

UNIT 4

OPERATOR OVERLOADING

LH – 7HRS

PRESENTED BY:

ER. SHARAT MAHARJAN

OOP IN C++

CONTENTS (LH – 7HRS)

- 4.1 Fundamental of operator overloading, Restriction on operator overloading, Operator functions as a class members,
- 4.2 Overview of unary and binary operator,
- 4.3 Prefix and postfix unary operator overloading,
- 4.4 Overloading binary operator (arithmetic operators overloading, comparison operator overloading, assignment operator overloading),
- 4.5 Data Conversion (basic to basic, basic to user-defined, user-defined to basic, user-defined to user-defined: routine in source object, routine in destination object)

4.1 Fundamental of Operator Overloading

- C++ provides a special function to change the current functionality of some operators within its class which is often called as operator overloading. Operator Overloading is the method by which we can change the function of some specific operators to do some different task.
- This can be done by declaring the function, its syntax is,

```
return-type operator op(arguments){  
    //Function body  
}
```
- In the above syntax **return_type** is value type to be returned to another object, **operator op** is the function where the **operator** is a keyword and **op** is the operator to be overloaded. Operator function must be either **non-static (member function)** or **friend function**.

Need of Operator Overloading:

- It is a **type of polymorphism** in which an operator is overloaded to give user defined meaning to it. Overloaded operator is **used to perform operation on user-defined data type**.
- Operator overloading allows C++ operators **to have user-defined meanings on user-defined types or classes**.
- For example **'+' operator** can be **overloaded to perform addition on various data types, like for Integer, String(concatenation) etc.**

Below are some criteria/rules to define the operator function:

- In **case of a non-static function**, the binary operator should have only one argument and unary should not have an argument.
- In the **case of a friend function**, the binary operator should have only two argument and unary should have only one argument.
- All the class member object **should be public if operator overloading is implemented**.
- Operators that **cannot be overloaded** are **dot operators(.), scope resolution operator(::), size operator(sizeof) and conditional operator(?:)**.
- Operator cannot be used to overload when declaring that function as friend function assignment operator(=), function call operator(()), subscripting operator([]) and arrow operator(->).

4.2 & 4.3 Overview of unary(prefix+postfix) and binary operator

Operator Overloading can be done by using **three approaches**, they are

1. **Overloading unary operator.**
2. **Overloading binary operator.**
3. **Overloading binary operator using a friend function.**

1. Overloading Unary Operator: Let us consider to **overload (++)** unary operator. In unary operator function, **no arguments should be passed**. It **works** only with **one class objects**. It is a **overloading of an operator operating on a single operand**.

a. Overloading Prefix Operator:

LAB 1:

```
#include<iostream>
#include<conio.h>
using namespace std;
class rectangle{
    private:
    public: int length, breadth;

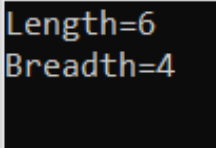
    rectangle(int l, int b){
        length=l;
        breadth=b;
    }

    void operator ++(){
        ++length;
        ++breadth;
    }

    void getData(){
        cout<<"Length="<<length<<endl;
        cout<<"Breadth="<<breadth<<endl;
    }
};

int main(){
    rectangle r1(5,3);
    ++r1;
    r1.getData();
    getch();
    return 1;}
```

OUTPUT:



```
Length=6
Breadth=4
```

b. Overloading Postfix Operator:

To overload the postfix operator we need to use int as dummy argument to the operator function. That int isn't really an argument and it doesn't mean integer. It is simply a signal to the compiler to create the postfix version of the increment and decrement operator. For example:

```
#include<iostream>          LAB 2:
```

```
#include<conio.h>
```

```
using namespace std;
```

```
class rectangle{
```

```
    private:
```

```
        int length,breadth;
```

```
    public:
```

```
        rectangle(int l, int b){
```

```
            length=l;
```

```
            breadth=b;
```

```
        }
```

```
        void operator ++(int){
```

```
            length++;
```

```
            breadth++;
```

```
        }
```

```
        void getData(){
```

```
            cout<<"Length="<<length<<endl;
```

```
            cout<<"Breadth="<<breadth<<endl;
```

```
        }
```

```
};
```

```
int main(){
```

```
    rectangle r1(5,3);
```

```
    r1++;
```

```
    r1.getData();
```

```
    getch();
```

```
    return 1;}
```

OUTPUT:

```
Length=6  
Breadth=4
```

2. Overloading Binary Operators:

Binary Operators are operators, which require two operands to perform the operation. When they are overloaded by **means of member function**, the function takes one argument whereas it takes two arguments in case of operator overloading by using friend function.

a. Overloading Plus (Binary) Operator: LAB 3:

```
#include<iostream>
#include<conio.h>
using namespace std;
class distances{
private:
    int feet, inch;
public:
    distances(){
    }
    distances(int f, int i){
        feet=f;
        inch=i;
    }
    distances operator +(distances d5){
        distances d4;
        d4.feet=feet+d5.feet;
        d4.inch=inch+d5.inch;
        d4.feet=d4.feet+d4.inch/12;
        d4.inch=d4.inch%12;
        return d4;
    }
    void getData(){
        cout<<"Feets="<<feet<<endl<<"Inches="<<inch<<endl;
    }
};

int main(){
    distances d1(2,7),d2(8,6),d3;
    d3=d1+d2;
    d3.getData();
    getch();
    return 0;
}
```



OUTPUT:

```
Feets=11
Inches=1
```

3. Overloading binary operator using a friend function:

LAB 4:

Pardaina

```
#include<iostream>
#include<conio.h>
using namespace std;
class distances{
    private:
        int feet, inch;
    public:
        distances(){
        }
        distances(int f, int i){
            feet=f;
            inch=i;
        }
        friend distances operator +(distances, distances);
        void getData(){
            cout<<"Feets="<<feet<<endl<<"Inches="<<inch<<endl;
        }
};

distances operator +(distances x, distances y){
    distances d;
    d.feet=x.feet+y.feet;
    d.inch=x.inch+y.inch;
    d.feet=d.feet+d.inch/12;
    d.inch=d.inch%12;
    return d;
}

int main(){
    distances d1(2,7),d2(8,6),d3;
    d3=d1+d2;
    d3.getData();
    getch();
    return 0;}
```

OUTPUT:

```
Feets=11
Inches=1
```

4.4 Overloading binary operator (arithmetic, comparison and assignment)

1. Arithmetic operators overloading: Already done in slides 8,9 and 10.

2. Comparison operator overloading:

- **Overloading comparison operator is almost similar to overloading plus (+) operator except that it must return value of an integer type. This is because result of comparison is always true or false. C++ treats true as non-zero value and false as zero.(0=false & 1=true)**
- We can overload < (less than) operator to compare two objects as follows:

LAB 5:

```
#include<iostream>
#include<conio.h>
using namespace std;
class time{
private:
    int hrs, mins;
public:
    time(int h, int m){
        hrs=h;
        mins=m;
    }
    int operator <(time t){
        int ft,st;    //first time and second time
        ft=hrs*60+mins;
        st=t.hrs*60+t.mins;
        if(ft<st){
            return 1;
        }
        else{
            return 0;
        }
    }
};

int main(){
    time t1(3,55),t2(4,20);
    if(t1<t2)
        cout<<"t1 is less than t2."<<endl;
    else
        cout<<"t2 is less than t1."<<endl;
    getch();
    return 0;
}
```

Handwritten annotations in blue ink: A bracket connects the `mins` parameter in the constructor to the `mins` member variable. Another bracket connects the `mins` member variable to the `mins` field in the `operator <` function. The numbers `4` and `3` are written next to the `mins` fields in the `operator <` function, indicating the values being compared.

OUTPUT:

```
t1 is less than t2.
```

3. Assignment Operator Overloading:

We can overload assignment (=) operator in C++. By overloading assignment operator, all values of one object can be copied to another object by using assignment operator. If the overloading function for the assignment operator is not written in the class, the compiler generates the function to overload the assignment operator.

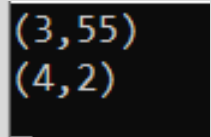
For example:

LAB 6:

```
#include<iostream>
#include<conio.h>
using namespace std;
class time{
    private:
        int hrs, mins;
    public:
        time(int h, int m){
            hrs=h;
            mins=m;
        }
        void operator =(time &t){
            hrs=t.hrs;
            mins=t.mins;
        }
        void getData(){
            cout<<"("<<hrs<<","<<mins<<")"<<endl;
        }
};

int main(){
    time t1(3,55),t2(4,2);
    t1.getData();
    t1=t2;
    t1.getData();
    getch();
    return 0;
}
```

OUTPUT:



```
(3,55)
(4,2)
```

4.5 Data Conversion padh

It is the process of **converting one type of data into another (int->float)**. A type conversion may either be **explicit (type casting)** or **implicit (automatic)**. Four types of situations might arise in the data conversion in this case:

1. Conversion **from basic type to another basic type(00)**
2. Conversion **from basic type to user defined type(01)**
3. Conversion **from user defined type to basic type(10)**
4. Conversion **from one user defined type to another user defined type(11)**

LAB 7: Conversion from basic type to another basic type:

Example:

```
#include<iostream>
#include<conio.h>
using namespace std;
class numbers{
    private:
        int a;
    public:
        numbers(float x){
            a=int(x);
        }
        void getData(){
            cout<<"Value of integer="<<a<<endl;
        }
};
int main(){
    numbers n(3.8);
    n.getData();
    getch();
    return 0;
}
```

OUTPUT:

Value of integer=3

2. Conversion from basic type to user defined type(distances(3.5)):

To convert basic types to user defined type (object) it is necessary to use the constructor. The constructor in this case takes single argument whose type is to be converted. For example:

```
#include<iostream>
```

LAB 8:

```
#include<conio.h>
```

```
using namespace std;
```

```
class distances{
```

```
    private:
```

```
        int feet,inch;
```

```
    public:
```

```
        distances(int f,int i){
```

```
            feet=f;
```

```
            inch=i;
```

```
        }
```

```
        distances(float x){
```

```
            feet=int(x);
```

```
            inch=12*(x-feet);
```

```
        }
```

```
        void getData(){
```

```
            cout<<"Feets="<<feet<<endl<<"Inches="<<inch<<endl;
```

```
        }
```

```
};
```

```
int main(){
```

```
    distances d(2.5);
```

```
    d.getData();
```

```
    getch();
```

```
    return 0;}
```

OUTPUT:

```
Feets=2
Inches=6
```

3. Conversion from user defined type to basic type(float(d)):

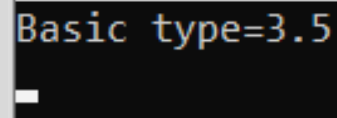
To convert user defined types (objects) to basic type it is necessary to overload cast operator. Overloaded cast operator doesn't have return type. Its implicit return type is the type to which object need to be converted. To convert object to basic type, we use conversion function as below:

```
#include<iostream>
#include<conio.h>
using namespace std;
class distances{
private:
    int feet,inch;
public:
    distances(int f,int i){
        feet=f;
        inch=i;
    }
    operator float(){ //typecast overloading
        float b=feet+inch/12.0;
        return b;
    }
};

int main(){
    distances d(3,6);
    float a=float(d);
    cout<<"Basic type="<<a<<endl;
    getch();
    return 0;
}
```

LAB 9:

OUTPUT:



```
Basic type=3.5
_
```

4. Conversion from user defined type to another user defined type:

This type of conversion can be carried out **either by a constructor or an operator function**. It depends upon where we want the **routine(code for conversion) to be located** – in the source class or in the destination class.

a. Routine in the Source class:

If we want to **convert objects of one class to object of another class**, it is necessary that the **operator function** be placed in the source class.

For example:

```

#include<iostream>
#include<conio.h>
using namespace std;
class distances{
private:
    int feet,inch;
public:
    distances(){
    }
    distances(int f,int i){
        feet=f;
        inch=i;
    }
    void getData(){
        cout<<"Feets="<<feet<<endl<<"Inches="<<inch<<endl;
    }
};

class dist{
private:
    int meter, centimeter;
public:
    dist(int m,int c){
        meter=m;
        centimeter=c;
    }
    operator distances(){ //routine in source class
        int f,i;
        f=meter*3.3;
        i=centimeter*0.4;
        f=f+i/12;
        i=i%12;
        return distances(f,i); //nameless temporary objects
    }
};

int main(){
    distances d1;
    dist d2(4,50);
    d1=distances(d2);
    d1.getData();
    getch();
    return 0;}

```

OUTPUT:

```

Feets=14
Inches=8

```

b. Routine in the Destination Class:

In this case, it is necessary that the **constructor** be placed in the destination class. This constructor is a single argument constructor and serves as an instruction for converting the argument's type to the class type of which it is a member. For example:

LAB 11:

OUTPUT:

```
Feets=14
Inches=8
```

```
1  #include<iostream>
2  #include<conio.h>
3  using namespace std;
4  class dist{
5      private:
6          int meter, centimeter;
7      public:
8          dist(int m,int c){
9              meter=m;
10             centimeter=c;
11         }
12         int getMeter(){
13             return meter;
14         }
15         int getCentimeter(){
16             return centimeter;
17         }
18     };
19     class distances{
20     private:
21         int feet, inch;
22     public:
23         distances(){
24         }
25         distances(int f,int i){
26             feet=f;
27             inch=i;
28         }
29         distances(dist d){
30             int m,c;
31             m=d.getMeter();
32             c=d.getCentimeter();
33             feet=m*3.3;
34             inch=c*0.4;
35             feet=feet+inch/12;
36             inch=inch%12;
37         }
38         void getData(){
39             cout<<"Feets="<<feet<<endl<<"Inches="<<inch<<endl;
40         }
41     };
42     int main(){
43         distances d1;
44         dist d2(4,50);
45         d1=d2;
46         d1.getData();
47         getch();
48         return 0;
49     }
```

THANK YOU FOR YOUR ATTENTION