Unit-4

Operator Overloading

Date.
Page No.

The method of making operators to work for user defined classess and having the ability to provide operators with a special user defined meaning is known as operator overloading.

OR

Operator overloading is a type of polymorphism in which an operator is overloaded to give user defined meaning to it.

For example: '+' operator can be overloaded to perform on various data types, like addition for integers, Concatenation for String etc.

In operator overloading semantics of an operator can be extended but the grammatical rules that govern its use such as the number of operands, precedence and associativity can not be changed. Remember that when an operator is overloaded, its original meaning is not lost. For example:- The operator +, which has been overloaded to add two vectors, can still be used to add two integers.

We can overload all the C++ operators except the following:

i) Class member access operators (., .*).
ii) Scope resolution operator (::).
iii) Size operator (sizeof).
iv) Conditional operator (?:).

The excluded operators are very less as compared to large number of operators which qualify for operator overloading definition.

Date.
Page No.

⊛ Defining Operator Overloading:

To define an additional task we must define that as a Operator function inside class as a class member. It must specify about task in relation to class to which the operator is being overloaded. The general form/Syntax of an operator function is:

```
return_type operator operator_symbol(argument_list)
{
    function body //task defined
}
```

Here, return_type is the type of value returned by a specified operation and operator is the keyword.

Operator functions must be either member functions 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.

⊛ Unary operator overloading:

If overloading takes place on a single operand using unary operators like increament(++), decreament(--), unary minus(-), logical not(!) etc. is called unary operator overloading.

The unary operator operate on the object for which they were called. Normally, this operator appears on the left side of object called prefix but sometimes they can be used as postfix as well.

Date.
Page No.

Example: Program to understand unary operator overloading.

```
#include<iostream>
using namespace std;
class space { int x;
              int y;
              int z;
public:
    void getdata(int a,int b,int c);
    void display(void);
    void operator-(); //unary minus overloaded.
};
void space::getdata(int a,int b,int c){
    x=a;
    y=b;
    z=c;
}
void space::display(void){
    cout<<x<<endl;
    cout<<y<<endl;
    cout<<z<<endl;
}
void space::operator-(){
    x=-x;
    y=-y;
    z=-z;
}
int main(){ space S;
    S.getdata(10,-20,30);
    cout<<"S:";
    S.display();
    -S; // activates operator-() function
```

Date. ___
Page No. ___

```
cout << "S:";
S.display();
return 0;
}
```

Output

```
S: 10
  -20
   30

S: -10
   20
   30
```

This can also be done with the help of friend function as follows.

Hint:
```
friend void operator-(space &S); //decleration
void operator-(space &S){        // takes object as argument
    S.x = -S.x;
    S.y = -S.y;
    S.z = -S.z;
}
```
//defining

Here, the argument is passed by refrence. If we pass the argument by value it well not work because only one copy of object that activated the call is passed to operator-(). So, the changes made inside the operator function will not reflect in the called function.

Date.
Page No.

Binary operator overloading:

If overloading is taking place on two operands using binary operators is called binary operator overloading. The same mechanism can be used to overload binary operator as in unary operator.

Example:- Program to understand binary operator overloading.

```
#include<iostream>
using namespace std;
class complex {
        float x; //real part
        float y; //imaginary part
public:
        complex (){} //constructor 1
Constructor 2 <-//  complex (float real, float imag){
                    x=real;
                    y=imag;
                }
        complex operator+ (complex);
        void display(void);
};
complex complex::operator+ (complex c){
        complex temp; //temporary
        temp.x = x+c.x;
        temp.y = y+c.y;
}
void complex::display(void)
{
        cout<<x<<"+j"<<y<<"\n";
}
```

return data type — class name

Date.
Page No.

```
int main(){
        complex C1,C2,C3; //invokes constructor 1
        C1=complex(2.5,3.5); //invokes constructor 2.
        C2=complex(1.6,2.7);
        C3=C1+C2; //invokes operator+() function
        cout<<"C1="; C1.display();
        cout<<"C2="; C2.display();
        cout<<"C3="; C3.display();
        return 0;
}
```

Output: (real part) (imaginary part)

C1 = 2.5 + i3.5
C2 = 1.6 + i2.7
C3 = 4.1 + i6.2

~~for~~ From complex complex::operator+(complex c) in the program we conclude the ~~following~~ following features of this function.

i) It receives only one complex type argument explicitly.
ii) It returns a complex type value.
iii) It is a member function of complex.

Date.
Page No.

⊛ Overloading Binary Operators using friend function:

Friend functions may be used in the place of member functions for overloading a binary operator, the only difference being that a friend function requires two arguments to be explicitly passed to it, while a member function requires only one.

The program for complex number discussed in previous section can be modified using a friend operator function as follows:

i) Replace the member function declaration by the friend function declaration.

```
friend complex operator+(complex, complex);
```

ii) Redefine the operator function as follows:

```
complex operator+(complex a, complex b).
{
    return complex ((a.x+b.x),(a.y+b.y));
}
```

In this case, the statement C3=C1+C2; is equivalent to C3=operator+(C1,C2);

Note: uncompatible types are type casting.

Date.
Page No.

⊛ Type Conversion:

The process of converting one data type into another is called type conversion or type casting. It is discussed already in detail in chapter 2. We are going to discuss here only about the situations that might arise in the data conversion between uncompatible types which are as follows:

⇒ Conversion from basic type to class type:

The conversion from basic type to class type (i.e user defined data type) can be illustrated will the following example.

```
#include<iostream.h>
#include<conio.h>
using namespace std;
class time { int year;
             int month;
             public;
             time(int y, int m){
                 year=y;
                 month=m;
             }
             time(float a){ year=int(a)
                 month=12*(a-year);
             }
void display(){ cout<<"year="<<year<<"\n";
                cout<<"month="<<month;
             }
};
```

use of parameterized constructor

Date.
Page No.

```
int main(){
        float y;
        cout << "Enter the year";
        cin >>y;
        time t =y;
        t.display();
}
```

(object)
(class type data)
(i.e, time float a)

Conversion from class to basic type:

In this type of conversion the source type is class type and the destination type is basic type. We used parameterized constructor for ~~conv~~ converting basic type to class type in previous section but constructor functions does not support this ~~over~~ type of operation. C++ allows us to define an overloaded casting operator that could be used to convert a class type data to a basic type. The general form of overloaded casting operator function is:

```
operator type_name(){
...
... (function statements)
...
}
```

This function converts a class type data to typename. For example, the operator int() converts a class type object to type int. The casting operator function should satisfy following conditions:

1. It must be a class member
2. It must not specify a return type.
3. It must not have any arguments.

Date.
Page No.

Example

```
#include<iostream>
using namespace std;
class time{
    int hrs;
    int mins;
public:
    time(int h, int m){
        hrs=h;
        mins=m;
    }
void display(){
    cout<<hrs<<"hours";
    cout<<mins<<"minutes";
}
operator int(){
    return hrs*60+mins;
}
};
int main(){
    int duration;
    time t(3,20);
    t.display();
    duration=t;
    cout<<duration;
    return 0;
}
```

Date:
Page No.

iii) Conversion from one class type to another class type:

One class type can be converted to another class type using conversion function source class and using constructor function in destination class. In case of conversion between objects constructor function is applied to destination class. A conversion function is applied to source file.

Example:

```
#include<iostream>
using namespace std;
class rupee{ public:
    int rs;
    void show(){ cout<<"money in rupees"<<rs; }
};
class dollar{ public:
    int doll;
    dollar(int x){ doll=x;
    }
    operator rupee(){ rupee temp;
        temp.rs=doll*50;
        return temp;
    }
    void show(){
        cout<<"Money in dollars="<<doll;
    }
};
```

Date.
Page No.

```
int main {
    dollar d1(5);
    d1.show();
    rupee r1=d1;
    r1.show();
    return 0;
    }
```

Summary

Conversion required	Conversion take place in	
	Source class	Destination class
Basic → class	Not applicable	Constructor
class → Basic	Casting operator	Not applicable
class → class	Casting operator	Constructor

There are mainly two data types in programming built-in (Basic) data type and class type (user-defined) data type. If the conversion takes place within the basic data types then this type of conversion is called implicit conversion or automatic conversion which is automatically done by the compiler. If the conversion takes place as uncompatible types which three we discussed then this type of conversion is called Explicit conversion.

Note :- While practicing programs on binary operator overloading practice these three types.

i) Arithmetic operators overloading (Already done).
ii) Comparision operator overloading
iii) Assignment operator overloading