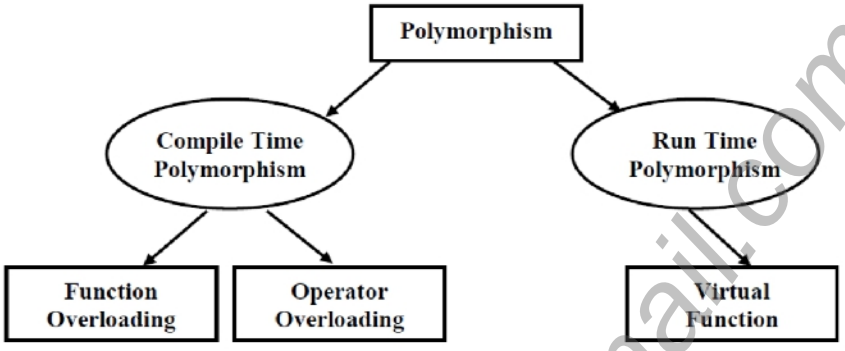
Chapter 5
Polymorphism

Polymorphism

- It is one of the important feature of OOP. Polymorphism means "one name but different forms".
- **Polymorphism** in c++ refers to an object's capacity to take several forms. Polymorphism allows us to perform the same action in multiple ways in c++.
- Polymorphism is divided into two types.
  1. **Compile time polymorphism**
  2. **Run Time Polymorphism**



Role of polymorphism

- Same method could be used for creating methods with different implementations.
- It helps programmer to reuse the existing code and class.
- Supports building extensible system.
- Polymorphism reduces the coupling between different functionalities.

Compile time Polymorphism

- Compile-time polymorphism is a polymorphism that is resolved during the compilation process. **This is also called early binding, static binding, static linking.**
- Compile Time Polymorphism is implemented by **function overloading and operator overloading.**
- In **operator overloading,** an operator exhibits different behaviors in different instances. For example, Operator symbol '+' is used for arithmetic operation between two numbers, however by overloading same operator '+' it can be used for different purpose like concatenation of strings.
- In **function Overloading**, a function call to the function are decided by the compiler in the sense that which function is going to be called is decided by the compiler at compile time. Example: [See Previous Chapter]

Runtime Polymorphism

- If appropriate member functions are chosen at **run time rather than compile time,** this is known as **runtime polymorphism, dynamic polymorphism, Late binding, or dynamic binding.** This can be achieved using **virtual function and pointer to objects of base class.**
- In C++, polymorphism indicates the form of a member function that can be changed at runtime. Such member functions are called **virtual functions** and the corresponding class is called **polymorphic class.**

Function Overloading

- C++ allows us to specify more than one definition for a function name in the same scope, which is called function overloading and these functions are said to be overloaded. So using function overloading, we can define more than one functions within a class with same name but with different signature.
- An overloaded is a declaration that is declared with the same name as a previously declared declaration in the same scope, except that both declarations have different arguments or signature and obviously different definition (implementation).
- Functions can be overloaded in two ways by varying Function signature
    i. based on the number of parameters in the function
    ii. based on the parameter data type.
- When you call an overloaded function or operator, the compiler determines the most appropriate definition to use, by comparing the argument types and arguments number we have used to call the function with the parameter types and number specified in the definitions. The process of selecting the most appropriate overloaded function or operator is called overload resolution.

Example:

- Following is the example where same function display() is being used to print different data types. Here function overloading is done based on different types of argument.

```
#include <iostream>
using namespace std;
class Demo
{
        public:
        void display(int i)
        {
                cout << "Printing int: " << i << endl;
        }

        void display(double f)
        {
                cout << "Printing float: " << f << endl;
        }

        void display(char* c)    // or void display (char [])
        {
                cout << "Printing character: " << c << endl;
        }
};
int main(void)
{
        Demo d;
        // Call print to print integer
        d.display(5);
        // Call print to print float
        d.display(500.263f);
        // Call print to print character
        d.display("Hello C++");
        return 0;
}
```

Output:

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

WAP to find the area of circle and rectangle using the concept of function / method overloading (based on number of arguments)

```
#include<iostream>
using namespace std;
class Demo
{
        public:
        double area(double r)
        {
                return (3.1416 * r * r);
        }

        double area(double l, double b)
        {
                return (l*b);
        }
};
void main()
{
        Demo d;
        double l,b,r,aor,aoc;
        cout<<"Enter length and breadth"<<endl;
        cin>>l>>b;
        aor=d.area(l,b);
        cout<<"Area of rectangle= "<<aor<<endl;
        cout<<"Enter radius of circle"<<endl;
        cin>>r;
        aoc=d.area(r);
        cout<<"Area of circle= "<<aoc;
}
```

Output:

```
Enter length and breadth
4
5
Area of rectangle= 20
Enter radius of circle
1
Area of circle= 3.1416
```


Operator Overloading

- The meaning of operators is already defined and fixed for basic types like: int, float, double etc in C++ language. For example: If you want to add two integers then, + operator is used.
- But, for user-defined types (like objects), we can define the meaning of operator, i.e., we can redefine the way that operator works. For example: If there are two objects of a class that contain string as its data member, you can use + operator to concatenate two strings. Suppose instead of strings if that class contains integer data member, then you can use + operator to add integers. This feature in C++ programming that

allows programmer to redefine the meaning of operator when they operate on class objects is known as operator overloading.

- **Operator overloading is the mechanism of giving special meanings to an operator.** It provides a flexible option for the operations of new definitions for most of the C++ operators. **Operator overloading refers to the use of an operator for different purpose in different data types.**
- In other words, operator overloading refers to giving the normal C++ operators (such as +, *, <=, += etc) additional meanings when they are applied to user-defined data types.
- Operator operating are profoundly used by programmer to make a program clearer. For example: you can replace the code like d3 = d1.add(d2); with d3 = d1 + d2; which is more readable and easier to understand.
- **Operator function is a special function which is used to define an additional task to an existing operator.**

Operator Function as Class Member

- To overload an operator, an operator function is defined inside a class as a **class member** as given below.

```
class class_name
{
........ ...... ......
public:
    return_type operator sign (argument/s)
    {
      ...... ..... ......
    }
........ ...... ......
};
```

OR

```
return_type Class_Name :: operator sign(argument/s)   // if the operator function is defined outside the class
{
}
```

- The return type comes first which is followed by keyword operator, followed by operator sign,i.e., the operator you want to overload like: +, <, ++ etc. and finally the arguments is passed.
- Then, inside the body of you want to perform the task you want when this operator function is called.
- This operator function is called when, the operator(sign) operates on the object of that class class_name.
- Operator overloading provides a flexible option for the creation of new definition for most of the C++ operators. So operator overloading is the feature of C++ realizing the **polymorphism**.

Restrictions on Operator overloading

- We can overload all C++ operator except the following:
  i. Class member access operators (. , ->)
  ii. Scope resolution operator (::)

iii. Size operator (sizeof)

iv. Condition operator (?:)

- We cannot overload the preprocessing symbols #.
- Even though the semantics of an operator can be extended, we cannot change its syntax (e.g. Number operands, precedence).

Types of Operator Overloading

There are two types of operator overloading

i. Unary operator overloading

ii. Binary operator overloading

Unary Operator Overloading

- Unary operators are those operators that act on a single operand.
- Unary plus(+), unary minus (-), Increment operator ++, Decrement operator – etc. are unary operators.
- **Overloading of unary operators without explicit arguments to an operator function is known as unary operator overloading**

WAP to demonstrates the concept of overloading unary operator ++ (prefix and postfix).

```
#include <iostream>
using namespace std;
class Check
{
        private:
                int i;
        public:
                Check()
                {
                        i=0;
                }
                void operator ++()// for prefix operator overloading
                {
                        ++i;// i=i+1
                }
                void operator ++(int)// for postfix operator overloading, here int is called dummy '
                                        argument which is used to differentiate between prefix and
                                        postfix call.
                {
                        ++i;//i=i+1
                }

                void Display()
                {
                        cout << "i=" << i << endl;
                }
};

int main()
{
        Check obj;
        obj.Display();
        ++obj;  //similar to obj.operator ++()
```

```
        obj.Display();
        obj++;
        obj.Display();
}
```

Outpu

```
i=0
i=1
i=2
```

WAP to demonstrate the concept of overloading unary operator-. The operator function must be defined outside the class.

Note: if a minus operator is used as unary operator it takes just one operand

```
#include<iostream>
using namespace std;
class Demo
{
        int x,y,z;
        public:
                void getdata(int a,int b,int c)
                {
                        x = a;
                        y = b;
                        z = c;
                }
                void display()
                {
                                cout<<x<<endl;
                                cout<<y<<endl;
                                cout<<z<<endl;
                }
                void operator -();
};

void Demo::operator-()
{
        x = -x;
        y = -y;
        z = -z;
}
int main()
{
        Demo a;
        a.getdata(1,2,-3);
        a.display();
        -a;   // similar to a.operator –();
        a.display();
}
```

Output:

```
1
2
-3
-1
-2
3
```

WAP to demonstrate overloading unary operator ++ that return a built-in type.

```
#include <iostream>
using namespace std;
class Demo
{
	private :
		int x;
	public :
		Demo (int a)
		{
			x=a;
		}
		int operator ++()
		{
			x++;
			return x;
		}
		void show()
		{
			cout<<x;
		}
};
int main()
{
	Demo obj1(10);
	int upd;
	cout<<"intial value=";
	obj1.show();
	upd=++obj1; // similar to upd= obj1.operator ++();
	cout<<endl<<"Updated value="<<upd;
}
```

Output:

```
intial value=10
Updated value=11
```

WAP to demonstrate overloading Unary operator ++ that return an object:

```
#include <iostream>
using namespace std;
class Demo
{
	private :
		int x;
	public :

		Demo()
		{
			x=0;
```

```
        }
        Demo (int p)
        {
                x=p;
        }
        void show ()
        {
                cout<<x;
        }
        Demo operator ++()
        {
                x++;
                Demo temp; // temporary object
                temp.x = x;
                return temp;
        }
};
int main()
{
        Demo obj1(20), obj2;
        cout<<"Original Value=";
        obj1.show();
        obj2=++obj1; // similar to obj2=obj1.operator(++);
        cout<<endl<<"Updated Value=";
        obj2.show();
}
```

Output

```
Original Value=20
Updated Value=21
--------------------------------
```

Binary Operator Overloading

- Binary operators are those operators that act on two operands.
- Binary plus(+), Binary minus (-), Greater than(>), Lesser than(<>), equal to (==), insertion operator(<<) and extraction operator(>>) etc. are binary operator.
- Overloading of binary Operator with a single explicit argument in operator function is known as **binary operator overloading**.

WAP to return the object of sum of two number entered by user by overloading binary operator +

```
#include<iostream>
using namespace std;
class Number
{
        public:
                int n;
                Number( )
                {
                        n=0;
                }
                Number(int num)
                {
                        n=num;
                }
```

```
        Number operator + (Number num2)
        {
                Number sum;
                sum.n=n + num2.n;
                return sum;
        }
        void display()
        {
                cout<<n;
        }
};
int main()
{
        Number n1(5),n2(10),n3;
        n3=n1+n2; //  n3 = n1.operator + (n2);
        n1.display();
        cout<<"+";
        n2.display();
        cout<<"=";
        n3.display();
}
```

```
5+10=15
```

WAP to return object of the difference between two complex number by overloading binary operator -.

```
#include <iostream>
using namespace std;
class Complex
{
        private:
        float real;
        float imag;
        public:
        Complex()
        {
                real=0;
                imag=0;
        }
        Complex(float r,float i)
        {
                real=r;
                imag=i;
        }
```

```
        // Operator overloading
        Complex operator - (Complex c2)
        {
                Complex temp;
                temp.real = real - c2.real;
                temp.imag = imag - c2.imag;
                return temp;
        }
        void output()
        {
                if(imag < 0)
                cout << "Output Complex number: "<< real << imag << "i";
                else
                cout << "Output Complex number: " << real << "+" << imag << "i";
        }
};
int main()
{
        Complex c1(2.5,-3.5), c2(1.6,4.5), result;
        // In case of operator overloading of binary operators in C++ programming,
        // the object on right hand side of operator is always assumed as argument by compiler.
        result = c1 - c2;  // this is equivalent to result=c1.operator-(c2)
        result.output();
}
```

```
Output Complex number: 0.9-8i
--------------------------------
```

Design a class Polar which describes a point in the plane using polar coordinates radius and angle. Use the operator + overloading to add two polar.

```
#include <iostream>
using namespace std;
#include <math.h>
#include <conio.h>
class Polar
{
        double radius;
        double angle;

        double getx()
        {
                return radius*cos(angle);
        }
```

```
        double gety()
        {

                return radius*sin(angle);
        }
        public:
        Polar()
        {
                radius=0.0;angle=0.0;
        }

        Polar(float r,float a)
        {
                radius=r;
                angle=a;
        }

        void display()
        {
                cout<<"("<<radius<<", "<<angle<<")";
        }
        Polar operator + (Polar o2)
        {
                Polar tmp;
                double x=getx()+o2.getx();
                double y=gety()+o2.gety();
                tmp.radius=sqrt(x*x + y*y);   //converts x and y todouble
                tmp.angle=atan(y/x);         //Polar co-ordinate.return polar(r,a);
                return tmp;
        }
};

int main()
{
        Polar o1(10,2),o2(10,5),o3;
        o3=o1+o2;
        cout<<"\no1 =";
        o1.display();
        cout<<"\no2 =";
        o2.display();
        cout<<"\no3 =";
        o3.display();
}
```

WAP to concatenate two string by overloading binary operator +.

```
        #include<iostream>
        #include<string.h>
        using namespace std;
        class String
        {
                char str[40];
                public:
                        String()
                        {
                                strcpy(str," ");
                        }
```

```
                String(char *mystr)
                {
                        strcpy(str,mystr);
                }
                void display()
                {
                        cout<<str;
                }
                String operator +(String s)
                {
                        String temp;
                        strcpy(temp.str,str);
                        strcat(temp.str,s.str);
                        return temp;
                }
};
int main()
{
        String s1("Bhesh");
        String s2("Bahadur");
        String s3;
        s3=s1+s2;  // s3= s1.operator +(s2);
        s1.display();
        cout<<" + ";
        s2.display();
        cout<<" = ";
        s3.display();
}
```

```
Bhesh + Bahadur = BheshBahadur
------------------------------
```

WAP to concatenate two string by overloading binary operator + and dynamic constructor

```
#include <iostream>
#include<string.h>
using namespace std;
class String
{
private :
        char *str;
public:
        String()
        {
                str=NULL;
        }
        String (char *b)
        {
                int len=strlen(b);
                str=new char(len+1);
                strcpy(str,b);
        }
        String operator +(String p)
        {
                String tmp;
                int len=strlen(str)+ strlen(p.str);
                tmp.str=new char(len+1);
                strcpy(tmp.str,str);
```

```
                strcat(tmp.str,p.str);
                return tmp;
        }
        void display()
        {
                cout<<str<<endl;
        }
};

int main()
{
        String str1("Bhesh"),str2("Bahadur"),str3;
        cout<<"First String="<<endl;
        str1.display();
        cout<<"Second String="<<endl;
        str2.display();
        cout<<"Concatinated String="<<endl;
        str3=str1+str2;// str3=str1.operator +(str2);
        str3.display();
}
```

```
First String=
Bhesh
Second String=
Bahadur
Concatinated String=
BheshBahadur
```

WAP to input two amount in Rupees and paisa format and find the greater one using > operator overloading.

```
#include<iostream.h>
#include<stdlib.h>
#include<conio.h>
class money
{
        int rs;
        float ps;
        public:
        money()
        {
                rs=0;
                ps=0.0;
        }
        money(int r,float p)
        {
                rs=r;
                 ps=p;
        }
```

```
        void show()
        {
                cout<<"Rs. : "<<rs<<" Ps. "<<ps;
        }
        void set()
        {
                cout<<"Enter Rs.:";cin>>rs;
                cout<<"Enter Ps.:";cin>>ps;
        }
        int operator>(money);
}; //end class
//definition of > outside the class definition
int money::operator>(money m2)
{
        float mm1=rs+ps/100;
        float mm2 = m2.rs+m2.ps/100;
        return(mm1>mm2)? true: false;
}
void main()
{
        money m1;
        m1.set();
        money m2;
        m2.set();
        cout<<"Amount 1:";
        m1.show();
        cout<<endl<<"Amount 2:";
        m2.show();
        if(m1>m2)  // similar to int x=m1.operator > (m2);
        {
                cout<<endl<<"Amount 1 is greater than amount2";
        }
        else
        {
                cout<<endl<<"Amount 1 is less than to amount2";
        }
        getch();
}
```

WAP to find the smallest number among 2 entered by user using < operator overloading.

```
#include<iostream>
#include<conio.h>
using namespace std;
class Smallest
{
	int n;
	public:
	Smallest(){}
	Smallest( int val)
	{
		n=val;
	}
	void show()
	{
		cout<<"N= "<<n;
	}
	void get()
	{
		cout<<"Enter a number "<<endl;
		cin>>n;
	}
	int operator<(Smallest s)
	{
		if(n<s.n)
		{
			return true;
		}
		else
		{
			return false;
		}
	}
};

int main()
{
	Smallest a,b,c;
	cout<<"First Number"<<endl;
	a.get();
```

```
        cout<<"Second Number"<<endl;
        b.get();
        if(a<b)   // similar to  int x=a.operator <(b);
        {
                cout<<"First number is smaller"<<endl;
                a.show();
        }
        else
        {
                cout<<"Second number is smaller"<<endl;
                b.show();
        }
}
```

WAP to input two ration (p/q) and check if they are equal or not using == operator overloading.

```
#include<iostream>
#include<conio.h>
using namespace std;
class ratio
{
        int num, den;
        public:
        ratio(){}
        ratio( int n, int d )
        {
                num =n;
                den=d;
        }
        void get()
        {
                cout<<"Nr:";cin>> num;
                cout<<"Dr:";cin>>den;
        }
        int operator==(ratio r)
        {
                return ((float)num/den==(float)r.num/r.den);
        }
};
```

```
int main ()
{
        ratio r1;
        r1.get();
        ratio r2;
        r2.get();
        if(r1==r2)
                cout<<"Two provided ratios are Equal Ratio";
        else
                cout<<"Two provided ratios are Unequal Ratio";
}
```

WAP to demonstrate the concept of overloading assignment operator =

```
#include <iostream>
using namespace std;
class Distance
{
        private:
                int feet;
                int inches;

        public:
                Distance()
                {
                        feet=0;
                        inches=0;
                }
                Distance(int f, int i)
                {
                        feet = f;
                        inches = i;
                }
                void operator = (Distance D )
                {
                        feet = D.feet;
                        inches = D.inches;
                }
                void displayDistance()
                {
                        cout << "Feet: " << feet<<endl;
                        cout <<  "Inch:" << inches << endl;
                }
};

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

        cout << "First Distance : ";
        D1.displayDistance();

        D2=D1; // D2.operator=(D1);
```

```
        cout << "Second Distance :";
        D2.displayDistance();
        return 0;
}
```

```
First Distance : Feet: 11
Inch:10
Second Distance :Feet: 11
Inch:10
```

WAP to add two number. Create new alias in for cin and out for cout to input and output by overloading insertion(<<) and extraction(>>) operator.

```
#include<iostream>
using namespace std;
class Input
{
        public:
                void operator >> (int &num)
                {
                        cin>>num;
                };
};
class Output
{
        public:
        void operator << (char *msg)
        {
                cout<<msg<<endl;
        }
        void operator << (int num)
        {
                cout<<num<<endl;
        }
};
int main()
{
        Input in;
        Output out;
        int fn, sn,sum;
        out<<"Enter First Number"; //out.operator <<("Enter First Number");
        in>>fn;//in.operator >> (fn);
        out<<"Enter Second Number";
        in>>sn;//in.operator >> (sn);
        sum=fn+sn;
        out<<"Result=";
        out<<sum;       //out.operator <<(sum);
}
```


Data type conversion (Type Casting)

- In an expression, constants and variables of different types can be mixed.
- The type of data to the right of an assignment operator is automatically converted into the type of the variable on the left.
- For example:

```
int m;
float x= 4.45667;
m=x;
```

convert x to an integer before its value is assigned to m.

- Thus, the fractional part is truncated. That is the type conversions are automatic as long as the data types involved are built-in type.
- Let's discuss what happens when they are user-defined data types.

C3 = C1 +C2;

- When the objects are of the same class type, the operations of addition and assignment are carried out smoothly and the compiler does not give any error message.
- The assignment of data items is handled by the compiler with no effort on the part of the programmer, whether they are basic or user defined if both the source and destination data items are of the same data-type.
- The variables may be of user-defined data type or basic data type.
- In case the data items are of different types, data conversion interface must be explicitly specified by the user.
- These include conversions between basic and user-defined types or between the user defined data items of different types.

1. Conversion between basic data types

- Consider the statement

```
float weight;
int age;
weight=age;
```

- The compiler calls a special routine to convert the value of age, which is represented in an integer format, to a floating-point format, so that it can be assigned to weight. The compiler has several built-in routines for the conversion of basic data types such as char to int, float to double etc. This features of the compiler, which performs conversion of data without the user intervention is known as **implicit type conversion.**
- The compiler can also be instructed explicitly to perform type conversion operation using typecast operators called as **explicit type conversion**. For example, to convert int to float, the statement is

weight = (float)age;

where the keyword float enclosed between braces is the typecast operator. This is C style of typecasting which is valid in C++ too. In C++, the above statement can also be expressed as:

weight = float(age);

2. Conversion between Objects and Basic types.

- The user cannot rely on the compiler to perform conversion from user-defined data types to basic data types and vice-versa, because the compiler does not know anything about the logical meaning of user defined data types.
- To perform meaningful conversion, the user must supply the necessary conversion function.

Conversion from basic to user-defined data types

- To convert data from a basic type to a user-defined type, the conversion function should be defined in user-defined object's class **in the form of the constructor**.
- The constructor function takes a single argument of basic data type as:

```
constructor(Basic_Type)
```

```
{
        //converting statements
}
```

WAP to input length in centimeter and convert it into meter using concept of conversion basic to user defined types.

```
#include<iostream>
using namespace std;
class Meter
{
    private:
    float m;
    public:
    Meter()
    {
        m=0;
    }
    Meter(float cm)   //  A constructor that converts the basic data type to user defined type
    {
        m=cm/100.0;
    }
    void showlength()
    {
        cout<<"Length (in meter)="<<m;
    }
};
int main()
{
    Meter m1;
    float cm;
    cout<<"Enter lengrh in centimeter";
    cin>>cm;
    //m1 is user-defined and cm is basic
    m1=cm;//convert from basic to user-defined ; // similar to m1(cm);
    m1.showlength();
}
```

Conversion user-defined to basic data types

- The conversion function should be defined in user-defined object's class in the form of the **operator function.**
- The operator function is defined as an overloaded basic data type which takes **no arguments**.

- It converts the data members of an object to basic data types and returns **a basic data-item**.
- The general form of operator function is as below.

```
operator basic_type()
{
    Conversion statements;
}
```

WAP to convert length in meter to centimeter using concept of conversion user defined type to basic data type.

```
#include<iostream>
#include<conio.h>
using namespace std;
class Meter
{
    private:
    float m;
    public:
    Meter()
    {
        m=0;
    }
    Meter (int x)
    {
        m=x;
    }
    operator float() // this is type casting operator function
    {
        float l;
        l=m*100.0;//meter to centimeter
        return (l);   // must return a type float
    }
};
int main()
{
    Meter m1(1);
    float cm1;
    //m1 is user-defined and l1 is basic
    cm1=m1;//convert from user-defined to basic; //similar to cm1= m1.operator float();
    cout<<"Length in cms="<<cm1;
}
```

Conversion between objects of different classes.

The C++ compiler does not support data conversion between objects of user-defined classes. Consider the following:

```
classA obj_a;
classB obj_b;
……
obj_a = obj_b;
```

The conversion method can be either defined in classA or classB depending on whether it should be one-**argument constructor or an operator function.**

i) **Conversion Routine in Source object: operator function**

The conversion routine in the source object's class is implemented as an operator function.

```
//destination object class
class classA
{
        //classA here……..
};

//souce object class
class classB
{
        private:
        ……..
        public:
        operator classA()//destination object class name
        {
                //code for conversion from classB to classA
        }
};
```

In an assignment statement such as,

```
obj_a=obj_b;
```

obj_b is the source object of the class classB and obj_a is the destination object of the class classA. The conversion operator classA( ) exists in the source object's class.

WAP to convert the length in CM to Meter using the concept of conversion of object between different classes. The conversion function must be defined in source object.

```
#include<iostream>
using namespace std;
class Meter
{
        float m;
        public:
                Meter ()
                {
                        m=0;
                }
                Meter (float mtr)
                {
                        m=mtr;
                }
```

```
		void display()
		{
			cout<<"Length in Meter="<<m;
		}

};
class Centimeter
{
	float cm;
	public:
		Centimeter()
		{
			cm=0;
		}
		Centimeter(float c)
		{
			cm=c;
		}
		operator Meter()
		{
				float mtr;
				mtr=cm/100;
				Meter m(mtr);
				return(m);
		}
};

int main()
{
	Centimeter c(200);
	Meter m;
	m=c;// m=c.operator Meter()
	m.display();
}
```

```
Length in Meter=2
```

WAP to convert the angle in degree to radian using the concept of conversion of object between different classes. The conversion function must be defined in source object.

Note: π rad = 180 Deg

radians = degrees × π / 180°

```
#include<iostream.>
#include<conio.h>
#define pi 3.14159
using namespace std;
class Radian
{
	float rad;
	public:
	Radian()
	{
		rad=0.0;
```

```
        }

        Radian (float r)
        {
                rad=r;
        }

        void display()
        {               cout<<"Radian ="<<rad;
        }
};
class Degree
{
        private:
        float degree;
                public:
                Degree()
                {
                        degree=0.0;
                }
                operator Radian()  // this is also called as casting operator function
                {
                        float radian;
                        radian=degree*pi/180.0;
                        return(Radian(radian)) ; // it returns the user defined Radian type
                        //return((Radian)radian);

                                        //or
                                //      Radian r(radian);
                                //      return r;
                }
                void input()
                {
                        cout<<"enter degree";
                        cin>>degree;
                }
        };
        int main()
        {
                Degree d1;
                Radian r1;
                d1.input();
                r1=d1;  //r1=d1.operator Radian();
                r1.display();
        }
```

```
enter degree180
Radian =3.14159
--------------------------------
```

ii) Conversion Routine in Destination Object: constructor function

The conversion routine can be defined in the destination object's class as a one-argument constructor.

```
//source object class
class classB
{
        //classB here……..
};

//destination object class
class classA
{
        private:
        ……..
        public:
        classA(classB obj_b)
        //destination object class name
        //object of source class
        {
                //code for conversion from classB to classA
        }
};
```

In an assignment statement such as,
obj_a=obj_b;
obj_b is the source object of the class classB and obj_a is the destination object of the class classA. The conversion constructor function classA(classB obj_b) exists in the destination object's class.'

WAP to convert the length in CM to Meter using the concept of conversion of object between different classes. The conversion function must be defined in destination object.

```
#include<iostream>
using namespace std;
class Centimeter
{
        float cm;
        public:
                Centimeter()
                {
                        cm=0;
                }
                Centimeter (float c)
                {
                        cm=c;
                }
                float getcentimeter()
                {
                        return(cm);
                }
};
class Meter
{
        float m;
        public:
                Meter ()
                {
```

```
                m=0;
        }
        Meter (Centimeter c)
        {
                m=c.getcentimeter()/100;
        }


        void display()
        {
                cout<<"Length in meter="<<m;
        }
};
int main()
{
        Centimeter c(200);
        Meter m;
        m=c;//M(C);
        m.display();

}
```

```
Length in meter=2
```

WAP to convert the angle in degree to radian using the concept of conversion of object between different classes. The conversion function must be defined in destination object.

```
#include<iostream>
#define pi 3.14159
using namespace std;
class Degree
{
        float degree;
        public:
        Degree()
        {
                degree=0.0;
        }
        float getdegree()
        {
                return degree;
        }
        void input()
        {
                cout<<"enter degree";
                cin>>degree;
        }
};
class Radian
{
        private:
        float rad;
        public:
        Radian()
        {
                rad=0.0;
```

```
        }

        Radian(Degree deg)
        {
                rad=deg.getdegree()*pi/180.0;
        }


        void display()
        {
                cout<<"Radian ="<<rad;
        }
};
int main()
{
        Degree d1;
        Radian r1;
        d1.input();
        r1=d1; //r1(d1);
        r1.display();
}
```

```
enter degree180
Radian =3.14159
--------------------------------
```

Functions/Method overriding

- The process of redefining the inherited methods of the base class in the derived class is called method overriding.
- The method in the derived class should have same signature and same return type as that of base class in order to override it.
- Since it refers to the subclass object and subclass method overrides the Parent class method, subclass method is invoked at runtime, it is also called **as Run Time Polymorphism**.

Example:

```
#include<iostream>
using namespace std;
class Base
{
        public:
        void display()
        {
                cout<<"This is hello world from base class";
        }
};
class Derived: public Base
{
        public:

                void display()
                {
```

```
                cout<<"This is hello world from derived class";
            }
    };

int main()
{
    Derived d;
    d.display();
}
```

```
This is hello world from derived class
--------------------------------
```

Overloading vs overriding

	Overloading	Overriding
Definition	Methods having same name but each must have different number of parameters or parameters having different types & order.	Sub class have method with same name and exactly the same number and type of parameters and same return type as super class method.
Meaning	More than one method shares the same name in the class but having different signature.	Method of base class is re-defined in the derived class having same signature.
Behaviour	To Add/Extend more to method's behaviour.	To Change existing behaviour of method.
Polymorphism	Compile Time	Run Time
Inheritance	Not Required	Always Required
Method Signature	Must have different signature	Must have same signature.

Pointer to base class and Virtual Function

- Polymorphism refers to the property by which objects belonging to different classes are able to respond to the same message, but in different form. For this we need to use a s**ingle pointer variable to refer to the objects of different classes.**
- We use **pointer to the base class** to refer to all derived objects.
- But the problem here is that, it **always executes the function in the base class**.

Example:

```
#include<iostream>
using namespace std;
class BaseClass
{
    public:
    void display()
    {
        cout<<"I am base class display method"<<endl;
    }
};
```

```
class DerivedClass: public BaseClass
{
	public:
		void display()
		{
			cout<<"I am derived class display"<<endl;
		}
};

int main()
{
	BaseClass bc;
	DerivedClass dc;
	BaseClass *bcptr;  //base pointer
	bcptr=&bc; //base address
	bcptr->display(); //access base class  method via base pointer
	bcptr= &dc; //derived address
	bcptr->display(); //again it access base class method via base pointer
}
```

```
I am base class display method
I am base class display method
```

- To select the appropriate member function while the program is running i.e. run time polymorphism, c++ provide **a mechanism called virtual function.**
- When same function name is used in base and derived classes, the function in base class is declared as virtual using keyword virtual preceding its normal declaration. So when a function is made virtual, then C++ determine which function to use at runtime based on the type of object pointed by the base pointer.
- Hence, we can execute different versions of virtual functions by making base pointer to point to different object to achieve **run time polymorphism or dynamic polymorphism**.

Example:

```
#include<iostream>
using namespace std;
class BaseClass
{

	public:
	virtual void display()
	{
		cout<<"I am base class display method"<<endl;
	}
};

class DerivedClass: public BaseClass
{
	public:

		void display()
		{
			cout<<"I am derived class display"<<endl;
		}
};
```

```
int main()
{
        BaseClass bc;
        DerivedClass dc;
        BaseClass *bcptr;
        bcptr=&bc;
        bcptr->display();
        bcptr= &dc;
        bcptr->display();

}
```

```
I am base class display method
I am derived class display
```

Abstract base class and pure virtual function(Deferred Method)

- A class is said to be abstract base class or simply abstract class, if we can't instantiate object of that class without creating its sub class.
- Such class only exists as parent of derived classes from which objects are instantiated.
- 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.
- If we need our method to be always overridden, then we can declare that method as a pure virtual function using **virtual keyword and without method definition**. Thus, declared methods of an abstract class must be defined in its subclass i.e., **concrete class.**
- A derive class that implements all the missing functionality is called **concrete class.** So, the concrete class provides implementations for the member functions that are not implemented in the base class.
- **Concrete class is a complete class, but abstract class is incomplete superclass.**
- **Abstract class can't be instantiated, but concrete class can be instantiated.**

Characteristics of abstract class

i. Abstract class cannot be instantiated, but pointers and references of Abstract class type can be created.

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

iii. Abstract classes are mainly used for creating interface, so that its derived classes can reuse those functionalities independently.

iv. Classes inheriting an Abstract Class must implement all pure virtual functions, or else they will become Abstract too.

- Pure virtual Functions are virtual functions with no definition. These are also called as **abstract method.** They start with virtual keyword and ends with = 0. Here is the syntax for a pure virtual function.

Syntax:

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

Example:

```
#include<iostream.h>
class Base          //Abstract base class
{
    public:
    virtual void show() = 0;          //Pure Virtual Function
};
class Derived:public Base // concrete class
{
    public:
    void show()
    {
            cout << "Implementation of Virtual Function in Derived class";
    }
};
int main()
{
    Base *b;
    Derived d;
    b = &d;
    b->show();
}
```

```
Implementation of Virtual Function in Derived class
```

Virtual function vs pure virtual function

i. A virtual function has implementation and gives the derived class the option of overriding the function while a pure virtual function doesn't provide and implementation and requires the derived class to override the function.

ii. A virtual function can be inline but pure virtual function can't be inline.

iii. A virtual function can be defined inside as well as outside the class but pure virtual function (if necessary) can be defined outside the class but not inside.

Programming example

1. **Create an abstract base class shape with two members base and height, a member function for initialization and a pure virtual function to compute area. Derive two specific classes triangle and rectangle which override the function area. Use these classes in a main function and display the area of triangle and rectangle.**

```
#include<iostream>
using namespace std;
class Shape
{
        protected:
                float base,height;
        public:

                void setdata(float a, float b)
                {
                        base=a;
                        height=b;
                }
```

```
                virtual float area()=0;
        };
        class Rectangle: public Shape
        {
                public:
                        float area()
                        {
                                return(base*height);
                        }
        };

        class Triangle: public Shape
        {
                public:
                float area()
                {
                        return(1/2.0*base*height);
                }
        };
        int main()
        {
                Rectangle r;
                Triangle t;
                r.setdata(9.0, 5.0);
                t.setdata(10,8);
                Shape *f;
                f=&r;
                cout<<"Area of rectangle= "<<f->area()<<endl;
                f=&t;
                cout<<"Area of trinagle= "<<f->area();
        }
```

```
Area of rectangle= 45
Area of trinagle= 40
--------------------------------
```

2. A class Figure consist of two fields dim1 and dim2, constructor to construct these fields and an abstract method float area() to compute area. Define two classes Rectangle and Triangle both of which consists of overridden method float area() method and constructor that invoke Figure constructor. Now write the c++ program to find the area of rectangle and triangle inside it.

```
#include<iostream>
using namespace std;
class Figure
{
        protected:
                float dim1,dim2;
        public:
                Figure(float a, float b)
                {
                        dim1=a;
                        dim2=b;
                }
                virtual float area()=0;
};
```

```
class Rectangle: public Figure
{
	public:
		Rectangle(float p, float q): Figure(p,q)
		{
		}
		float area()
		{
			return(dim1*dim2);
		}
};
class Triangle: public Figure
{
	public:
	Triangle(float p, float q): Figure(p,q)
	{
	}
	float area()
	{
		return(1/2.0*dim1*dim2);
	}
};
int main()
{
	Rectangle r(9.0,5.0);
	Triangle t(10.0,8.0);
	Figure *f;
	f=&r;
	cout<<"Area of rectangle= "<<f->area()<<endl;
	f=&t;
	cout<<"Area of trinagle= "<<f->area();
}
```

```
Area of rectangle= 45
Area of trinagle= 40
```

Compile Time Vs Runtime Polymorphism

Compile Time Polymorphism	Run time Polymorphism
In Compile time Polymorphism, the call is resolved by the compiler.	In Run time Polymorphism, the call is not resolved by the compiler.
It is also known as Static binding, Early binding and overloading as well.	It is also known as Dynamic binding, Late binding and overriding as well.
Method overloading is the compile-time polymorphism where more than one methods share the same name with different parameters or signature and different return type.	Method overriding is the runtime polymorphism having the same method with same parameters or signature but associated withcompared, different classes.
It is achieved by function overloading and operator overloading.	It is achieved by virtual functions and pointers.
It provides fast execution because the method that needs to be executed is known early at the compile time.	It provides slow execution as compare to early binding because the method that needs to be executed is known at the runtime.
Compile time polymorphism is less flexible as all things execute at compile time.	Run time polymorphism is more flexible as all things execute at run time.
Inheritance is not involved.	Inheritance is involved.

Virtual Destructors

- The virtual mechanism works only when we have a base class pointer to a derived class object.
- To implement virtual functions, C++ uses a special form of table known as the virtual table. The **virtual table** is a lookup table of functions used to resolve function calls in a dynamic/late binding manner.
- In C++, constructor cannot be virtual, because when constructor of a class is executed, there is no virtual table in the memory, means no virtual pointer defined yet. So, the constructor should always be non-virtual. But destructor can be virtual.
- We know that destructor is responsible to free the memory dynamically allocated memory.
- But, while deleting a derived class object using a pointer to a base class that has a non-virtual destructor may result undefined behavior.
- This below examples illustrates this problem.

```
#include<iostream>
using namespace std;
class BaseClass
{
    public:
    BaseClass()
    {
        cout<<"Constructing base \n";
    }
    ~BaseClass()
    {
        cout<<"Destructing base \n";
    }
};
class DerivedClass: public BaseClass
{
```

```
        public:
        DerivedClass()
        {
                cout<<"Constructing derived \n";
        }
        ~DerivedClass()
        {
                cout<<"Destructing derived \n";
        }
};
int main()
{
                DerivedClass d;
                BaseClass *b=&d;
                delete(b);

}
```

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

- So, the problem here is the destructor of derived class in not invoked as expected hence failing to free the memory space effectively under late binding method. This may result memory leak problem in long run. To correct this situation, the base class should be defined with a virtual destructor as illustrated by the below example.

```
#include<iostream>
using namespace std;
class BaseClass
{
    public:
    BaseClass()
    {
            cout<<"Constructing base \n";
    }
    virtual ~BaseClass()
    {
            cout<<"Destructing base \n";
    }
};
class DerivedClass: public BaseClass
{
    public:
    DerivedClass()
    {
            cout<<"Constructing derived \n";
    }
    ~DerivedClass()
    {
            cout<<"Destructing derived \n";
    }
};
int main()
{
```

```
    DerivedClass d;
    BaseClass *b = &d;
    delete (b);
}
```

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

- We should add a virtual destructor even if it does nothing.

Polymorphic Variable and pure polymorphism

- A polymorphic variable is a variable that can hold values of different type during the course of execution.
- A polymorphic variable derives their power using combined concept of inheritance, overriding and substitution principle. It can be defined as variable that is declared as one class and hold values from sub class.
- Pure polymorphism means a function with polymorphic variable as parameter. Different effects are formed by using different types of value.

Object Pointers

//See Previous Chapter

This Pointer

//See Previous Chapter