



Operator Overloading

Chapter 4

Object Oriented Programming

By DSBaral





Introduction

- Operator overloading is a mechanism where we define how the operator can be used to operate on objects or user defined types
- The operators in C++ such as +, -, /, * can operate on basic types but they cannot operate on user defined types without writing additional piece of code.
- The mechanism of adding special meaning to an operator to operate on objects or user defined types is called operator overloading.
- Operator overloading provides an option for the extension of the meaning of the operator when applied to user defined data types.

OOP, Operator Overloading by DSBaral


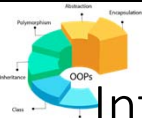
2



Introduction (Contd...)

- To add complex numbers using member functions we could have made a function to use it in following ways.
`c3.add(c1, c2);`
or
`c3=c1.add(c2);`
- Now, if we overload + operator to add two complex numbers, above statement can be replaced by a more mathematical expression
`c3=c1+c2;`
- Operator overloading allows the user to extend the language and makes the code more readable.



OOP, Operator Overloading by DSBaral 3



Introduction (Contd...)

- All C++ operators are overloadable except the following:
 - member access operators (.)
 - pointer to member access (.*)
 - scope resolution operator (::)
 - condition operator (?:)
 - size operator (sizeof)
 - run-time type information operator (typeid)

OOP, Operator Overloading by DSBaral 4





Introduction (Contd...)

- Points to remember
 - When an operator is overloaded, its original syntactic rules such as the number of operands, precedence, and associativity of the operator remain the same.
 - For example, a unary operator cannot be used as binary and vice versa.
 - It is not possible to define new operator tokens, but we can use the function call mechanism when the available operators are not adequate.
 - For example, we cannot make a new operator ** for power function.
 - While overloading operators we should not change the basic meaning of the operator.
 - For example, we should not overload + to perform multiplication.
 - The concept of operator overloading can also be applied to data conversion similar to the conversion of basic data types by adding the required code.

OOP, Operator Overloading by DSBaral

5





Syntax of Operator Overloading

- The operator overloading is done by making a function of the operator with the keyword `operator` and the operator symbol.
- The operator function is in the following form:

```
return_type operator operator_symbol(parameter_list)
{
    //body of function
}
```
- The operator overloading can be done by a member operator function or a non member operator function.
- For a member operator function, the function must be non static.

OOP, Operator Overloading by DSBaral

6






Syntax of Operator Overloading (Contd...)

- The operator function when declared as member function has the following syntax.

```
class class_name
{
    //.....
public:
    return_type operator operator_symbol([arg]);
    //.....
};
return_type class_name::operator operator_symbol([arg])
{
    //body of function
}
```

OOP, Operator Overloading by DSBaral 7






Syntax of Operator Overloading (Contd...)

- The operator function when declared as non member friend function has the following syntax.

```
class class_name
{
    //.....
public:
    friend return_type operator operator_symbol(arg[,arg]);
    //.....
};
return_type operator operator_symbol(arg[,arg])
{
    //body of function
}
```



OOP, Operator Overloading by DSBaral 8

Syntax of Operator Overloading (Contd...)

- The non member operator function takes one more argument than the member operator function.
 - For unary operator overloading, member operator function has no arguments and non member operator function has one argument.
 - For binary operator overloading, member operator function has one argument and non member operator function has two arguments.

OOP, Operator Overloading by DSBaral 9





Unary Operator Overloading

- The operators which operate on a single operand (data) are called unary operators.
- The unary operators in C++ are either used as prefix or postfix with the operand.
- The unary operator defined as a member function has the following form


```
class class_name
{
    //.....
public:
    return_type operator operator_symbol();//prefix
    return_type operator operator_symbol(int);//postfix
    //.....
};
```

OOP, Operator Overloading by DSBaral 10




Unary Operator Overloading (Contd...)

```
//for prefix
return_type class_name::operator operator_symbol(){
    //.....
}
//for postfix
return_type class_name::operator operator_symbol(int){
    //.....
}
```

← dummy argument

- If # is a prefix unary operator then the operation #var is interpreted as var.operator#().
- Similarly, if # is a postfix unary operator then the operator var# is interpreted as var.operator#(int).

OOP, Operator Overloading by DSBaral 11



Unary Operator Overloading (Contd...)



- The unary operator defined as a non member function has the following forms

```
retrun_type operator operator_symbol(obj){ //prefix
    //.....
}
retrun_type operator operator_symbol(obj,int){ //postfix
    //.....
}
```

← dummy argument

- If # is the prefix unary operator then the operation #var is interpreted as operator#(var).
- Similarly, if # is the postfix operator then the operation var# is interpreted as operator#(var,int).

OOP, Operator Overloading by DSBaral 12




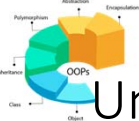
Unary Operator Overloading (Contd...)

- The following program shows the overloading of increment (++) operator

```
class counter {  
private:  
    unsigned count;  
public:  
    counter() {count=0;} //constructor  
    counter(unsigned n):count(n){}  
    int val(){return count;}  
    void operator ++();  
};  
void counter::operator ++(){  
    count=count + 1;  
}
```

OOP, Operator Overloading by DSBaral

13





Unary Operator Overloading (Contd...)

```
int main()  
{  
    counter c1,c2(5);  
    cout<<"\n counter 1="<<c1.val();  
    cout<<"\n counter 2="<<c2.val();  
    //c1++; //Error, no postfix overloading  
    ++c1;  
    ++c2;  
    cout<<"\n counter 1="<<c1.val();  
    cout<<"\n counter 2="<<c2.val();  
    return 0;  
}
```

OOP, Operator Overloading by DSBaral

14




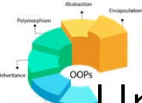
Unary Operator Overloading (Contd...)

- The above program generates error for postfix operation, so we need to overload the postfix operator with dummy argument of type `int` as

```
class counter {  
    //...  
public:  
    //...  
    void operator ++(int); //postfix ++  
};  
void counter::operator ++(int){  
    count=count + 1;  
}
```

OOP, Operator Overloading by DSBaral

15



Unary Operator Overloading (Contd...)

- Now the following operation does not generate the error

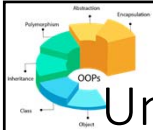
```
int main(){  
    counter c1,c2;  
    c1++; //postfix operator called  
    ++c2; //prefix operator called  
    //...  
}
```

- The operator functions in above example do not return value so

```
counter c3;  
c3=++c1; //generates error
```

OOP, Operator Overloading by DSBaral

16



Unary Operator Overloading (Contd...)

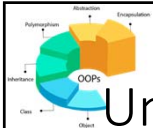
- This problem can be solved by making the operator function to return the value of counter type as

```
class counter {
    //...
public:
    //...
    counter operator ++(); //prefix ++
    counter operator ++(int); //postfix ++
};
counter counter::operator ++(){
    count=count+1;
    return counter(count); //or return *this;
}
```

- Similarly overloading can be defined for postfix

OOP, Operator Overloading by DSBaral

17



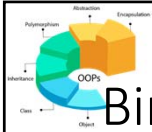
Unary Operator Overloading (Contd...)

- Similarly to overload increment operator with non member function the overloading can be done as

```
class counter {
    //...
    friend counter operator ++(counter &c); //prefix ++
    friend counter operator ++(counter &c,int); //postfix ++
};
counter operator ++(counter &c){
    c.count++; return c;
}
counter operator ++(counter &c,int){
    c.count++; return c;
}
```

OOP, Operator Overloading by DSBaral

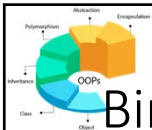
18



Binary Operator Overloading

- The operators which operate on two operands (data) are called binary operators.
- The binary operator defined as a member function has the following form

```
class class_name {
    //.....
public:
    return_type operator operator_symbol(class_name arg);
    //.....
};
return_type class_name::operator
operator_symbol(class_name arg)
{
    //.....
}
```

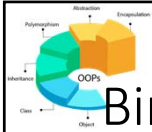


Binary Operator Overloading (Contd...)

- If # is a binary operator, then the operator `var1#var2` is interpreted as `var1.operator#(var2)`.
- The binary operator defined as a non member function has the following form

```
return_type operator operator_symbol (class_name obj1,
class_name obj2)
{
    //.....
}
```

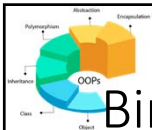
- If # is the binary operator then the operation `var1#var2` is interpreted as `operator#(var1, var2)`.



Binary Operator Overloading (Contd...)

- Following program shows the overloading of binary operator plus(+).

```
class complex
{
private:
    float real,imag;
public:
    complex(){}
    complex(float r1, float im):real(r1),imag(im){}
    void display(){cout<<"("<<real<<","<<imag<<")";}
    complex operator+(complex cn);
};
```





Binary Operator Overloading (Contd...)

```
complex complex::operator+(complex cn)
{
    complex temp;
    temp.real=real+cn.real;
    temp.imag=imag+cn.imag;
    return temp;
}

int main()
{
    complex c1(4.4,3.6),c2(1.8,1.06),c3;
    c3=c1+c2; //overloaded operator function called
    c3.display();
    //...
}
```

*float rreal = real + cn.real;
float rimag = imag + cn.imag;
return complex(rreal, rimag);*





Binary Operator Overloading (Contd...)

- When the plus operator (+) is overloaded as non member function

```
class complex {  
    //...  
public:  
    //...  
    friend complex operator+(complex cn1, complex cn2);  
};  
complex operator+(complex cn1, complex cn2){  
    complex temp;  
    temp.real=cn1.real+cn2.real;  
    temp.imag=cn1.imag+cn2.imag;  
    return temp;  
}
```

Handwritten notes:
 $float \> real = cn1.real + cn2.real;$
 $float \> imag = cn1.imag + cn2.imag;$
 $return complex(real, imag);$



OOP, Operator Overloading by DSBaral 23



Assignment

- Try to overload following operators
 - Index operator
 - New/delete operator
 - Assignment operator
 - Function call operator
 - Other overloadable operators



OOP, Operator Overloading by DSBaral 24

Data Conversion

- Data conversion or type conversion is one important aspect in programming.
- While writing programs we need to work with different data types and there can be situations where operation among different data is to be performed.
- The binary operators in C++ cannot operate if both the operands are of different types.
 - They are to be converted to same type before operation
- For basic types data conversion is done automatically or the programmer can convert explicitly (type cast).
- For basic types conversion routine is internally implemented but the conversion of user defined data (objects) is to be defined by the programmer.



OOP, Operator Overloading by DSBaral 25

Data Conversion (Contd...)

- Data conversion is done in following ways
 - Between basic types
 - Automatic (implicit, using promotion rules)
 - Type cast (explicit, by the programmer)
 - Basic to user defined and vice versa
 - User defined to user defined
- Among these conversions basic to basic conversion methods are already discussed.
- Here we will be discussing second and third conversion methods

OOP, Operator Overloading by DSBaral 26

Conversion from Basic to User-Defined Type

- The one parameter constructor of following type


```
class counter {
public:
    counter(int n);
};
```

 can be invoked as


```
counter c1=5;
```



 This call is equivalent to


```
counter c1=counter(5);
```

 Or we can also assign integer to object of counter as


```
counter c2;
c2=6; //c2=counter(6);
```
- In both of these cases the integer data is converted to counter type before initialization or assignment.

OOP, Operator Overloading by DSBaral 27






Conversion from Basic to User-Defined Type (Contd...)

- Now we can conclude that to convert data from basic to user defined type we can implement constructor in the destination class that takes argument of basic type.
- The conversion function from basic to user defined type has the following form


```
class class_name {
private:
    //.....
public:
    //.....
    class_name(basic_data_type) {
        //conversion statements
    }
};
```

OOP, Operator Overloading by DSBaral 28

Conversion from Basic to User-Defined Type (Contd...)

- The following example illustrates the conversion from basic to user defined type.

```


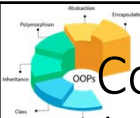
class engdist
{
private:
    int feet;
    float inch;
public:
    engdist(){}
    engdist(float in){
        feet=static_cast<int>(in/12);
        inch=in-feet*12;
    }
}
  
```

Handwritten notes illustrating conversion:

- `engdist` ↔ `float`
- `feet` ↔ `inches`
- `2 ft` ↔ `25.4 in`
- `1.4 in` (under `2 ft`)

OOP, Operator Overloading by DSBaral

29


Conversion from Basic to User-Defined Type (Contd...)

```

void showdist() {
    cout<<"Eng Distance: "<<feet<<"\'\'<<inch<<"\'";
};
int main(){
    engdist ed;//ed is user-defined
    float indist;//indist is basic (float)
    cout<<"\nEnter distance in Inches scale:";
    cin>>indist;
    ed=indist;//equivalent to ed=engdist(indist);
                //converts from float to engdist
    ed.showdist();
    return 0;
}
  
```

OOP, Operator Overloading by DSBaral

30



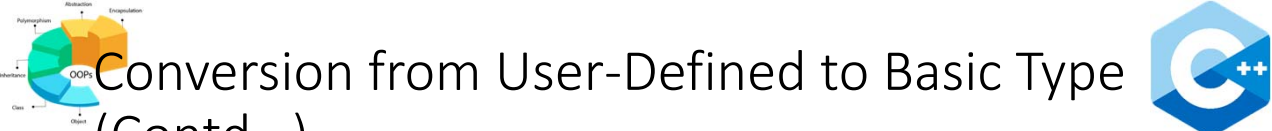
Conversion from User-Defined to Basic Type

- To convert from user defined to basic type we can define overloaded conversion (casting) operator of the destination type (basic type) as a member of the user defined class.
- Since cast operators of basic types are also overloadable they can be overloaded in the user defined class to convert from that user defined type to basic type as

```
basic_type_var=(basic_type)obj;
```
- For example

```
indist=(float)ed; //ed is of type EngDist
```

OOP, Operator Overloading by DSBaral 31





Conversion from User-Defined to Basic Type (Contd...)

- The conversion from basic type to user defined type has the following form.

```
class class_name
{
private:
    //.....
public:
    //.....
    operator basic_data_type()
    {
        //conversion code
    }
};
```

OOP, Operator Overloading by DSBaral 32






Conversion from User-Defined to Basic Type (Contd...)

- The following code shows how to overload cast operator of basic type for conversion

```
class engdist
{
public:
    //...
    void getdist(){
        cout<<"Enter feet: ";cin>>feet;
        cout<<"Enter inches: ";cin>>inch;
    }
    operator float(){
        return feet*12+inch;
    }
};
```

OOP, Operator Overloading by DSBaral 33



Conversion from User-Defined to Basic Type (Contd...)

- Now this class can be used as

```
int main()
{
    engdist ed;//ed is user-defined
    float indist;// fer is basic
    ed.getddist();
    indist=ed;//equivalent to indist=(float)ed;
    //convert from engdist to float;
    cout<<"Distance in inches: "<<indist;
    return 0;
}
```

- Actually conversion is automatically done in arithmetic expression as well.



OOP, Operator Overloading by DSBaral 34

Conversion from User Defined to User Defined Type

- Sometimes we may need to convert one user defined data type to another user defined data type.
- The C++ compiler does not support data conversion between objects of user-defined classes.
- The conversion functions for those conversions should have to be made by the programmer.
- As user defined types are also data types we can make conversion functions as a constructor that takes argument of source type object or overload cast operator function of destination type.

OOP, Operator Overloading by DSBaral 35

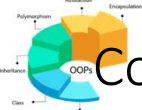




Conversion from User Defined to User Defined Type (Contd...)

- The choice between these two methods for data conversion depends on whether the conversion function be defined in source type class or destination type class.
- Let's see the following case


```
classA objA;
classB objB;
//.....
objA = objB;
```
- For this conversion the conversion function can be defined in `classA` (destination) as one argument constructor that takes argument of type `classB` (source) or in `classB` (source) as the cast (conversion) operator function of type `classA` (destination).

OOP, Operator Overloading by DSBaral 36






Conversion from User Defined to User Defined Type (Contd...)

- Conversion code in destination class has the following form.

```
//source object class
class classB {
    //body of classB
};
//destination object class
class classA {
public:
    classA(classB objB){
        //code for conversion from classB to classA
    }
};
```

OOP, Operator Overloading by DSBaral 37






Conversion from User Defined to User Defined Type (Contd...)

- Following program illustrates this concept

```
class Celsius
{
private:
    float ctemp;
public:
    Celsius(float t=0){ctemp=t;}
    float val(){return ctemp;}
    void display() {cout<<ctemp<<"degrees";}
};
```

OOP, Operator Overloading by DSBaral 38





Conversion from User Defined to User Defined Type (Contd...)

```
class Fahrenheit
{
private:
    float ftemp;
public:
    Fahrenheit(float t=0){ftemp=t;}
    Fahrenheit(Celsius ct){
        ftemp=ct.val()*9/5+32;
    }
    void display(){cout<<ftemp<<"degrees";}
};
```

OOP, Operator Overloading by DSBaral

39





Conversion from User Defined to User Defined Type (Contd...)

```
int main()
{
    Celsius cel(37);
    Fahrenheit far;
    far=cel; //interpreted as far=Fahrenheit(cel);
    cout<<"Given Celsius: ";
    cel.display();
    cout<<"\nEquivalent Fahrenheit: ";
    far.display();
    return 0;
}
```

OOP, Operator Overloading by DSBaral

40





Conversion from User Defined to User Defined Type (Contd...)

- Conversion code in source class the code has the following form.

```
//destination object class
class classA {
    //body of classA
};
//source object class
class classB {
public:
    operator classA(){
        //code for conversion from classB to classA
    }
};
```

OOP, Operator Overloading by DSBaral 41





Conversion from User Defined to User Defined Type (Contd...)

- Conversion in source class is done as follows.

```
class Fahrenheit
{
private:
    float ftemp;
public:
    Fahrenheit(float t=0){ftemp=t;}
    void display(){
        cout<<ftemp<<"degrees";
    }
};
```

OOP, Operator Overloading by DSBaral 42

Conversion from User Defined to User Defined Type (Contd...)



```

class Celsius
{
private:
    float ctemp;
public:
    Celsius(float t=0){ctemp=t;}
    float val(){return ctemp;}
    void display() {cout<<ctemp<<"degrees";}
    operator Fahrenheit(){
        return Fahrenheit(ctemp*9/5+32);
    }
};

```

- The usage is same as the conversion in earlier program as
`far=cel; //far is of type Fahrenheit and cel is of Celsius`
`//far=(Fahrenheit)cel; or far=Fahrenheit(cel);`



OOP, Operator Overloading by DSBaral 43

Conversion from User Defined to User Defined Type (Contd...)

- For vice versa conversion between objects of two classes the conversion code can be written
 - in both classes as constructors in destination classes with source type arguments
 - in both classes as cast operator of other type (destination type).
 - in one class as constructor with source type and cast operator function of other type (destination type).
 - in other class as constructor with source type and cast operator function of other type (destination type).

OOP, Operator Overloading by DSBaral 44





Assignments

- Write programs for following conversions
 - Conversion from Kg to pound system.
 - Conversion from Cartesian coordinate to polar coordinate system.
 - Conversion from meter system to feet system.
 - Conversion from 24 hour clock to 12 hour
 - Conversion from liter to gallons
 - Conversion from pound/ounce to Kg/gm
 - Conversion from meter/cm to feet/inches

OOP, Operator Overloading by DSBaral

45





Explicit Constructors

- A single argument constructor allows us to convert from the argument type of the constructor to the class type automatically.
- By default, a single argument constructor also defines an implicit conversion as

```
Test t1=2;//initializes t1 with Test(2)
or
Test t2;
//.....
t2=5;//assign t2 with Test(5)
```
- But sometimes we may need to construct object with a single value of another type but we may not want implicit conversion.

OOP, Operator Overloading by DSBaral

46

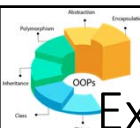



Explicit Constructors (Contd...)

- The implicit conversion can be prevented by declaring the constructor as explicit.
- The constructor is declared as explicit by placing the keyword `explicit` before the declaration of the one argument constructor as

```
class Test{
private:
    int data;
public:
    Test(){data=0;}
    explicit Test(int n){data=n;}
};
```

OOP, Operator Overloading by DSBaral 47



Explicit Constructors (Contd...)

- After declaring the constructor as explicit, it can be invoked only explicitly and will not be invoked implicitly.
- Now, after declaring one parameter constructor as explicit.

```
Test t1=2;//error: implicit conversion not allowed
or
Test t2;
//.....
t2=5;//error: implicit conversion not allowed
```

- But

```
Test t1(2);//ok: explicit conversion allowed
or
t2=Test(5);//ok: explicit conversion allowed
```

OOP, Operator Overloading by DSBaral 48