Unit 3 Polymorphism

Polymorphism- Introduction to Polymorphism, Types of Polymorphism, Operator Overloading concept of overloading, operator overloading, Overloading Unary Operators, Overloading Binary Operators, Data Conversion, Type casting (implicit and explicit), Pitfalls of Operator Overloading and Conversion, Keywords explicit and mutable.

Function overloading, Run Time Polymorphism- Pointers to Base class, virtual function and its significance in C++, pure virtual function and virtual table, virtual destructor, abstract base class.

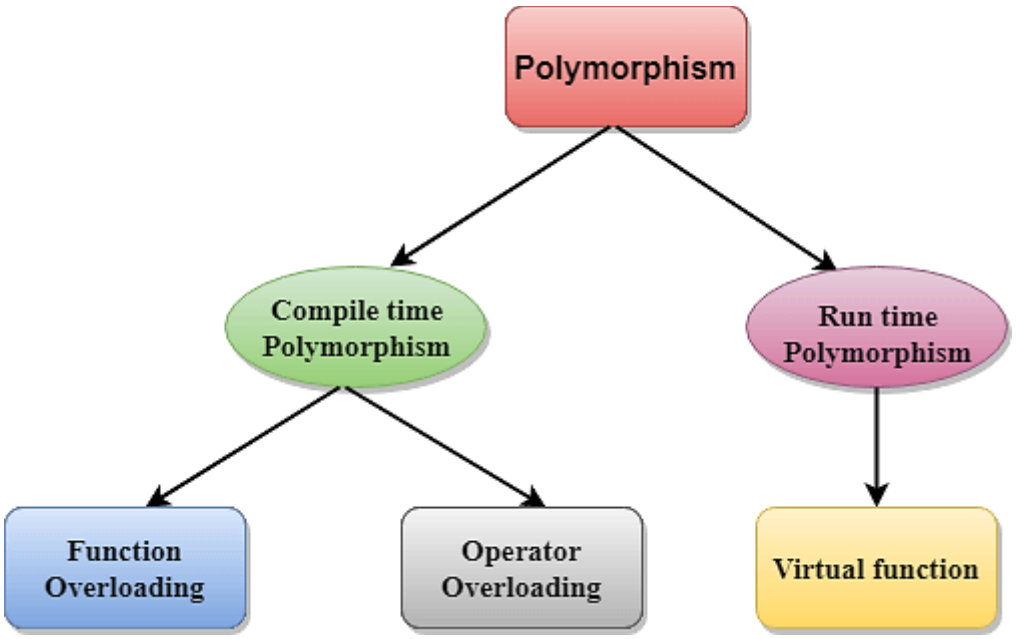
3. 1 Introduction to Polymorphism

The term "Polymorphism" is the combination of "poly" + "morphs" which means many forms. It is a greek word. In object-oriented programming, we use 3 main concepts: inheritance, encapsulation, and polymorphism.

Real Life Example of Polymorphism

Let's consider a real-life example of polymorphism. A lady behaves like a teacher in a classroom, mother or daughter in a home and customer in a market. Here, a single person is behaving differently according to the situations.

3.2 There are two types of polymorphism in C++:



Compile time polymorphism:

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.

Run time polymorphism:

Run time polymorphism is achieved when the object's method is invoked at the run time instead of compile time. It is achieved by method overriding which is also known as dynamic binding or late binding.

Differences b/w compile time and run time polymorphism

Compile time polymorphism	Run time polymorphism
The function to be invoked is known at the compile time.	The function to be invoked is known at the run time.
It is also known as overloading, early binding and static binding.	It is also known as overriding, Dynamic binding and late binding.
Overloading is a compile time polymorphism where more than one method is having the same name but with the different number of parameters or the type of the parameters.	Overriding is a run time polymorphism where more than one method is having the same name, number of parameters and the type of the parameters.
It is achieved by function overloading and operator overloading.	It is achieved by virtual functions and pointers.
It provides fast execution as it is known at the compile time.	It provides slow execution as it is known at the run time.
It is less flexible as mainly all the things execute at the compile time.	It is more flexible as all the things execute at the run time.

3.3 Operator Overloading: Concept of Overloading

If we create two or more members having the same name but different in number or type of parameter, it is known as C++ overloading. In C++, we can overload:

1. methods,
2. constructors, and
3. indexed properties

It is because these members have parameters only.

Operator overloading

Operator overloading is an important concept in C++. It is 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.

Overloading Unary Operators

The unary operators operate on a single operand and following are the examples of Unary operators −

1. The increment (++) and decrement (--) operators.
2. The unary minus (-) operator.
3. The logical not (!) operator.

The unary operators operate on the object for which they were called and normally, this operator appears on the left side of the object, as in !obj, -obj, and ++obj but sometime they can be used as postfix as well like obj++ or obj--.

Example

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

class Distance {
   private:
      int feet;             // 0 to infinite
      int inches;           // 0 to 12

   public:
      // required constructors
      Distance() {
         feet = 0;
         inches = 0;
      }
      Distance(int f, int i) {
         feet = f;
         inches = i;
      }

      // method to display distance
      void displayDistance() {
         cout << "F: " << feet << " I:" << inches <<endl;
```

```
      }

      // overloaded minus (-) operator
      Distance operator- () {
         feet = -feet;
         inches = -inches;
         return Distance(feet, inches);
      }
};

int main() {
   Distance D1(11, 10), D2(-5, 11);

   -D1;                    // apply negation
   D1.displayDistance();   // display D1

   -D2;                    // apply negation
   D2.displayDistance();   // display D2

   return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
F: -11 I:-10
F: 5 I:-11
```

Overloading Binary Operators

You can redefine or overload most of the built-in operators available in C++. Thus, a programmer can use operators with user-defined types as well. Overloaded operators are functions with special names: the keyword "operator" followed by the symbol for the operator being defined. Like any other function, an overloaded operator has a return type and a parameter list.

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;
}
```

When the above code is compiled and executed, it produces the following result −

```
Volume of Box1 : 210
Volume of Box2 : 1560
Volume of Box3 : 5400
```

Overloadable/Non-overloadable Operators

Following is the list of operators which can be overloaded −

+	-	*	/	%	^
&	|	~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	||
+=	-=	/=	%=	^=	&=
|=	*=	<<=	>>=	[]	()
->	->*	new	new []	delete	delete []

Following is the list of operators, which cannot be overloaded −

::	.*	.	?:

3.4 Data Conversion

A user-defined data types are designed by the user to suit their requirements, the compiler does not support automatic type conversions for such data types therefore, the user needs to design the conversion routines by themselves if required.

Example

```
#include <bits/stdc++.h>
using namespace std;

class Time {
    int hour;
    int mins;
```

```
public:
    // Default Constructor
    Time()
    {
        hour = 0;
        mins = 0;
    }

    // Parameterized Constructor
    Time(int t)
    {
        hour = t / 60;
        mins = t % 60;
    }

    // Function to print the value
    // of class variables
    void Display()
    {
        cout << "Time = " << hour
             << " hrs and "
             << mins << " mins\n";
    }
};

int main()
{
    // Object of Time class
    Time T1;
    int dur = 95;

    // Conversion of int type to
    // class type
    T1 = dur;
    T1.Display();

    return 0;
}
```

Output

```
Time = 1 hrs and 35 mins
```

3.5 Type casting

A type cast is basically a conversion from one type to another. There are two types of type conversion:

1. Implicit Type Conversion Also known as 'automatic type conversion'.

Example

```
// An example of implicit conversion

#include <iostream>
using namespace std;

int main()
{
    int x = 10; // integer x
    char y = 'a'; // character c

    // y implicitly converted to int. ASCII
    // value of 'a' is 97
    x = x + y;

    // x is implicitly converted to float
    float z = x + 1.0;

    cout << "x = " << x << endl
        << "y = " << y << endl
        << "z = " << z << endl;
    return 0;
}
```

Output

```
x = 107
y = a
z = 108
```

2. Explicit Type Conversion: This process is also called type casting and it is user-defined.

Example

```
// C++ program to demonstrate
// explicit type casting

#include <iostream>
using namespace std;

int main()
{
    double x = 1.2;
```

```
    // Explicit conversion from double to int
    int sum = (int)x + 1;

    cout << "Sum = " << sum;

    return 0;
}
```

Output

```
Sum = 2
```

3.6 Pitfalls of Operator Overloading

In operator overloading, any C++ existing operations can be overloaded, but some exceptions.

Which operators Cannot be overloaded?

1. Conditional [?:], size of, scope(::), Member selector(.), member pointer selector(.*) and the casting operators.
2. We can only overload the operators that exist and cannot create new operators or rename existing operators.
3. At least one of the operands in overloaded operators must be user-defined, which means we cannot overload the minus operator to work with one integer and one double. However, you could overload the minus operator to work with an integer and a mystring.
4. It is not possible to change the number of operands of an operator supports.
5. All operators keep their default precedence and associations (what they use for), which cannot be changed.
6. Only built-in operators can be overloaded.

Pitfalls of Type Casting

1. When you will use type casting, you can lose some information or data.
2. Accuracy can be lost while using type casting.

3. When a double is cast to an int, the fractional part of a double is discarded which causes the loss of fractional part of data.

3.7 Keywords explicit and mutable

Mutable data members are those members whose values can be changed in runtime even if the object is of constant type. It is just opposite to constant.

Sometimes logic required to use only one or two data members as a variable and another one as a constant to handle the data. In that situation, mutability is very helpful concept to manage classes.

Example

```
#include <iostream>
using namespace std;
code
class Test {
   public:
      int a;
   mutable int b;
   Test(int x=0, int y=0) {
      a=x;
      b=y;
   }
   void seta(int x=0) {
      a = x;
   }
   void setb(int y=0) {
      b = y;
   }
   void disp() {
      cout<<endl<<"a: "<<a<<" b: "<<b<<endl;
   }
};
int main() {
   const Test t(10,20);
   cout<<t.a<<" "<<t.b<<"\n";
   // t.a=30; //Error occurs because a can not be changed, because object is
constant.
   t.b=100; //b still can be changed, because b is mutable.
   cout<<t.a<<" "<<t.b<<"\n";
   return 0;
}
```

Explicit Keyword in C++ is used to mark constructors to not implicitly convert types in C++. It is optional for constructors that take exactly one argument and work on constructors (with a single argument) since those are the only constructors that can be used in typecasting.

Example

```
// C++ program to illustrate default
// constructor without 'explicit'
// keyword
#include <iostream>
using namespace std;

class Complex {
private:
    double real;
    double imag;

public:

    // Default constructor
    Complex(double r = 0.0,
            double i = 0.0) : real(r),
                              imag(i)
    {
    }

    // A method to compare two
    // Complex numbers
    bool operator == (Complex rhs)
    {
        return (real == rhs.real &&
                imag == rhs.imag);
    }
};

// Driver Code
int main()
{
    // a Complex object
    Complex com1(3.0, 0.0);

    if (com1 == 3.0)
        cout << "Same";
    else
        cout << "Not Same";
    return 0;
}
```

Output

```
Same
```

3.8 Function overloading

You can have multiple definitions for the same function name in the same scope. The definition of the function must differ from each other by the types and/or the number of arguments in the argument list. You cannot overload function declarations that differ only by return type.

Example

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

class printData {
   public:
      void print(int i) {
        cout << "Printing int: " << i << endl;
      }
      void print(double  f) {
        cout << "Printing float: " << f << endl;
      }
      void print(char* c) {
        cout << "Printing character: " << c << endl;
      }
};

int main(void) {
   printData pd;

   // Call print to print integer
   pd.print(5);

   // Call print to print float
   pd.print(500.263);

   // Call print to print character
   pd.print("Hello C++");

   return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
Printing int: 5
Printing float: 500.263
Printing character: Hello C++
```

3.8 Run Time Polymorphism- Pointers to Base class

One of the key features of class inheritance is that a pointer to a derived class is type-compatible with a pointer to its base class. Polymorphism is the art of taking advantage of this simple but powerful and versatile feature.

Example

```
// pointers to base class
#include <iostream>
using namespace std;

class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
};

class Rectangle: public Polygon {
  public:
    int area()
      { return width*height; }
};

class Triangle: public Polygon {
  public:
    int area()
      { return width*height/2; }
};

int main () {
  Rectangle rect;
  Triangle trgl;
  Polygon * ppoly1 = &rect;
  Polygon * ppoly2 = &trgl;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  cout << rect.area() << '\n';
  cout << trgl.area() << '\n';
```

```
  return 0;
}
```

Output

```
20
10
```

3.9 Virtual function and its significance in C++

- 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

1. Virtual functions must be members of some class.
2. Virtual functions cannot be static members.
3. They are accessed through object pointers.
4. They can be a friend of another class.
5. A virtual function must be defined in the base class, even though it is not used.
6. 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.
7. We cannot have a virtual constructor, but we can have a virtual destructor

8. Consider the situation when we don't use the virtual keyword.

Example

```
#include <iostream>
using namespace std;
class A
{
   int x=5;
    public:
    void display()
    {
        std::cout << "Value of x is : " << x<<std::endl;
    }
};
class B: public A
{
    int y = 10;
    public:
    void display()
    {
        std::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
```

3.10 Pure Virtual Function and Virtual Table (Vtable)

A pure virtual function is a virtual function in C++ for which we need not to write any function definition and only we have to declare it. It is declared by assigning 0 in the declaration.

An abstract class is a class in C++ which have at least one pure virtual function.

1. Abstract class can have normal functions and variables along with a pure virtual function.

2. Abstract class cannot be instantiated, but pointers and references of Abstract class type can be created.
3. Abstract classes are mainly used for Upcasting, so that its derived classes can use its interface.
4. If an Abstract Class has derived class, they must implement all pure virtual functions, or else they will become Abstract too.
5. We can't create object of abstract class as we reserve a slot for a pure virtual function in Vtable, but we don't put any address, so Vtable will remain incomplete.

Example

```
#include<iostream>
using namespace std;
class B {
   public:
      virtual void s() = 0; // Pure Virtual Function
};

class D:public B {
   public:
      void s() {
         cout << "Virtual Function in Derived class\n";
      }
};

int main() {
   B *b;
   D dobj;
   b = &dobj;
   b->s();
}
```

Output

```
Virtual Function in Derived class
```

3.11 Virtual destructor

Deleting a derived class object using a pointer to a base class, the base class should be defined with a virtual destructor

Example

```
#include<iostream>
using namespace std;
class b {
   public:
      b() {
         cout<<"Constructing base \n";
      }
      virtual ~b() {
         cout<<"Destructing base \n";
      }
};
class d: public b {
   public:
      d() {
         cout<<"Constructing derived \n";
      }
      ~d() {
         cout<<"Destructing derived \n";
      }
};
int main(void) {
   d *derived = new d();
   b *bptr = derived;
   delete bptr;
   return 0;
}
```

Output

```
Constructing base
Constructing derived
Destructing derived
Destructing base
```

3.12 Abstract base class

The C++ interfaces are implemented using **abstract classes** and these abstract classes should not be confused with data abstraction which is a concept of keeping implementation details separate from associated data. A class is made abstract by declaring at least one of its functions as **pure virtual** function.

Example

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

// Base class
```

```
class Shape {
   public:
      // pure virtual function providing interface framework.
      virtual int getArea() = 0;
      void setWidth(int w) {
         width = w;
      }

      void setHeight(int h) {
         height = h;
      }

   protected:
      int width;
      int height;
};

// Derived classes
class Rectangle: public Shape {
   public:
      int getArea() {
         return (width * height);
      }
};

class Triangle: public Shape {
   public:
      int getArea() {
         return (width * height)/2;
      }
};

 int main(void) {

   Rectangle Rect;
   Triangle  Tri;
   Rect.setWidth(5);
   Rect.setHeight(7);

   // Print the area of the object.
   cout << "Total Rectangle area: " << Rect.getArea() << endl;
   Tri.setWidth(5);
   Tri.setHeight(7);

   // Print the area of the object.
   cout << "Total Triangle area: " << Tri.getArea() << endl;

   return 0;
}
```

Output

```
Total Rectangle area: 35
Total Triangle area: 17
```