Chapter 7 : Operator overloading & Type conversion

Reference : E. Balaguruswamy Object Oriented Programming With C++; chapter-7.

Topics:

- **Topic 1: Basic of operator overloading**
- **Topic 2: Unary operator overloading**
- **Topic 3: Binary operator overloading**
- **Topic 4: Type conversion**

Questions:

Topic 1: Basic of operator overloading

1. What is operator overloading? When an operator is overloaded,does it lose any of its original functionality?
2. Describe the operator function?
3. State the rules for operator overloading.Which operators can not be overloaded and why?
4. Differentiate between operator overloading and function overloading?

Topic 2: Unary operator overloading

5. What are the rules for unary operator overloading?
6. Write a program to explain the use of unary operator overloading.

Topic 3: Binary operator overloading

7. What are the rules for binary operator overloading?
8. Create a class FLOAT that contains one float data member.Overload all the four arithmetic operators so that they operate on the object of FLOAT.

9. Design a class polar which describes a point in the plane using polar coordinates radius and angle.Use the overloaded + operator add two objects of polar.
   [hint : you need to use the following trigonometric formula:
   $x = r * \cos(a);$
   $y = r * \sin(a);$
   $a = \tan^{-1}(y/x);$
   $r = \sqrt{(x^2 + y^2)}$

10. Create a class called COMPLEX that has two private data called real and imaginary.Include member function input() to input real & imaginary values,show() to display complex numbers.Overload + and - operator to add and subtract two complex numbers.

Topic 4: Type conversion

11. Describe type conversion.
12. We have two classes **X** and **Y**. If **a** is an object of **X** and **b** is an object of **Y** and we want to say a = b;what type of conversion routine should be used and where?
13. A friend function can not be used to overload the assignment operator(=),explain why?

Q:1:: What is operator overloading? When an operator is overloaded,does it lose any of its original functionality?

Answer:

Operator overloading: Operator overloading is a compile-time polymorphism. It is an idea of giving special meaning to an existing operator in C++ without changing its original meaning.In C++, we can make operators work for user-defined classes. This means C++ has the ability to provide the operators with a special meaning for a data type, this ability is known as operator overloading. For example,

```
int a;
float b,sum;
sum = a + b;
```

Here, variables “a” and “b” are of types “int” and “float”, which are built-in data types. Hence the addition operator ‘+’ can easily add the contents of “a” and “b”. This is because the addition operator “+” is predefined to add variables of built-in data type only.

```
class A {
    statements;
};

int main()
{
    A a1, a2, a3;

    a3 = a1 + a2;

    return 0;
}
```

In this example, we have 3 variables “a1”, “a2” and “a3” of type “class A”. Here we are trying to add two objects “a1” and “a2”, which are of user-defined type i.e. of type “class A” using the “+” operator. This is not allowed, because the addition operator “+” is predefined to operate only on built-in data types. But here, “class A” is a user-defined

type, so the compiler generates an error. This is where the concept of “Operator overloading” comes in.

Now, if the user wants to make the operator “+” add two class objects, the user has to redefine the meaning of the “+” operator such that it adds two class objects. This is done by using the concept of “Operator overloading”. So the main idea behind “Operator overloading” is to use C++ operators with class variables or class objects.

Redefining the meaning of operators really does not change their original meaning; instead, they have been given additional meaning along with their existing ones.

Q:2:: Describe the operator function?

Answer:

To define an additional task to an operator, we must specify what it means in relation to the class to which the operator is applied. This is done with the help of a special function, called operator function, which describes the task. The general form of an operator function is:

```
return type classname :: operator op(arglist)
{
        Function body              // task defined
}
```

Where,

return type = the type of value returned by the specified operation;

operator = keyword;

op = the operator being overloaded[+, - , * , / , << etc].

operator op = the function name.

Operator functions must be either member functions (non-static) or friend functions. A basic difference between them is that a friend function will have only one argument for unary operators and two for binary operators, while a member function has no arguments for unary operators- and only one for binary operators. This is because the object used to invoke the member function is passed implicitly and therefore is available for the member function. This is not the case with friend functions, arguments may be

passed either by value or by reference. For example, operator functions are declared for class name **time** using prototypes as:

```
time operator + (time); //binary addition
friend time operator + (time , time); //binary addition

time operator - (); //unary minus
friend time operator - (time); //unary minus

int operator == (time)   //binary assignment
friend int operator == (time , time)   //binary assignment
```

Q:3::State the rules for operator overloading.Which operators can not be overloaded and why?

Answer:

Rules for operator overloading

1. The process of overloading involves the following steps:
    - Create a class that defines the data type that is to be used in the overloading operation.
    - Declare the operator function operator op( ) in the public part of the class. It may be either a member function or a friend function.
    - Define the operator function to implement the required operations.
2. Only existing operators can be overloaded. New operators cannot be created.
3. The overloaded operator must have at least one operand that is of user-defined type.
4. We cannot change the basic meaning of an operator. That is to say, we cannot redefine the plus(+) operator to subtract one value from the other.
5. Overloaded Operators follow the syntax rules of the original operation . They cannot be overridden.
6. There are some operators that can not be overloaded.

```
sizeof
typeid
Scope resolution (::)
Class member access operators (.(dot), .* (pointer to member
operator))
Ternary or conditional (?:)
```

7. We cannot use friend functions to overload certain operators,but member functions can be used to overload them:

=	Assignment operator
( )	Function call operator
[ ]	Subscripting operator
->	Class member access operator

8. Unary operators, overloaded by means of a member function, take no explicit arguments and return no explicit values, but, those overloaded by means of a friend function, take one reference argument (the object of the relevant class).
9. Binary operators overloaded through a member function take one explicit argument and those which are overloaded through a friend function take two explicit arguments.
10. When using binary operators overloaded through a member function, the left hand operand must be an object of the relevant class.
11. Binary arithmetic operators such as +, - , * and / must explicitly return a value.They must not attempt to change their own arguments

Types of unary operators

Types of unary operators are mentioned below:

1. Unary minus (–)
2. Increment (++)
3. Decrement (—)
4. NOT (!)
5. Addressof operator (&)
6. sizeof()

Rest are binary operators

Q:4:: Differentiate between operator overloading and function overloading?

Answer:

	Function overloading		Operator overloading
1	Function overloading means using a single name and giving more functionality to it.	1	Operator overloading means adding extra functionality for a certain operator.
2	We can overload the function with the same name but with different parameters.	2	We can overload (define custom behavior) for operators such as '+', '-', '()', '[]'.
3	Function overloading allows us to call it in multiple ways.	3	Operator overloading allows operators to have their extending meaning beyond its predefined operational meaning.
4	When an operator is overloaded, the operator has different meanings, which depend on the type of its operands.	4	When a function is overloaded, the same function name has different interpretations depending on its signature, which is the list of argument types in the functions parameter list.

Q:5:: What are the rules for unary operator overloading?

Answer:

<u>Rules for unary operator overloading</u>

1.The process of overloading involves the following steps:

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

2. Only existing operators can be overloaded. New operators cannot be created.
3. Overloaded operator must have at least one operand that is of user-defined type.
4. We cannot change the basic meaning of an operator. That is to say, we cannot redefine the plus(+) operator to subtract one value from the other.

5. Overloaded Operators follow the syntax rules of the original operation .They cannot be overridden.

6. There are some operators that can not be overloaded.

```
sizeof
typeid
Scope resolution (::)
Class member access operators (.(dot), .* (pointer to member
operator))
Ternary or conditional (?:)
```

7. We cannot use friend functions to overload certain operators,but member functions can be used to overload them:

=	Assignment operator
( )	Function call operator
[ ]	Subscripting operator
->	Class member access operator

8. Unary operators, overloaded by means of a member function, take no explicit arguments and return no explicit values, but, those overloaded by means of a friend function, take one reference argument (the object of the relevant class).

Types of unary operators

Types of unary operators are mentioned below:

1. Unary minus (–)
2. Increment (++)
3. Decrement (—)
4. NOT (!)
5. Addressof operator (&)
6. sizeof()

Q:6::Write a program to explain the use of unary operator overloading.

Answer:

As member function:

```
#include<iostream>
using namespace std;
class code
{
        int a;
    public:
        code(int x)
        {
            a=x;
        }
        void operator- ()    //unary operator overloading
        {                     //as member function
           a = -a;
        }
        void display()
        {
            cout<<a<<endl;
        }
};
int main()
{
    code obj1(6);
    cout<<"Before unary operator overloading : ";
    obj1.display();

    -obj1;
    cout<<"After unary operator overloading : ";
    obj1.display();
    return 0;
}
```

```
Before unary operator overloading : 6
After unary operator overloading : -6
```

Code link 1

Code link 2

Code link 3

Code link 4

As friend function:

```cpp
#include<iostream>
using namespace std;
class code
{
        int a;
    public:
        code(int x)
        {
            a=x;
        }
        //declaring friend function for
        //unary operator overloading
        friend void operator- (code &);

        void display()
        {
            cout<<a<<endl;
        }
};
//defining friend function for unary operator overloading
void operator- (code &cd)
{
    cd.a = -cd.a;
}
int main()
{
    code obj1(6);
    cout<<"Before unary operator overloading : ";
    obj1.display();

    -obj1;
    cout<<"After unary operator overloading : ";
    obj1.display();
    return 0;
}
```

```
Before unary operator overloading : 6
After unary operator overloading : -6
```

Code link 1

Code link 2

Code link 3

Q:7::What are the rules for binary operator overloading?

Answer :

Rules for binary operator overloading

1. The process of overloading involves the following steps:
    a. Create a class that defines the data type that is to be used in the overloading operation.
    b. Declare the operator function operator op( ) in the public part of the class. It may be either a member function or a friend function.
    c. Define the operator function to implement the required operations.
2. Only existing operators can be overloaded. New operators cannot be created.
3. Overloaded operator must have at least one operand that is user-defined type.
4. We cannot change the basic meaning of an operator. That is to say, we cannot redefine the plus(+) operator to subtract one value from the other.
5. Overloaded Operators follow the syntax rules of the original operation . They cannot be overridden.
6. There are some operators that can not be overloaded: **sizeof()**, **typeid()**, scope resolution(**::**), class member access operators(**.** , **.***), ternary operator(**? :**) .
7. We cannot use friend functions to overload certain operators,but member functions can be used to overload them: assignment operator (**=**), function call operator **( )**, subscripting operator **[ ]**, class member access operator(**->**) .
8. Binary operators overloaded through a member function take one explicit argument and those which are overloaded through a friend function take two explicit arguments.
9. When using binary operators overloaded through a member function, the left hand operand must be an object of the relevant class.
10. Binary arithmetic operators such as +, - , * and / must explicitly return a value.They must not attempt to change their own arguments.

Q:8::Create a class FLOAT that contains one float data member.Overload all the four arithmetic operators so that they operate on the object of FLOAT.

Answer :

```cpp
#include <iostream>
using namespace std;
class FLOAT
{
    float real;
public:
    FLOAT(){};
    FLOAT (float r)
    {
        real = r;
    }
    void display()
    {
       cout<<real<<endl;
    }
    FLOAT operator+ (float a)
    {
        FLOAT  temp;
        temp.real = real + a;
        return temp;
    }
    FLOAT operator- (float a)
    {
        FLOAT  temp;
        temp.real = real - a;
        return temp;
    }
    FLOAT operator* (float a)
    {
        FLOAT  temp;
        temp.real = real * a;
        return temp;
    }
    FLOAT operator/ (float a)
    {
        FLOAT  temp;
        temp.real = real / a;
        return temp;
    }
};

int main()
{
    FLOAT c5,c4,c3,c2,c1(100.50);
    cout<<"Before operation object value: ";
    c1.display();

    cout<<"After sum operation : ";
    c2 = c1 + 10.0;
    c2.display();
    cout<<"After subtract operation : ";
    c3 = c1 - 10.0;
    c3.display();
    cout<<"After multiplication operation : ";
    c4 = c1 * 10.0;
    c4.display();
    cout<<"After division operation : ";
    c5 = c1 / 10.0;
    c5.display();

    return 0;
}
```

Output :

```
Before operation object value: 100.5
After sum operation : 110.5
After subtract operation : 90.5
After multiplication operation : 1005
After division operation : 10.05

Process returned 0 (0x0)   execution time : 0.120 s
Press any key to continue.
```

Code link :

Q:9::Design a class polar which describes a point in the plane using polar coordinates radius and angle.Use the overloaded + operator add two objects of polar.

[hint : you need to use the following trigonometric formula:

x = r * cos(a);

y = r * sin(a);

a = $\tan^{-1}(y/x)$;

r = $\sqrt{(x^2+ y^2)}$

Answer :

```cpp
#include <iostream>
#include <math.h>
#include <cmath>
using namespace std;
class polar
{
    float radius, angle ,x,y;
public:
    polar(){};
    polar (float r, float a)
    {
        radius = r;
        angle = a;
        a = a *(M_PI / 180); //converting degree to radian
        x = radius * cos(a);
        y = radius * sin(a);
    }
    void display()
    {
        cout<<"\tIn polar co-ordinate system (r,a) = ( "<<radius<<" unit , "<<angle<<
" degree )"<<endl;
        cout<<"\tIn cartesian  co-ordinate system (x,y)= ( "<<x<<" , "<<y<<" )"<<endl
;
    }
    polar operator+ (polar c2)
    {
        polar temp;
        temp.x = x + c2.x;
        temp.y = y + c2.y;
        temp.angle = atan(temp.y/temp.x);
        temp.angle = temp.angle *(180 / M_PI); //   converting radian to degree
        temp.radius = sqrt((temp.x * temp.x)+(temp.y * temp.y));
        return temp;
    }
};

int main()
{
    polar c3,c2,c1;
    float r1,r2,a1,a2;
    cout<<"Enter 1st point radius r & angle a(in degree) = ";
    cin>>r1>>a1;
    c1 = polar(r1,a1);
    cout<<"Enter 2nd point radius r & angle a(in degree) = ";
    cin>>r2>>a2;
    c2 = polar(r2,a2);
    cout<<"\n";
    cout<<"First point value : "<<endl;
    c1.display();
    cout<<"Second point value : "<<endl;
    c2.display();
    cout<<"\n";
    c3 = c1 + c2;
    cout<<"After sum operation resultant point value = "<<endl;
    c3.display();

    return 0;
}

```

Output :

```
Enter 1st point radius r & angle a(in degree) = 4 60
Enter 2nd point radius r & angle a(in degree) = 8 30

First point value :
        In polar co-ordinate system (r,a) = ( 4 unit , 60 degree )
        In cartesian  co-ordinate system (x,y)= ( 2 , 3.4641 )
Second point value :
        In polar co-ordinate system (r,a) = ( 8 unit , 30 degree )
        In cartesian  co-ordinate system (x,y)= ( 6.9282 , 4 )

After sum operation resultant point value =
        In polar co-ordinate system (r,a) = ( 11.6373 unit , 39.8961 degree )
        In cartesian  co-ordinate system (x,y)= ( 8.9282 , 7.4641 )

Process returned 0 (0x0)   execution time : 11.564 s
Press any key to continue.
```

Code link :

Q:10::Create a class called COMPLEX that has two private data called real and imaginary.Include member function input() to input real & imaginary values,show() to display complex numbers.Overload + and - operator to add and subtract two complex numbers.

Answer :

Code link :

```cpp
#include <iostream>
#include <cmath>
using namespace std;
class Complex
{
        int real, imag;
    public:
    Complex()
    {
        real = imag = 0;
    }
    Complex (int r, int i)
    {
        real = r;
        imag = i;
    }
    void display()
    {
        if(imag >= 0)
            cout<< real << " + " << imag << "i"<<endl;
        else
            cout<< real << " - " << abs(imag) << "i"<<endl;
    }
    Complex operator+ (Complex c2)
    {
        Complex temp;
        temp.real = real + c2.real;
        temp.imag = imag + c2.imag;
        return temp;
    }
    friend Complex operator- (Complex & , Complex &);
};
Complex operator- (Complex &obj1 , Complex &obj2)
{
        Complex temp;
        temp.real = obj1.real - obj2.real;
        temp.imag = obj1.imag - obj2.imag;
        return temp;
}
int main()
{
    Complex c4,c3,c2(7,5),c1(10,-15);
    c1.display();
    c2.display();

    cout<<"After sum operation : ";
    c3 = c1 + c2;
    c3.display();

    cout<<"After subtract operation : ";
    c4 = c1 - c2;
    c4.display();

    return 0;
}
```

Output:

```
10 - 15i
7 + 5i
After sum operation : 17 - 10i
After subtract operation : 3 - 20i

Process returned 0 (0x0)   execution time : 0.116 s
Press any key to continue.
```

Q:11:: Describe type conversion.

Answer :

C++ provides a mechanism to perform automatic type conversion if all variables are of basic type. For user defined data type, programmers have to convert it by using constructor or by using casting operator.

The type of data to the right of an assignment operator is automatically converted to the data type of variable on the left. Consider the following example:

```
int a , b = 5;
float x , y = 10.45;
a = y;//implicit type conversion from float to integer
x = (float)b;//explicit type conversion from integer to float
```

Here, the value of a = 10 and x = 5.0.

The type conversion is automatic as far as data types involved are built in types. We can also use the assignment operator in case of objects to copy values of all data members of the right hand object to the object on the left hand. The objects in this case are of the same data type. But if objects are of different data types we must apply conversion rules for assignment. Three types of situations arise in user defined data type conversion.

1. Basic type to Class type
2. Class type to Basic type
3. Class type to Class type

1. Basic type to Class type: To perform this conversion, the idea is to use the constructor to perform type conversion during the object creation.

2.Class type to Basic type: The constructor functions do not support conversion from a class to basic type. C++ allows us to define an overloaded casting operator that converts a class type data to basic type. The general form of an overloaded casting operator function, also referred to as a conversion function, is:

Syntax : **operator typename()**

{ //statement;

}

Now, this function converts a user-defined data type to a primitive data type. For Example, the **operator double(){ }** converts a class object to type double, the **operator int() { }** converts a class type object to type int, and so on.

3.One Class type to another Class type: Sometimes we would like to convert one class data type to another class type. Example: *Obj1 = Obj2;*

Obj1 is an object of class one and Obj2 is an object of class two. The class two type data is converted to class one type data and the converted value is assigned to the Obj1. Since the conversion takes place from class two to class one, two is known as the source class and one is known as the destination class.

Such conversion between objects of different classes can be carried out by either a constructor or a conversion function. Which form to use, depends upon where we want the type-conversion function to be located, whether in the source class or in the destination class.

Conversion	Source class	Destination class
Basic to class	Not applicable	Constructor
Class to Basic	Casting operator	Not applicable
Class to class	Casting operator	Constructor

Below is the example of basic to class type and class to basic type conversion program:

```cpp
#include<iostream>
using namespace std;
class time
{
        int hour , minutes;
    public:
        time(){};
        time(int n)
        {
            hour = n / 60;
            minutes = n % 60;
        }
        void display()
        {
            cout<<"Time = "<<hour<<" hours :: "<<minutes<<" minutes"<<endl;
        }
        operator int()  //conversion function
        {
            int h;
            cout<<"Enter extra hour = ";
            cin>>h;
            hour = hour + h; // t1.hour = t1.hour + h;
            return (hour);
        }
};
int main()
{
    time t1;
    int m , x;
    cout<<"Enter the time in minutes : ";
    cin>>m;
    t1 = m; // Basic to class type : it work as , t1 = time(m);
    t1.display();
    x = t1; // Class to basic type : it work as , x = t1.int();
    cout<<"Total hours = "<<x<<endl;
    return 0;
}
```

```
Enter the time in minutes : 234
Time = 3 hours :: 54 minutes
Enter extra hour = 4
Total hours = 7
```

Code link

Q:12::We have two classes *X* and *Y*. If *a* is an object of *X* and *b* is an object of *Y* and we want to say a = b;what type of conversion routine should be used and where?

Answer : We have to use one class to another class conversion.Such conversion between objects of different classes can be carried out by either a constructor or a conversion function. Which form to use, depends upon where we want the type-conversion function to be located, whether in the source class or in the destination class.

If we want to use a conversion function for one class to another class conversion then we have to use it in the source class.The conversion takes place in the source class and the result is given to the destination class object.

But if we want to use a constructor function for one class to another class conversion then we have to use it in the destination class.The conversion takes place in the destination class and the result is given to the source class object.

Here are example programs by using: 1. Constructor function
2. Conversion function

1. By using constructor function: Code link 1
2. By using conversion function: Code link 2

Q:13::A friend function can not be used to overload the assignment operator(=),explain why?

Answer:

First of all we need to understand the way assignment operator works, the right hand operand is the source and the left hand operand is the target of assignment, the left hand operand is typically called the LValue and the right hand operand is called RValue. Now, LValue must be a variable and can not be constant, RValue can be a variable as well as a constant.

When we overload assignment operator as member function it is the LValue object for which that assignment operator is called and the RValue object is passed as parameter

and if the RValue is a constant and we have provided a constructor to convert that constant to object then using that constructor that particular constant (which was a RValue) would be converted to object and will passed as parameter to assignment operator.

Now if we provide a constant as LValue and say we are allowed to overload assignment operator as friend function then both the operands will be passed to the friend function as parameter, now if there is a constructor that receives a constant and creates object, then compiler will use that constructor and convert that LValue (which is a constant for the given scenario) to object and will make a call to the assignment operator and we will be able to modify a constant in that way, which will be completely illogical and illegal.
Hence, to avoid modifying constant as LValue, assignment operator was restricted to be overloaded as a friend function but can be overloaded by member function.