Object Oriented Programming in C++

Segment-4

Course Code: 2301

Prepared by Sumaiya Deen Muhammad

Lecturer, CSE, IIUC

Contents

Operator Overloading:

- Binary operator overloading
- Unary operator overloading
- Relational and logical operator overloading
- Operator overloading using friend functions
- Limitations of operator overloading.

Operator Overloading

Operator overloading is one of the most exciting features of object-oriented programming. It is an important technique that has enhanced the power of extensibility of C++.it provides a flexible option for the creation of new definitions for most of the C++ operators. operator overloading gives you the opportunity to redefine the C++ language. The meaning of an operator is always same for variable of basic types like: int, float, double etc. For example: To add two integers, + operator is used.

However, for user-defined types (like: objects), you can redefine the way operator works. For example:

If there are two objects of a class that contains string as its data members. You can redefine the meaning of + operator and use it to concatenate those strings. This feature in C++ programming that allows programmer to redefine the meaning of an operator (when they operate on class objects) is known as operator overloading.

Why is operator overloading used?

We can write any C++ program without the knowledge of operator overloading. However, operator operating are profoundly used by programmers to make program intuitive. For example,

We can replace the code like:

```
calculation = add(multiply(a, b),divide(a, b));
```

to

```
calculation = (a*b)+(a/b);
```

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 we want to overload. Like: +, <, -, ++
- we can pass arguments to the operator function in similar way as functions.

Unary Operator 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.

Following example explain how unary minus (-) operator can be overloaded.

Unary Operator Overloading (contd.)

```
class Distance {
   private:
      int feet;
      int inches;
   public:
     Distance() {
         feet = 0;
         inches = 0;
      }
      Distance(int f, int i) {
         feet = f;
         inches = i;
      }
```

```
void displayDistance() {
     cout << "F: " << feet << " I:" << inches;
     }
     // overloaded minus (-) operator
     Distance operator- () {
        feet = -feet;
        inches = -inches;
        return Distance(feet, inches);
     }
};
```

```
  int main() {
  Distance D1(11, 10), D2(-5, 11);
  D1.displayDistance();
  -D1;              // apply negation
  D1.displayDistance();
  -D2;              // apply negation
  D2.displayDistance();
  return 0;
}
```

Unary Operator Overloading(contd.)

```
class Test
{
   private:
      int count;
   public:
       Test(): count(5){}

       void operator ++()
       {
          count = count+1;
       }
       void Display() { cout<<"Count: "<<count; }
};
```

```
int main()
{
    Test t;
    // this calls "function void operator ++()" function
    ++t;
    t.Display();
    return 0;
}
```

Binary Operator Overloading

Binary operators can be overloaded just as easily as unary operators. It operates on two operands. We'll look at examples that overload arithmetic operators, comparison operators, and arithmetic assignment operators. Binary operator takes two arguments.

For example: +,-,/ etc. opearotor opeartoes on two operands. These are binary operators.

The following example shows Addition operator (+) overloading:

Binary Operator Overloading (contd.)

Some common binary operators in computing include:

Relational Operators:

Equal (==)

Not equal (!=)

Less than (<)

Greater than (>)

Greater than or equal to (>=)

Less than or equal to (<=)

Logical Operator:

Logical AND (&&)

Logical OR (| |)

Arithmetic Operator:

Plus (+)

Minus (-)

Multiplication (*)

Divide (/)

(+) Operator Overloading

```
class Weight
{ private: int kilo;
    public:
    Weight()
    {     kilo = 5;   }

    Weight(int  k)
    {      kilo = k;   }
```

```
Weight operator + (const Weight& obj)
    {
        int total;
        total = kilo + obj.kilo;
        cout<<"Total: "<<total<<endl;
    }
}; int main()
{
    Weight w1,w2(20);
    Weight w3;
    w3 = w1 + w2;
    return 0;
}
```

Example-2

```
class Distance
{
private:
int feet, inches;
public:
Distance() : feet(0), inches(0.0)
{ }
Distance(int ft, float in) : feet(ft), inches(in)
{ }
void getdist()
{
   cout<<"Enter Feet:";
   cin>>feet;
   cout<<"Enter inch:";
   cin>>inches;
}
```

```
void showdist()
{
  cout << "Feet:"<< feet << " and Inch:" << inches << endl;
}

Distance operator + (const Distance& d2)
{
  int f  = feet + d2.feet;            //add the feet
  int i = inches + d2.inches;       //add the inches
  return Distance(f, i);             //initialized to sum
}
};
```

Example-2 (contd.)

```
int main()
{
Distance dist1, dist3, dist4;       //define distances
dist1.getdist();  //get dist1 from user
Distance dist2(6, 7); //define, initialize dist2
dist3 = dist1 + dist2;         //single ‘+’ operator
dist4 = dist1 + dist2 + dist3; //multiple ‘+’ operators
```

```
//display all lengths
cout << "dist1 = "; dist1.showdist(); cout << endl;
cout << "dist2 = "; dist2.showdist(); cout << endl;
cout << "dist3 = "; dist3.showdist(); cout << endl;
cout << "dist4 = "; dist4.showdist(); cout << endl;
return 0;
  }
```

Example-2 (contd.)

Output:

```
Enter feet: 4
Enter inches: 5
dist1 = feet:4 and Inch:5 ← from user
dist2 = feet:6 and Inch:7 ← initialized in program
dist3 = feet:10 and Inch:12 ← dist1+dist2
dist4 = feet:20 and Inch:24 ← dist1+dist2+dist3
```

Assignment Operator (=) Overloading

```
class Distance {
   private:
      int feet;
      int inches;
   public:
Distance() {
         feet = 0;
         inches = 0;
      }
      Distance(int f, int i) {
         feet = f;
         inches = i;
      }
```

```
void operator = (const Distance &D ) {
         feet = D.feet;
         inches = D.inches;
      }

      // method to display distance
      void displayDistance() {
         cout << "F: " << feet
          cout <<  " I:" <<  inches << endl;
      }
};
```

```
int main() {
   Distance D1(11, 10), D2(509, 119);

   cout << "First Distance : ";
   D1.displayDistance();
   cout << "Second Distance :";
   D2.displayDistance();

   // use assignment operator
   D1  = D2;
   cout << "First Distance :";
   D1.displayDistance();

   return  0;
}
```

(/) Operator Overloading

```
#include<iostream>
using namespace std;
class Weight
{ private: int kilo;
    public:
    Weight()
    {
        kilo = 25;
    }
    Weight(int k)
    {
        kilo = k;
    }
Weight operator / (const Weight& obj)
    {
        int total;
        total = kilo / obj.kilo;
        cout<<"Total: "<<total<<endl;
    }
};
int main()
{
    Weight w1,w2(5);
    Weight w3;
    w3 = w1  / w2;
    return 0;
}
```

Relational Operator Overloading

```
class Distance {
  private:
    int feet;
    int inches;
  public:
    Distance() {
      feet = 0;
      inches = 0;
    }
    Distance(int f, int i) {
      feet = f;
      inches = i;
    }
    void displayDistance() {
      cout << "F: " << feet << " I:" << inches <<endl;
    }
```

```
    // overloaded < operator
    bool operator <(const Distance& d) {
      if(feet < d.feet && inches < d.inches) {
        return true;
      }
      else  return false;
    }
};
int main() {
  Distance D1(10, 11), D2(20, 21);
  if( D1 < D2 ) {
    cout << "D1 is less than D2 " << endl;
  } else {
    cout << "D1 is not less than D2 " << endl;
  }
  return 0;
}
```

Relational and logical operator overloading

Follow the examples that we have practiced in Lab class.

Operator overloading using friend functions

- Operator overloading using Friend function offers better flexibility to the class.
- These functions are not a members of the class and they do not have 'this' pointer.
- When you overload a unary operator you have to pass one argument.
- When you overload a binary operator you have to pass two arguments.
- Friend function can access private members of a class directly.

Operator overloading using friend functions (contd.)

- **Syntax:**

```
friend return-type operator operator-symbol (Variable 1, Varibale2)
{
    //Statements;
}
```

Operator overloading using friend functions Example

```
class UnaryFriend
{   int a=10;
    int b=20;
    int c=30;
    public:
        void getvalues()
        {
            cout<<"Values of  A, B & C\n";
            cout<<a<<"\n"<<b<<"\n"<<c<<"\n"<<endl;
        }
        void show()
        {   cout<<a<<"\n"<<b<<"\n"<<c<<"\n"<<endl;    }

        void friend operator-(UnaryFriend &x);
};
```

```
void operator-(UnaryFriend &x)
{
    x.a = -x.a;
    x.b = -x.b;
    x.c = -x.c;
}
int main()
{   UnaryFriend x1;
    x1.getvalues();
    cout<<"Before Overloading\n";
    x1.show();
    cout<<"After Overloading \n";
    -x1; //unary minus operator overloaded
     x1.show();
     return 0;
}
```

Operator overloading using friend functions Example (contd.)

In the above program, **operator –** is overloaded using friend function. The **operator()** function is defined as a **Friend function.** The statement **-x1** invokes the **operator()** function. The object **x1** is created of class **UnaryFriend.** The object itself acts as a source and destination object. This can be accomplished by sending reference of an object. The object **x1** is a reference of object **x.** The values of object **x1** are replaced by itself by applying negation.

Limitations of Operator Overloading

- Self study