex c1(10, 5);
cout<<"\nThe Complx Number1:";
c1.print();
Complex c2(2, 4);
cout<<"\nThe Complx Number1:";
c2.print();
Complex c3 = c1 + c2; // An example call to "operator+"
cout<<"\nThe Sum of 2 Complex Numbers=";
c3.print();
return 0;
}

```



The screenshot shows a dark-themed terminal window with the following output:

```

The Complx Number1:10 + i5
The Complx Number1:2 + i4
The Sum of 2 Complex Numbers=12 + i9
-

```

In the bottom right corner, there is a watermark that reads: "Activate Windows Go to Settings to activate Windows."

### **Example2:**

/\*C++ program to add two distances using binary plus (+) operator overloading.\*/

```

#include<iostream>

using namespace std;

class Distance
{
    private:

```



```

        int feet,inches;

    public:

//function to read distance
void readDistance(void)
{
    cout << "Enter feet: ";

    cin >>feet;

    cout << "Enter inches: ";

    cin >>inches;

}

//function to display distance
void dispDistance(void)
{
    cout << "Feet:" << feet << "\t" << "Inches:" << inches << endl;

}

//add two Distance using + operator overloading
Distance operator+(Distance &dist1)
{
    Distance tempD;    //to add two distances

    tempD.inches= inches + dist1.inches;

    tempD.feet = feet  + dist1.feet + (tempD.inches/12);

    tempD.inches=tempD.inches% 12;

    return tempD;
}

```

```

    }
};

int main()
{
    Distance D1,D2,D3;

    cout << "Enter first distance:" << endl;

    D1.readDistance();

    cout << endl;

    cout << "Enter second distance:" << endl;

    D2.readDistance();

    cout << endl;

    //add two distances

    D3=D1+D2;


    cout << "Total Distance:" << endl;

    D3.dispDistance();

    cout << endl;

    return 0;

}

```

### **Example 3:**

**Write a C++ Program to Demonstrate Unary Operator Overloading using friend function.**

```
#include<iostream>
```

```
using namespace std;

class Space
{
    int x;

    int y;

    int z;

    public:

        void getdata(int a,int b,int c);

        void display(void);

        friend void operator-(space &s);

};

void space :: getdata(int a,int b,int c)
{
    x=a;

    y=b;

    z=c;

}

void space :: display(void)
{
    cout<<x<<" ";

    cout<<y<<" ";

    cout<<z<<"\n";

}
```

```

void operator-( space &s)
{
    s.X=-s.X;
    s.y=-s.y;
    s.Z=-s.Z;
}

int main()
{
    space s;
    s.getdata(10,-20,30);
    cout<<"\nBefore  S obeject Values : ";
    s.display();
    -s;
    cout<<"\nAfter S object Values :";
    s.display();
    return 0;
}

```

## **Runtime Polymorphism**

**Runtime polymorphism:** This type of polymorphism is achieved by Function Overriding.

**Function Overriding** on the other hand occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be “**Overridden**”.

## **C++ Virtual function**

- A C++ virtual function is a member function in the base class that you redefine in a derived class. It is declared using the virtual keyword.
- It is used to tell the compiler to perform dynamic linkage or late binding on the function.
- There is a necessity to use the single pointer to refer to all the objects of the different classes. So, we create the pointer to the base class that refers to all the derived objects. But, when base class pointer contains the address of the derived class object, always executes the base class function. This issue can only be resolved by using the 'virtual' function.
- A 'virtual' is a keyword preceding the normal declaration of a function.
- When the function is made virtual, C++ determines which function is to be invoked at the runtime based on the type of the object pointed by the base class pointer.

### **Rules of Virtual Function**

- Virtual functions must be members of some class.
- Virtual functions cannot be static members.
- They are accessed through object pointers.
- They can be a friend of another class.
- A virtual function must be defined in the base class, even though it is not used.
- The prototypes of a virtual function of the base class and all the derived classes must be identical. If the two functions with the same name but different prototypes, C++ will consider them as the overloaded functions.
- We cannot have a virtual constructor, but we can have a virtual destructor
- Consider the situation when we don't use the virtual keyword.

```
#include <iostream>
```

```
using namespace std;
```

```
class A
```

```
{
```

```
    int x=5;
```

```
    public:
```

```
        void display()
```

```

        {
            cout << "Value of x is : " << x<<std::endl;
        }
};

class B: public A
{
    int y = 10;
    public:
    void display()
    {
        cout << "Value of y is : " <<y<< std::endl;
    }
};

int main()
{
    A *a;
    B b;
    a = &b;
    a->display();
    return 0;
}

```

**Output:**

Value of x is : 5

**Explanation:** In the above example, \* a is the base class pointer. The pointer can only access the base class members but not the members of the derived class. Although C++ permits the base pointer to point to any object derived from the base class, it cannot directly access the members of the derived class. Therefore, there is a need for virtual function which allows the base pointer to access the members of the derived class.

### **C++ virtual function Example**

Let's see the simple example of C++ virtual function used to invoke the derived class in a program.

```
#include <iostream>

using namespace std;

class A
{
    public:
    virtual void display()
    {
        cout << "Base class is invoked"<<endl;
    }
};

class B:public A
{
    public:
    void display()
    {
```

```

        cout << "Derived Class is invoked"<<endl;
    }

};

int main()
{
    A* a;    //pointer of base class

    B b;    //object of derived class

    a = &b;

    a->display(); //Late Binding occurs
}

```

Output:

Derived Class is invoked

## **Pure Virtual Function**

- A virtual function is not used for performing any task. It only serves as a placeholder.
- When the function has no definition, such function is known as "do-nothing" function.
- The "do-nothing" function is known as a “pure virtual function”. A pure virtual function is a function declared in the base class that has no definition relative to the base class.
- A class containing the pure virtual function cannot be used to declare the objects of its own, such classes are known as abstract base classes.
- The main objective of the base class is to provide the traits to the derived classes and to create the base pointer used for achieving the runtime polymorphism.

**Pure virtual function can be defined as:**



**virtual void display() = 0;**

```
#include <iostream>
```

```
using namespace std;
```

```
class Base
```

```
{
```

```
    public:
```

```
        virtual void show() = 0;
```

```
};
```

```
class Derived : public Base
```

```
{
```

```
    public:
```

```
        void show()
```

```
        {
```

```
            cout << "Derived class is derived from the base class." << std::endl;
```

```
        }
```

```
};
```

```
int main()
```

```
{
```

```
    Base *bptr;
```

```
    //Base b;
```

```
    Derived d;
```

```
    bptr = &d;
```

```
    bptr->show();
```

```

        return 0;
    }

```

Output:

Derived class is derived from the base class.

**Note:** In the above example, the base class contains the pure virtual function. Therefore, the base class is an abstract base class. We cannot create the object of the base class.

## Difference between Virtual Function and Pure Virtual Function

BASIS FOR COMPARISON	VIRTUAL FUNCTION	PURE VIRTUAL FUNCTION
Basic	'Virtual function' has their definition in the base class.	'Pure Virtual Function' has no definition in the base class.
Declaration	virtual  funct_name(parameter_list)  { . . . . . };	virtual  funct_name(parameter_list)=0;
Derived class	All derived classes may or may not override the	All derived classes must override the virtual function of

BASIS FOR COMPARISON	VIRTUAL FUNCTION	PURE VIRTUAL FUNCTION
	virtual function of the base class.	the base class.
Effect	Virtual functions are hierarchical in nature; it does not affect compilation if any derived classes do not override the virtual function of the base class.	If all derived classes fail to override the virtual function of the base class, the compilation error will occur.
Abstract class	No concept.	If a class contains at least one pure virtual function, then it is declared abstract.

## **Exception Handling in C++**

Errors can be broadly categorized into two types. We will discuss them one by one.

1. Compile Time Errors
2. Run Time Errors

**Compile Time Errors** – Errors caught during compiled time is called Compile time errors.

**Example:** Compile time errors include library reference, syntax error or incorrect class import.

**Run Time Errors** - They are also known as exceptions. An exception caught during run time creates serious issues.

**Example:** User divides a number by zero, this will compile successfully but an exception or run time error will occur due to which our applications will be crashed. In order to avoid this we'll introduce “**Exception handling technics**” in our code.

In C++, Error handling is done using three keywords(or)three techniques.  
They are:

1. try
2. catch
3. throw

### **1.try block**

The code which can throw any exception is kept inside(or enclosed in) a **try** block. Then, when the code will lead to any error, that error/exception will get caught inside the **catch** block.

**Syntax:**

```
try
{
    //Code which raise the Exception
}
```

### Example:

```
int a=10,b=0;
try
{
    int c=a/b;(Division-by-zero Exception error will be raised)
}
```

### 2.catch block

- block is intended to catch the error and handle the exception raised by try block.
- We can have multiple catch blocks to handle different types of exception and perform different actions when the exceptions occur.
- For example, we can display descriptive messages to explain why any particular exception occurred.

### Syntax:

```
catch(Exceptionname ExceptionObject)
{
    //Code to handle exception
}
```

### Example:

```
catch(DivisionByZero e)
{
    Cout<<e;
}
```

### 3.throw Statement

- Whenever we have the exception in try block, if we want to handle that exception explicitly by ourselves then that can be thrown using “throw” statement.
- It is used to throw exceptions to exception handler i.e. it is used to communicate information about error.

- A **throw** expression accepts one parameter and that parameter is passed to handler.
- **throw** statement is used when we explicitly want an exception to occur, then we can use **throw** statement to throw or generate that exception.

Syntax:

```
throw ExceptionObject;
```

## **Throwing Exceptions**

- Exceptions can be thrown anywhere within a code block using **throw** statement.
- The operand of the throw statement determines a type for the exception and can be any expression and the type of the result of the expression determines the type of exception thrown.

Following is an example of throwing an exception when dividing by zero condition occurs –

```
double division(int a, int b)
{
    if( b == 0 )
    {
        throw "Division by zero condition!";
    }
    return (a/b);
}
```

## **Catching Exceptions**

- The **catch** block following the **try** block catches any exception.
- We can specify what type of exception you want to catch and this is determined by the exception declaration that appears in parentheses following the keyword catch.

```
try
{
    // protected code
}
catch( ExceptionName e )
{
    // code to handle ExceptionName exception
}
```

```
}
```

### **Example:**

```
#include <iostream>
#include<conio.h>
using namespace std;
int main()
{
    int a=10, b=0, c;
    // try block activates exception handling
    try
    {
        if(b == 0)
        {
            // throw custom exception
            throw "Division by zero not possible";
            c = a/b;
        }
    }
    catch(char* ex) // catches exception
    {
        cout<<ex;
    }
    return 0;
}
```

### **Using Multiple `catch` blocks**

Below program contains multiple `catch` blocks to handle different types of exception in different way.

```
#include <iostream>
#include<conio.h>
using namespace std;
```

```

int main()
{
    int x[3] = {-1,2};
    for(int i=0; i<2; i++)
    {
        int ex = x[i];
        try
        {
            if (ex > 0)
                // throwing numeric value as exception
                throw ex;
            else
                // throwing a character as exception
                throw 'ex';
        }
        catch (int ex) // to catch numeric exceptions
        {
            cout << "Integer exception\n";
        }
        catch (char ex) // to catch character/string exceptions
        {
            cout << "Character exception\n";
        }
    }
}

```

**Note:** The try block can throw any number of Exceptions but the catch block can handle only one Exception at a time.