OPERATOR OVERLOADING

Operator overloading

- Introduction
- Overloading unary operators, binary operators
- Overloading binary operators using friend function
- Type conversions
- Rules for overloading operators

Operator overloading:

- Operator overloading (less commonly known as ad-hoc polymorphism) is a specific case of polymorphism (part of the OO nature of the language) in which some or all operators like +, = or == are treated as polymorphic functions and as such have different behaviors depending on the types of its arguments.
- 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:
- Box operator+(const Box&);

  declares the addition operator that can be used to add two Box objects and returns final Box object. Most overloaded operators may be defined as ordinary non-member functions or as class member functions.

  In case we define above function as non-member function of a class then we would have to pass two arguments for each operand as follows
- Box operator+(const Box&, const Box&);

How to overload operators in C++ programming?

To overload an operator, a special operator function is defined inside the class as:

```
class className
{
   ... .. ...
   public
     returnType operator symbol (arguments)
     {
        ... .. ...
     }
   ... .. ...
};
```

Here, returnType is the return type of the function.

The returnType of the function is followed by operator keyword.

Symbol is the operator symbol you want to overload. Like: +, <, -, ++

You can pass arguments to the operator function in similar way as functions.

Overloadable/Non-overloadableOperators

Following is the list of operators which can be overloaded.

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

Following is the list of operators, which can not be overloaded

1. Scope Resolution Operator (::)
2. Pointer-to-member Operator (.*)
3. Member Access or Dot operator (.)
4. Ternary or Conditional Operator (?:)
5. Object size Operator (sizeof)
6. Object type Operator (typeid)

Unary Operators Overloading

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

The increment (++) and decrement (--) operators.

The unary minus (-) operator.

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--.

Following example explain how minus (-) operator can be overloaded for prefix as well as postfix usage.

```
#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.operator-();
   D1.displayDistance();  // display D1
   -D2;  // apply negation D2.operator-();
   D2.displayDistance();    // display D2
   return 0;
}
```

output:

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

Prefix ++ Increment Operator Overloading with no return type

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


class Check
{
    private:
      int i;
    public:
      Check(): i(0) {  }
      void operator ++()
        { ++i; }
      void Display()
        { cout << "i=" << i << endl; }
};
```

```
int main()
{
    Check obj;
// Displays the value of data member i for object obj
    obj.Display();


// Invokes operator function void operator ++( )
    ++obj;

// Displays the value of data member i for object obj
    obj.Display();


    return 0;
}
```

Output

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

Example 2: Prefix Increment ++ operator overloading with return type

```
#include <iostream>
using namespace std;
class Check
{
  private:
    int i;
  public:
    Check(): i(0) {  }


    // Return type is Check
    Check operator ++()
    {
       Check temp;
       ++i;
       temp.i = i;
       return temp;
    }
    void Display()
       { cout << "i = " << i << endl; }
};
int main()
{
    Check obj, obj1;
    obj.Display();
    obj1.Display();
    obj1 = ++obj;
    obj.Display();
    obj1.Display();
    return 0;
}
```

Output

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

Postfix Increment ++ Operator Overloading

Overloading of increment operator up to this point is only true if it is used in prefix form. This is the modification of above program to make this work both for prefix form and postfix form.

```
#include <iostream>
using namespace std;
class Check
{
  private:
    int i;
  public:
    Check(): i(0) {  }
    Check operator ++ ()
    {
        Check temp;
        temp.i = ++i;
        return temp;
    }
    // Notice int inside barcket which indicates postfix increment.
    Check operator ++ (int)
    {
        Check temp;
        temp.i = i++;
        return temp;
    }

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

```
int main()
{
    Check obj, obj1;
    obj.Display();
    obj1.Display();


    // Operator function is called, only then value of obj is assigned to obj1
    obj1 = ++obj;
    obj.Display();
    obj1.Display();


    // Assigns value of obj to obj1, only then operator function is called.
    obj1 = obj++;
    obj.Display();
    obj1.Display();
    return 0;
}
```

Output

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

A Binary operator(+) overaloading example:

```
#include<iostream>
using namespace std;
class Complex {
private:
    int real, imag;
public:
    Complex(int r = 0, int i =0)  {
real = r;   imag = i;
}
 // This is automatically called when '+' is used with between two Complex objects
Complex operator + (Complex const &obj)
{
        Complex res;
        res.real = real + obj.real;
        res.imag = imag + obj.imag;
        return res;
    }
void print() { cout << real << " + i" << imag << endl; }
};

int main()
{
    Complex c1(10, 5), c2(2, 4);
    Complex c3 = c1 + c2; // An example call to "operator+" c1.operator +(c2)
    c3.print();
}
```

Output:

```
12 + i9
```

Operator Overloading using Friend Function

```
#include <iostream>
using namespace std;
class Point {
  int x, y;
public:
  Point() {} // needed to construct temporaries
  Point(int px, int py) {
    x = px;
    y = py;
  }


  void show() {
    cout << x << " ";
    cout << y << "\n";
  }
  friend Point operator+(Point op1, Point op2);
// now a friend function
  Point operator=(Point op2);
};
// Now, + is overloaded using friend function.
Point operator+(Point op1, Point op2)
{
  Point temp;
  temp.x = op1.x + op2.x;
  temp.y = op1.y + op2.y;
  return temp;
}
int main()
{
  Point ob1(10, 20), ob2( 5, 30),ob3;
  ob1 = ob1 + ob2; //operator +(ob1,ob2);
  ob1.show();
  return 0;
}
```

You can also overload Relational operator like == , != , >= , <= etc. to compare two user-defined object.

```
#include <iostream>
#include <string>
using namespace std;
class Car
{
private:
    string m_make;
    string m_model;
public:
 Car(string make, string model)
  : m_make(make), m_model(model)
    {
    }
friend bool operator== (const Car &c1, const Car &c2);
 friend bool operator!= (const Car &c1, const Car &c2);
};
 bool operator== (const Car &c1, const Car &c2)
{
    return (c1.m_make== c2.m_make &&  c1.m_model== c2.m_model);
}
 bool operator!= (const Car &c1, const Car &c2)
{
    return !(c1== c2);
}
```

```
int main()
{
    Car corolla ("Toyota", "Corolla");
    Car camry ("Toyota", "Camry");
     if (corolla == camry)
  cout << "a Corolla and Camry are
the same.\n";
     if (corolla != camry )
 cout << "a Corolla and Camry are not
the same.\n";
     return 0;
}
```

output:

a Corolla and Camry are not the same.

C++ program for unary logical NOT (!) operator overloading.

```
#include<iostream>
using namespace std;
class NUM
{
   private:
      int n;
   public:
      //function to get number
      void getNum(int x)
      {
         n=x;
      }      //function to display number
      void dispNum(void)
      {
         cout << "value of n is: " << n;
      }
      //unary ! operator overloading
      void operator ! (void)
      {
         n=!n;
      }
};
int main()
{
   NUM num;
   num.getNum(10);
   cout << "Before calling Operator Overloading:";
   num.dispNum(); cout << endl;
   !num;   //overloading ! operator
   cout << "After  calling Operator Overloading:";
   num.dispNum();       return 0;  }
```

In C++, stream insertion operator “<<” is used for output and extraction operator “>>” is used for input.

We must know following things before we start overloading these operators.

1) cout is an object of ostream class and cin is an object istream class

2) These operators must be overloaded as a global function. And if we want to allow them to access private data members of class, we must make them friend.

Why these operators must be overloaded as global?

In operator overloading, if an operator is overloaded as member, then it must be a member of the object on left side of the operator. For example, consider the statement “ob1 + ob2” (let ob1 and ob2 be objects of two different classes). To make this statement compile, we must overload ‘+’ in class of ‘ob1’ or make ‘+’ a global function.

The operators ‘<<' and '>>' are called like 'cout << ob1' and 'cin >> ob1'. So if we want to make them a member method, then they must be made members of ostream and istream classes, which is not a good option most of the time. Therefore, these operators are overloaded as global functions with two parameters, cout and object of user defined class.

```
#include <iostream>
using namespace std;
 class Complex
{
private:
    int real, imag;
public:
    Complex(int r = 0, int i =0)
    {  real = r;   imag = i; }
    friend ostream & operator << (ostream &out, const Complex &c);
    friend istream & operator >> (istream &in,  Complex &c);
};
ostream & operator << (ostream &out, const Complex &c)
{
    out << c.real;
    out << "+i" << c.imag << endl;
    return out;
}
```

```
istream & operator >> (istream &in, Complex &c)
{
    cout << "Enter Real Part ";
    in >> c.real;
    cout << "Enter Imaginary Part ";
    in >> c.imag;
    return in;
}
 int main()
{
   Complex c1;
   cin >> c1;
   cout << "The complex object is ";
   cout << c1;
   return 0;
}
```

Output:

```
Enter Real Part 10
Enter Imaginary Part 20
The complex object is 10+i20
```

Copy constructor Vs. Assignment operator

Assignment operator is used to copy the values from one object to another already existing object. For example

```
time tm(3,15,45);        //tm object created and initialized
time t1;                 //t1 object created
t1 = tm;                 //initializing t1 using tm
```

Copy constructor is a special constructor that initializes a new object from an existing object.

```
time tm(3,15,45);        //tm object created and initialized
time t1(tm);             //t1 object created and initialized using tm object
```

Overloading New and Delete operator

The new and delete operators can also be overloaded like other operators in C++. New and Delete operators can be overloaded globally or they can be overloaded for specific classes.

If these operators are overloaded using member function for a class, it means that these operators are overloaded only for that specific class.

If overloading is done outside a class (i.e. it is not a member function of a class), the overloaded 'new' and 'delete' will be called anytime you make use of these operators (within classes or outside classes). This is global overloading.

Syntax for overloading the new operator :

```
void* operator new(size_t size);
```

The overloaded new operator receives size of type size_t, which specifies the number of bytes of memory to be allocated. The return type of the overloaded new must be void*.The overloaded function returns a pointer to the beginning of the block of memory allocated.

Syntax for overloading the delete operator :

```
void operator delete(void*);
```

The function receives a parameter of type void* which has to be deleted. Function should not return anything.

NOTE: Both overloaded new and delete operator functions are static members by default. Therefore, they don't have access to this pointer .

// CPP program to demonstrate Overloading new and delete operator for a specific class

```
#include<iostream>
#include<stdlib.h>
using namespace std;
class student
{
    string name;
    int age;
public:
    student()
    {
        cout<< "Constructor is called\n" ;
    }
    student(string name, int age)
    {
        this->name = name;
        this->age = age;
    }
void display()
    {
        cout<< "Name:" << name << endl;
        cout<< "Age:" << age << endl;
    }
    void * operator new(size_t size)
    {
        cout<< "Overloading new operator with size: " << size << endl;
        void * p = ::new student();
//void * p = malloc(size); will also work fine
        return p;
    }
    void operator delete(void * p)
    {
        cout<< "Overloading delete operator " << endl;
        free(p);
    } };
```

```
int main()
{
    student * p = new student("Yash", 24);
     p->display();
    delete p;
}
```

output:

```
Overloading new operator with size: 16
Constructor is called
Name:Yash
Age:24
Overloading delete operator
```

Type Conversions

In C programming language, when different types of constants and variables are used in expression, C automatically perform type conversion based on some fixed rules. In assignment operation, variable at the right hand side is automatically converted to the type of the variable on the left. The Same can also be possible in C++ programming language.

```
int a ;
float b = 3.14654;
a = b;
```

Here ‘b’ is float variable but it is at the right side in the assignment statement so converted into the integer format.

But this automated type promotion will work well if both data types are of primary data type or both are of same user-defined data type. But it will create problem when one data type is user-defined data type and another is primary data type. So for that we have to use some special function for type conversion as in such cases automatic type conversion can not be performed by the language itself.

There are three types of type conversion are possible:

- Conversion from basic type to the class type.
- Conversion from class type to basic type.
- Conversion from one class to another class type.

Basic to Class Type Conversion

The conversion from basic type to the class type can be performed by two ways:

1.Using constructor

2. Using Operator Overloading

1.Using Constructor

We can use constructor to perform type conversion during the object creation.

Consider the following example with class ‘Time’ in which we want to assign total time in minutes by integer variable ‘duration’.

To achieve that we have implemented one constructor function which accepts one argument of type integer as follow:

Explanation of the below program

Here,we have created an object “t1” of class “Time” and during the creation we have assigned integer variable “duration”. It will pass time duration to the constructor function and assign to the “hrs” and “min” members of the class “Time”.

Program to convert basic type to class type using constructor

```
#include "iostream.h"
#include "conio.h"
class Time
{
        int hrs,min;
        public:
          Time(int);
                void display();
};
Time :: Time(int t)
{
cout<<"Basic Type to ==> Class Type Conversion..."<<endl;
        hrs=t/60;
        min=t%60;
}
void Time::display()
{
        cout<<hrs<< ": Hours(s)"<<endl;
        cout<<min<< " Minutes"<<endl;
}
void main()
{
int duration;
cout<<"Enter time duration in minutes";
cin>>duration;
Time t1=duration;
t1.display();
}
```

2.Using Operator Overloading

We can also achieve type conversion by operator overloading.

We can overload assignment operator for this purpose.

Above example of Time class can be rewritten for type conversion using operator overloading concept to overload the assignment operator (=) as follow:

Program to convert from basic type to class type using operator overloading

```
#include <iostream>
using namespace std;
class Time
{
        int hrs,min;
        public:
                void display();
                void operator=(int); //
overloading function
};
void Time::display()
{
cout<<hrs<< ": Hour(s) "<<endl ;
cout<<min<<": Minutes"<<endl
;}
```

```
void Time::operator=(int t)
{
        cout<<"Basic Type to ==>
Class Type Conversion..."<<endl;
        hrs=t/60;
        min=t%60;
}
```

```
void main()
{
Time t1;
int duration;
cout<<"Enter time duration in
minutes";
cin>>duration;
cout<<"object t1 overloaded
assignment..."<<endl;
t1=duration;
t1.display();
cout<<"object t1 assignment operator
2nd method..."<<endl;
t1.operator=(duration);
t1.display();
}
```

Conversion from class type to basic type.

The syntax for the conversion function is as under:

```
operator typename( )
{
    ....
    ....
    ....
}
```

For example suppose we want to assign time in hours and minutes in the form of total time in minutes into one integer variable “duration” then we can write the type conversion function as under:

Program to demonstrate Class type to Basic type conversion

```
#include <iostream>
using namespace std;
class Time
{
        int hrs,min;
        public:
Time(int ,int);   // constructor
operator int();   // casting operator
                     function
~Time()          // destructor
{
cout<<"Destructor called..."<<endl;
}

};
```

```
Time::Time(int a,int b)
{
cout<<"Constructor called with two
parameters..."<<endl;
        hrs=a;
        min=b;
}

Time :: operator int()
{
cout<<"Class Type to Basic Type
Conversion..."<<endl;
        return(hrs*60+min);
}
```

```
int main()
{
        int h,m,duration;
        cout<<"Enter Hours ";
         cin>>h;
        cout<<"Enter Minutes ";
        cin>>m;
        Time t(h,m);       // construct object
        duration = t;      // casting conversion OR duration = (int)t
        cout<<"Total Minutes are "<<duration;
        cout<<"2nd method operator overloading "<<endl;
        duration = t.operator int();
        cout<<"Total Minutes are "<<duration;
        return 0;
}
```

Notice the statement in above program where conversion took place.

duration = t;

We can also specify the casting type and write the same statement by the following way to achieve the same result.

duration = (int) t; // Casting

The conversion function should satisfy the following condition:

- It must be a class member.
- It must not specify the return value even though it returns the value.
- It must not have any argument.

Conversion from one class to another class can be performed either by using the constructor or type conversion function.

Using Conversion Function

The following program demonstrates conversion from one class to another class type with the help of conversion function where, we have overloaded "=" operator for conversion purpose.

Program to convert one class to another class

```
#include <iostream>
using namespace std;
class inventory1
{
        int ino,qty;
        float rate;
        public:
inventory1(int n,int q,float r)
{
ino=n;
qty=q;
rate=r;
}
inventory1()
{
cout<<"\n Inventory1's Object Created"; }
int getino()
{
return(ino);
}
float getamt()
{
return(qty*rate);
}
void display()
{
cout<<endl<<"ino = "<<ino<<" qty = "<<qty<<" rate = "<<rate;
}
};
```

```
class inventory2
{
        int ino;
        float amount;
        public:

void operator=(inventory1 I)
{
ino=I.getino();
amount=I.getamt();
}
void display()
{
cout<<endl<<"ino = "<<ino<<"
amount = "<<amount;
}
};
```

```
void main()
{
        clrscr();
        inventory1 I1(1001,30,75);
        inventory2 I2;
        I2=I1;
        //inventory2 I2=I1;
        I1.display();
        I2.display();
}
```

Program to convert class Time to another class Minute.

```
#include <iostream>
using namespace std;
class Time
{
        int hrs,min;
        public:
        Time(int h,int m)
        {
                hrs=h;
                min=m;
        }
        Time()
        {
cout<<"\n Time's Object Created";
        }
int getMinutes()
{
int tot_min = ( hrs * 60 ) + min ;
return tot_min;
}
void display()
{
cout<<"Hours: "<<hrs<<endl ;
 cout<<" Minutes : "<<min <<endl ;
}
};
```

```
class Minute
{
        int min;
        public:

    Minute()
    {
       min = 0;
    }
void operator=(Time T)
{
        min=T.getMinutes();
}
void display()
{
cout<<"\n Total Minutes : "
<<min<<endl;
} };
```

```
void main()
{
Time t1(2,30);
 t1.display();
Minute m1;
 m1.display();

m1 = t1;  // conversion from Time to
Minute

t1.display();
m1.display();
}
```

Program converts from one class type to another using a conversion function and a constructor

```
#include <iostream>
using namespace std;
class Kilometers
{
private:
    double kilometers;
public:
Kilometers(double kilometers):
kilometers(kilometers) {}
    void display()
    {
cout << kilometers << "
kilometeres";
    }
double getValue()
    {    return kilometers;    }
};
```

```
 class Miles
{
private:
    double miles;
public:
    Miles(double miles) : miles(miles) {}
    void display()
    {
        cout << miles << " miles";
    }
    operator Kilometers()
    {
        return Kilometers(miles*1.609344);
    }
```

```
Miles(Kilometers kilometers)
    {
        miles = kilometers.getValue()/1.609344;
    }
};
int main(void)
{
 /*Converting using the conversion function */
    Miles m1 = 100;
    Kilometers k1;
     k1 = m1;
    m1.display();
    cout << " = ";
    k1.display();
    cout << endl;
    // Converting using the constructor
    Kilometers k2 = 100;
    Miles m2 = k2;    // same as:  Miles m2 = Miles(k2);
    k2.display();
    cout << " = ";
    m2.display();
    cout << endl;
}
```

Output

```
100 miles = 160.934 kilometeres
100 kilometeres = 62.1371 miles
```

Rules for operator overloading

Following are the general rules for operator overloading.

1) Only built-in operators can be overloaded. New operators can not be created.

2) Arity of the operators cannot be changed.

3) Precedence and associativity of the operators cannot be changed.

4) Overloaded operators cannot have default arguments except the function call operator () which can have default arguments.

5) Operators cannot be overloaded for built in types only. At least one operand must be used defined type.

6) Assignment (=), subscript ([]), function call (“()”), and member selection (->) operators must be defined as member functions

7) Except the operators specified in point 6, all other operators can be either member functions or a non member functions.

8) Some operators like (assignment)=, (address)& and comma (,) are by default overloaded.