Operator Overloading and Data(type) conversion

The mechanism of giving special meanings to an operator is called operator overloading. The operator such as +, -, +=, >, >> etc are designed to operate only on standard data types in structured programming language such as C/C++. The operator + can be used to perform the addition operation on integer, floating point etc
.
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. We can overload all C++ operator except the following:

- Class member access operators (. , .*)
- Scope resolution operator (::)
- Size operator (**sizeof**)
- Condition operator (?:)

Even though the semantics of an operator can be extended, we cannot change its syntax. When an operator is overloaded, its original meaning is not lost. The grammatical rules defined by C++ that govern its use such as the number of operands, precedence, and associatively of the operator remain the dame for overloaded operators.

The concept of operator overloading can also be applied to data conversion. C++ offers automatic conversion of primitive data types. But the conversion of user defined data types requires some effort on the part of the programmer.
Hence we see that operator overloading concepts are applied for **extending capability of operators to operate on user defined data** and for **data conversion**.

The keyword **operator** is used for overloading the C++ operators.

```
retrun_type operator operator_symbol(argu_list)
{
     //body of function.
}
```

The keyword operator indicates that the operator symbol following it, is the C++ operator to be overloaded to operate on members of its class.

Overloading without explicit arguments to an operator function is known as **unary operator overloading** and overloading with a single explicit argument is known as **binary operator overloading**. However, with **friend functions**, unary operators take one explicit argument and binary operators take two explicit arguments.

Unary Operator Overloading

We can overload unary operator to an object in much the same way as is applied to an **int** or **float** variable. The syntax for overloading the unary operator is:

```
retrun_type operator operator_symbol()
{
     //body of function.
}
```

The following examples illustrate the overloading of unary operators:

- `Counter operator +( ); //Counter is user-defined type`

- `int operator –( );`
- `void operator ++( );`

```
//overloading unary operator
#include<iostream.h>
#include<conio.h>

class Counter
{
      private:
            unsigned int count;
      public:
            Counter() {count=0;}//constructor
            int get_count(){return count;}
      void operator ++() {count++;}//increment count
};
void main()
{

      Counter c1,c2;

      cout<<"\n c1="<<c1.get_count();
      cout<<"\n c2="<<c2.get_count();
      c1++;//unary operator on objects
      c2++;
      ++c2;
      cout<<"\n c1="<<c1.get_count();
      cout<<"\n c2="<<c2.get_count();
      // int x=5; x++; cout<<"x="<<x;
      //original meaning is not destroyed.
      getch();
}
```

By overloading we can use ++ operator for decrement and -- operator for increment giving different semantics. The process of operator overloading generally involves following steps.

1. Declare a class whose objects are to be manipulated using operators.
2. Declare the operator function, in public part of class. It can either normal member function or friend function.
3. Define operator function within the body of a class or outside the body of the class but function prototype must be inside the class body.

Invoking overloading unary operator:

`Prefix form: operator objectname;`
`Postfix form: objectname operator;`

Invoking binary operator:

`Object1 operator object2;`

Overloading Unary operator that return a value:

If we have to use overloaded operator function for return a value as:

```
Obj2=obj1++;  //returned object of obj++ is assigned to obj2
```

```
//Example: unary operator overloading  with return type.
#include <iostream.h>
class sample
{
      private :
            int x;
      public :
            sample (){x=10;} // constructor
            int getvalue (){return x;}
            sample operator ++()
            {
                  x++;
                  sample temp;   // temporary object
                  temp.x = x;
                  return temp;
            }

};
void main()
{
      sample obj1,obj2;
      cout<<endl<<"intial obj1="<<obj1.getvalue();
      cout<<endl<<"intial obj2="<<obj2.getvalue();
      obj1++; obj1++;
      obj2=obj1++;
      cout<<endl<<"Final obj1="<<obj1.getvalue();
      cout<<endl<<"final obj2="<<obj2.getvalue();
}

//output

intial obj1=10
intial obj2=10
Final obj1=13
final obj2=13
```

Nameless temporary object : A convenient way to return an object is to create a nameless temporary object in the return statement itself . In the above program, modify class definition by

```
public :
      sample() {x=10;}      //constructor without argument
      sample (int val) {x=val;} // one argument constructor
      ……..
      ……..
      sample operator++()
      {
            x=x+1; // or x++;call one argument constructor
                   //from nameless object.
      return sample(x);
```

```
}
```

Here statement return sample(x); creates a nameless object by passing an initialization value.

Note : When ++ or – is used in its overloaded role, there is no difference between pre and post – operations . i.e. object ++ and ++obj has the same role.
i.e. obj2=++obj1; and obj2=obj++; has exactly same effect

So to distinguish postfix and prefix operation,c++ provides additional syntax to express this prefix and postfix operation. The operator function.

```
      Operator++()  above  is  defined  to  indicate  prefix  and  postfix
operation as

      // prefix operation
      sample operator ++ ()
      {
      return sample(++val); // object is created with ++val
                  // i.e new valueof val and returned
      }

// postfix operation
sample operator ++ (int)
{
Return sample(val++) // object is created with val++
                // i.e old value and value is returned.
}
```

We can give increment role to -- operator and decrement role to ++ operator defining operator function as

```
// decrement role to ++
sample operator ++()
{
x--;
return sample(x);
}

// increment role to --
sample operator --()
{
x++;
return sample(x); // increments

}

//overloading unary ++ to increment time by 1 seconds
#include<iostream.h>
class time
{
      int hrs;
      int min;
      int sec;
      public:
            time(){hrs=0;min=0;sec=0;}
```

```
            time(int h,int m,int s)
            {
                  hrs=h;min=m;sec=s;
            }
            void read()
            {
                  cout<<"Enter time:hh mm ss:";
                  cin>>hrs>>min>>sec;
            }
            void show()
            {
                  cout<<hrs<<":"<<min<<":"<<sec;
            }
            //unary increment operator overloading
            void operator++()
            {
                  ++sec;
                  if(sec>=60)
                  {
                        sec=0;
                        ++min;
                  }
                  if(min>=60)
                  {
                        min=0;
                        ++hrs;
                  }
            }
      };
            void nexttime(time &t)
            {
                  cout<<"Time :"; t.show();
                  ++t;
                  cout<<"On increment time becomes: "; t.show();
                  cout<<endl;
            }

void main()
{
      time t1(3,56,59);
      time t2(4,58,59);
      nexttime(t1);
      nexttime(t2);
      ++t2;
      nexttime(t2);
      time now;
      now.read();
      nexttime(now);
}

//output
Time :3:56:59   On increment time becomes: 3:57:0
Time :4:58:59   On increment time becomes: 4:59:0
Time :4:59:1    On increment time becomes: 4:59:2
Enter time:hh mm ss:3 59 59
Time :3:59:59   On increment time becomes: 4:0:0
```

Binary operator overloading

Binary operators can be overloaded as unary operator. The syntax for overloading the biary operator is:

```
retrun_type operator operator_symbol(arg)
{
      //body of function.
}
```

The binary overloaded operator function takes the first object as an implicit operand and the second operand must be passed explicitly. The data members of the first object are accessed without using the dot operator whereas, the second argument members can be accessed using the dot operator if argument is an object, and otherwise it can be accessed directly.

The following examples illustrate the overloading of binary operators:

- `Complex operator +( complex c);`
- `int operator –(int a );`
- `void operator /(complex c );`
- `int operator *(complex c);`

```
//Addition of 2D vector without overloading
#include<iostream.h>
#include<conio.h>
class vector
{
      int x,y;
      public:
   vector(){}
      vector(int a, int b)
      {x=a; y=b; }
      void get()
      {
            cout<<"Enter x:"; cin>>x;
            cout<<"Enter y:"; cin>>y;
      }
      vector addvector(vector v2)
      {
            vector temp;
            temp.x=x+v2.x;
            temp.y=y+v2.y;
            return temp;
      }
      void put()
      {
            cout<<" (x,y) =("<<x<<","<<y<<")";
      }
}; //end class

void main()
{
      vector v1;
      cout<<"Enter V1"<<endl;
```

```
      v1.get();
      vector v2,v3;
      cout<<"Enter V2:";
      v2.get();
      v3= v1.addvector(v2);          //call to add function
      cout<<endl<<"V1=";      v1.put();   //that adds two vectors in 2D
      cout<<endl<<"V2=" ; v2.put();
      cout<<endl<<"V3="; v3.put();
   getch();
}

//Addition of 2D vector with overloading
#include<iostream.h>
#include<conio.h>
class vector
{
      int x,y;
      public:
   vector(){}
      vector(int a, int b)
      {x=a; y=b; }
      void get()
      {
            cout<<"Enter x:"; cin>>x;
            cout<<"Enter y:"; cin>>y;
      }
      vector operator+(vector v2)
      {
            vector temp;
            temp.x=x+v2.x;
            temp.y=y+v2.y;
            return temp;
      }
      void put()
      {
            cout<<" (x,y) =("<<x<<","<<y<<")";
      }
}; //end class

void main()
{
      vector v1, v2,v3;
   cout<<"Enter V1"<<endl;
   v1.get();
      cout<<"Enter V2:";
      v2.get();
      v3= v1 + v2;            //call to add function
      cout<<endl<<"V1=";      v1.put();   //that adds two vectors in 2D
      cout<<endl<<"V2=" ; v2.put();
      cout<<endl<<"V3="; v3.put();
   getch();
}



//overloading binary operators
#include<iostream.h>
```

```
#include<conio.h>

class complex
{
   private:
            float rl;
            float im;
   public:
      complex() {}
      complex(float real, float imag)
      {rl=real;im=imag;}
      void display()
      {
            cout<<"("<<rl<<"+i"<<im<<")";
      }
      complex operator+(complex c)
      {
            complex temp;
            temp.rl=rl+c.rl;
            temp.im=im+c.im;
            return temp;
      }
};
void main()
{
      complex c1,c2,c3;
      c1=complex(4.4,3.6);
      c2=complex(1.8,1.06);
      c3=c1+c2;
      c1.display();cout<<" + ";c2.display();
      cout<<" = "; c3.display();
      getch();
}
```

Concatenating strings with overloaded + operator

In C, + operator can not concatenate two strings. In C++ it is possible to use + operator to concatenate two strings using overloaded + . The original mining of + can not be altered for basic data type but we are giving additional meaning to this +.

```
//String concatenation using + operator
#include<iostream.h>
#include<string.h>
#include<conio.h>
class String
{
      private:
            char str[40];
      public:
            String()
            {
                  strcpy(str," ");
            }
            String(char *mystr)
            {
                  strcpy(str,mystr);
            }
            void display()
```

```
            {
                  cout<<str;
            }
            String operator +(String s)
            {
                  String temp;
                  temp=str;
                  strcat(temp.str,s.str);
                  return temp;
            }
};//end of class String

void main()
{
      String s1="Tribhuvan";
   String s2="University";
   String s3;
   s3=s1+s2;
   s1.display();cout<<" + ";
   s2.display(); cout<<" = ";
   s3.display();
   getch();
}
```

Overloaded relational operator

The relational operators can also be overloaded as other binary operators to extend their semantics. The following example shows the '>' operator overloading to use it for comparison of two user defined objects.

Example:

```
//overloading relational operator
#include<iostream.h>
#include<stdlib.h>
#include<conio.h>

class money
{
      private:
            int rs; float ps;
      public:
            money(){rs=0;ps=0.0;} //no argument constructor
            money(int r,float p)
            {
                  rs=r; ps=p;
            }

            void show()
            {
                  cout<<"Rs. : "<<rs<<" Ps. "<<ps;
            }
            void get()
            {
                  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.get();
            money m2;
            m2.get();
            cout<<"Amount 1:";m1.show();
            cout<<endl<<"Amount 2:";m2.show();
            if(m1>m2)
                  cout<<endl<<"Amount 1 is greater than amount2";
            else
                  cout<<endl<<"Amount 1 is less than  to  amount2";
            getch();
}
```

Overloading equality operator

```
//overloading == operator
#include<iostream.h>
#include<conio.h>
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 (num*r.den==den*r.num);
            }
};
void main ()
{
      ratio r1;
      r1.get();
      ratio r2;
      r2.get();
      if(r1==r2)
            cout<<"Equal Ratio";
      else
            cout<<"Unequal Ratio";
```

```
      getch();
}
```

Overloading with Friend Functions

Friend function plays a very important role in operator overloading by providing the flexibility denied by the member functions of a class. The only difference between a friend function and member function is that, the friend functions requires the arguments to be explicitly passed to the function and processes them explicitly, where as member function considers the first argument implicitly.

Syntax:

```
friend return_type operator Operator_symbol(args……)
{
      //body…….
}
```

Example:

```
//Addition of 2D vector with overloading
#include<iostream.h>
#include<conio.h>
class vector
{
      int x,y;
      public:
            vector(){}
            vector(int a, int b)
            {x=a; y=b; }
            void get()
            {
                  cout<<"Enter x:"; cin>>x;
                  cout<<"Enter y:"; cin>>y;
            }

            void put()
            {
                  cout<<" (x,y) =("<<x<<","<<y<<")";
            }
            friend vector operator +(vector v1, vector v2);
}; //end class

vector operator +(vector v1, vector v2)
{
            vector temp;
            temp.x=v1.x+v2.x;
            temp.y=v1.y+v2.y;
            return temp;
}


void main()
{
      vector v1, v2,v3;
```

```
        cout<<"Enter V1"<<endl;
        v1.get();
        cout<<"Enter V2:";
        v2.get();
        v3= v1 + v2;            //call to add function
        cout<<endl<<"V1=";      v1.put();   //that adds two vectors in 2D
        cout<<endl<<"V2=" ; v2.put();
        cout<<endl<<"V3="; v3.put();
        getch();
}
```

Overloading Stream [insertion (<<) and extraction(>>)] operator

One of the advantages of using the iostream objects is that we can customize them to support our own classes.
For overloading stream operator << and >> for output and input user-defined objects the syntax is

For operator <<:

```
Ostream& operator <<(ostream& objname, user-defined type &objname)
{
        //output statements with cout objects
        //return statement
}
```

For Operator >>

```
istream& operator >>(istream& objname, user-defined type &objname)
{
        //input statements with cin objects
        //return statement
}
```

Following examples show how to overload ostream's operator << and istream's operator >> so that they can get and display a user-defined object.

Example 1:

```
#include<iostream.h>
class currency
{
        int rupees;
        int paisa;

public:
        currency();
        currency(int ,int);
        friend istream& operator>>(istream&,currency&);
        friend ostream& operator<<(ostream&,currency&);
};

currency::currency() //default constructor
{
        rupees =0;
        paisa =0;
```

```
}

currency::currency(int r,int p)
{
      rupees =r;
      paisa =p;
}

istream& operator>>(istream& din,currency &m)
{
      din>>m.rupees;
      din>>m.paisa;
      return din;
}

ostream& operator<<(ostream& dout, currency &m)
{
      dout<<"Rs:"<<m.rupees;
      dout<<"Ps:"<<m.paisa<<endl;
      return dout;
}

void main()
{
      currency c1,c2;
      cout<<"Enter currency c1:";
      cin>>c1;
      cout<<"Enter currency c2:";
      cin>>c2;
      currency c3(41,52);
      cout<<"c1 is :"<<c1<<endl;
      cout<<"c2 is :"<<c2<<endl;
      cout<<"c3 is ;"<<c3<<endl;
}
```

Output of program:

```
Enter currency c1:25 60
Enter currency c2:20 95
c1 is :Rs:25Ps:60

c2 is :Rs:20Ps:95

c3 is ;Rs:41Ps:52
```

EXAMPLE 2:

```
#include<iostream.h>
#define SIZE 3
class vector
{
      int V[SIZE];
public:
      vector(); //constructor 1
      vector(int*);//constructor 2
```

```
      friend vector operator*(int,vector); //scalar *vector
      friend vector operator*(vector,int);//vector*scalar
      friend istream& operator>>(istream&,vector &); //overloading >>
      friend ostream& operator<<(ostream&,vector&); //overloading <<
}; //end of class

vector::vector() //initialize to zero
{
      for(int i=0;i<SIZE;i++)
            V[i]=0;
}

vector::vector(int *x)
{
      for(int i=0;i<SIZE;i++)
            V[i]=x[i];
}

vector operator *(int a,vector b) //friend *
{
      vector c;
      for(int i=0;i<SIZE;i++)
            c.V[i]=a*b.V[i];
      return c;
}

vector operator *(vector b,int a)
{
      vector c;
      for(int i=0;i<SIZE;i++)
            c.V[i]=b.V[i]*a;
      return c;
}

istream& operator>>(istream& vin,vector &b)
{
      for(int i=0;i<SIZE;i++)
            vin>>b.V[i];
      return vin;
}

ostream& operator <<(ostream& vout,vector &b)
{
      vout<<"("<<b.V[0];
      for(int i=1;i<SIZE;i++)
            vout<<","<<b.V[i];
      vout<<")";
      return vout;
}

void main()
{
      int X[SIZE]={2,4,6};
      vector v; //constructor 1 executed
```

```
        vector v1=X;  //constructor 2 executed
        cout<<"Enter elements of vector v:"<<endl;
        cin>>v;  //invokes operator>>()
        cout<<endl;
        cout<<"v ="<<v<<endl; //invokes operator<<()
        vector p,q;
        p=2*v; //overloaded * operator
        q=v1*2;
        cout<<endl;
        cout<<"Vector p is: "<<p<<endl;
        cout<<"Vector q is : "<<q<<endl;
}
```

Output:

```
Enter elements of vector v:
2 3 6

v =(2,3,6)

Vector p is: (4,6,12)
Vector q is : (4,8,12)
```

Rules for overloading operators

- Only existing operators can be overloaded. New operators cannot be created.
- The overloaded operator must have at least one operand that is of user-defined type.
- We cannot change the meaning of an operator. That is, we cannot redefine the plus (+) operator to subtract one value from other.
- Overloaded operators follow the syntax rules of the original operators. That cannot be overridden.
- As described above, all operators cannot be overloaded.
- Unary operators, overloaded by means of a member function take no explicit arguments and return no explicit values. But, those overloaded by means of friend function take one argument.
- Binary operators, overloaded by means of a member function take one explicit argument. But, those overloaded by means of friend function take two arguments.

Data Conversion

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. Lets 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 are 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 be instructed explicitly to perform type conversion operators known as typecast operators. 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 any thing about the logical meaning of user defined data types. To perform meaningful conversion, the user must supply the necessary conversion function.

a) **Conversion 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
}
```

```
//conversion from basic type to object
#include<iostream.h>
#include<conio.h>
```

```
class Meter
{
      private:
      float length;
   public:
      Meter()
      {
            length=0;
      }
      Meter(float l)
      {
            length=l/100.0;

      }
      void showlength()
      {
            cout<<"Length (in meter)="<<length;
      }

   };
void main()
{
      Meter m1;
   float l1;
   cout<<"\nEnter lengrh(in cms):";
   cin>>l1;
   //m1 is user-defined and l1 is basic
   m1=l1;//convert from basic to user-defined ;
   m1.showlength();
      getch();
}
```

b) **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.

```
//conversion from object to basic types
#include<iostream.h>
#include<conio.h>
class Meter
{
      private:
      float length;
   public:
      Meter()
      {
            length=0;
      }
      operator float()
      {
            float l;
         l=length*100.0;//meter to centimeter
```

```
        return (l);
      }
      void getlength()
      {
            cout<<"\nEnter length (in meters):";
        cin>>length;
      }
   };
void main()
{
      Meter m1;
      float l1;
      m1.getlength();
      //m1 is user-defined and l1 is basic
      l1=m1;//convert from user-defined to basic;
      cout<<"Length in cms="<<l1;
      getch();
}
```

c. 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.

```
//conversion from objects of different classes
//degree to radian
#include<iostream.h>
#include<conio.h>
#define pi 3.14159
class Radian
{
      private:
      float rad;
   public:
      Radian()
      {
            rad=0.0;
      }
      Radian (float r)
      {
            rad=r;
      }
      float getradian()
      {
            return rad;
      }
      void display()
      {
            cout<<"Radian ="<<getradian();
      }
};

class Degree
{
      private:
      float degree;
   public:
      Degree()
      {
            degree=0.0;
      }

      operator Radian()
      {
         float radian;
         radian=degree*pi/180.0;
                  return(Radian(radian));
      }
      void input()
      {
            cout<<"enter degree";
            cin>>degree;
      }

};

void main()
```

```
{
	Degree d1;
	Radian r1;
	//d1 and r1 are objects
	d1.input();
	r1=d1;
	r1.display();
	getch();
}
```

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:
		operator 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.

```
//conversion from objects of different classes
//constructor function
//degree to radian
#include<iostream.h>
#include<conio.h>
#define pi 3.14159

class Degree
{
	private:
	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;
      }
      float getradian()
      {
            return rad;
      }
      Radian(Degree deg)
      {
            rad=deg.getdegree()*pi/180.0;
      }

      void display()
      {
            cout<<"Radian ="<<getradian();
      }
};


void main()
{
      Degree d1;
   Radian r1;
   //d1 and r1 are objects
   d1.input();
   r1=d1;
      r1.display();
   getch();
}
```

Study this example after reading topics inheritance:

//Conversion from Degree to Radian and vice –versa.

```
//conversion from objects of different classes
//degree to radian  and vice-versa
#include<iostream.h>
#include<conio.h>
#define pi 3.14159

class Degree
{
      private:
            float degree;
      public:
            Degree()
            {
                  degree=0.0;
            }
            Degree(float dr)
            {
                  degree=dr;
            }

            float getdegree()
            {
                  return degree;
            }
            void input()
            {
                  cout<<"enter degree";
                  cin>>degree;
            }

            void display()
            {
                  cout<<"Degree:"<<degree;
            }

};
class Radian
{
      protected:
      float rad;
   public:

      Radian()
       {
            rad=0.0;
       }
            float getradian()
            {
                  return rad;
            }
            Radian(float r)
```

```
            {
                  rad=r;
            }
            Radian(Degree deg)
            {
                  rad=deg.getdegree()*pi/180.0;
            }
            void input()
            {
                  cin>>rad;
            }
            void display()
            {
                  cout<<"Radian ="<<getradian();
            }
};

 class RADIAN: public Radian
 {
      public:
            operator Degree()
            {
                  float d;
                  d=rad*180/pi;
                  return(Degree(d));
            }
            RADIAN():Radian(){}
            RADIAN(Degree degr):Radian(degr)
            {

            }

      };

void main()
{
      Degree d1;
      RADIAN r1;
      //d1 and r1 are objects
      cout<<"input degree value:";
      d1.input();
      r1=d1;
      r1.display();
      cout<<"INput RAdian Value:";
      r1.input();
      d1=r1;
      d1.display();
   getch();
}
```