# Operator Overloading & Type Conversions

## Operator Overloading

Operator overloading is one of the crucial feature of C++ language. The concept by which we can give additional meaning to an operator of C++ language is known as operator overloading. In other words, operator overloading refers to giving the normal C++ operators (such as +, \*, <=, += etc.) additional meanings when they are applied to user-defined data types. We can redefine or overload most of the built-in operators available in C++. Thus a programmer can use operators with user-defined types as well.

**For example,** + operator in C++ work only with basic type like *int* and *float* means c = a + b is calculated by compiler if a, b and c are basic types, suppose a, b and c are objects of user defined class, compiler give error. However, using operator overloading we can make this statement legal even if a, b and c are objects.

Actually, when we write statement c = a + b (and suppose a, b and c are objects of class), the compiler call a member function of class. If a, b and c are basic type then compiler calculates a + b and assigns that to c.

**There are two types of operator overloading:**

1. Unary operator overloading (++, --, - etc.)
2. Binary operator overloading (+, \*, -, / etc.)

## Defining Operator Overloading

Operator overloading is done with the help of a special function, called ***operator function***. Operator function can be defined either friend function or member function of the class. If operator function is a member function it can be defined either inside or outside of the class definition. The general form of an operator function is:

return-type **operator** op(arglist.)

***friend*** return-type **operator** op(arglist.); //if operator function is a friend function

{

**//Function body.**

}

Where **return-type** is the type returned by the operation, **operator** is the keyword, and **op** is the operator (+, -, \*, etc.) of C++ which is being overloaded as well as function name and **arglist** is argument passed to function.

**For example,** to add two objects of type distance each having data members feet of type int and inches of type float, we can overload + operator as follows:

distance operator +(distance d2)

{

**//function body.**

}

Here, distance is a class name, operator is a keyword, and + is an operator. And we can call this operator function with the same syntax that is applied to its basic types as follows:

d3 = d1 + d2; //d1 is the calling object.

**Note:** Operator overloading is done through operator function. The operator function must be either a *non-static member function* or *friend function* of a class. Most important difference between a member function and friend function is that a friend function will have only one argument for unary operators and two for binary operators. While a member function will have no arguments for unary operators and only one for binary operators. *The reason is object used to invoke the member function is passed implicitly and thus it is available for the member function. In other word, member function can always access the particular object for which they have been called.*

## Rules For Overloading Operators

* The overloaded operator must have at least one operand that is of user-defined type.
* 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 relevant class).
* Binary operators overloaded through a member function take one explicit argument and those which are overloaded through a friend function take two explicit arguments.
* When using binary operators overloaded through a member function, the left-hand operand must be an object of the relevant class.
* Binary operators such as +, -, \*, / must explicitly return a value.

## Overloading Unary Operators (pre++, post ++, pre --, post--, unary-)

Unary operators are those operators that act on a single operand. ++, -- etc. are unary operators. Here are some examples that show how unary operators can be applied to objects.

**Syntax For Calling operator function:**

op **object\_name; `**

**E.g.: ++ S, S++, -S** etc.

**For post-increment and post-decrement:**

**object\_name** op**;**

**E.g.: S++, S--**etc.

##### Example1: Unary operator overloading for Unary Minus (-) operator

void display()

{

cout<<"x= " <<x<<" ";

cout<<"y= "<<y<<" ";

cout<<"z= "<<z<<endl;

}

};

int main()

{

Space A(4,-5,-6);

cout<<" A: ";

A.display();

-A; **//Calling operator function,**

**//equivalent to A.operator-();**

cout<<"-A: ";

A.display();

return 0;

}

class Space

{

int x;

int y;

int z;

public:

Space (){x=y=z=0;} //default constructor.

Space (int a, int b, int c)//Parameterized constructor.

{

x=a;

y=b;

z=c;

}

void operator-() **//defined inside the class.**

{

x= -x;

y= -y;

z= -z;

}

##### //Program to overload unary minus operator using friend function.

class Space

{

int x;

int y;

int z;

public:

Space(){x=y=z=0;}

Space(int a, int b, int c)

{

x=a;

y=b;

z=c;

}

friend void operator-(Space &); **//friend function, so reference argument.**

void display()

{

cout<<"x= " <<x<<" ";

cout<<"y= "<<y<<" ";

cout<<"z= "<<z<<endl;

}

};

void operator-(Space &S)

{

S.x= -S.x;

S.y= -S.y;

S.z= -S.z;

}

int main()

{

Space A(4, -5, -6);

cout<<" A: ";

A.display ();

-A; **//equivalent to operator-(A).**

cout<<"-A: ";

A.display ();

return 0;

}

##### Example2: Overloading Unary Operator (pre++)

#include <iostream>

using namespace std;

class rectangle

{

int length;

int breadth;

public:

rectangle(int l, int b)

{

length = l;

breadth = b;

}

void operator ++() **//operator function defined inside the class.**

{

++length;

++breadth;

}

void display()

{

cout<<"Length = "<<length<<endl<<"Breadth = "<<breadth<<endl;

}

};

int main()

{

rectangle r1(5, 6);

cout<<"Before increment: "<<endl;

r1. display();

++r1; **//calling operator function, same as; r1.operator ++();**

cout<<endl<<"After increment: "<<endl;

r1. display();

return 0;

}

In this example, we used prefix notation. We can also use postfix notation as follows:

void operator ++(**int**)

{

length++;

breadth++;

}

Here 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 operator. We can call this operator function as follows:

r1++; **(TRY IT OUT!!!)**

**//Program to overload pre-increment operator using friend function.**

class rectangle

{ int length;

int breadth;

public:

rectangle(int l, int b)

{

length = l;

breadth = b;

}

friend void operator ++(rectangle &); **//friend operator function declaration.**

void display()

{

cout<<"Length = "<<length<<endl<<"Breadth = "<<breadth<<endl;

}

};

void operator ++(rectangle & R)

{

++R.length;

++R.breadth;

}

int main()

{

rectangle r1(5, 6);

cout<<"Before increment: "<<endl;

r1. display();

++r1; **//calling operator function, same as; operator ++(r1);**

cout<<endl<<"After increment: "<<endl;

r1. display();

return 0;

}

##### Example3: Overloading pre-decrement(--) operator

#include<iostream>

using namespace std;

class Distance

{

int feet;

float inch;

public:

Distance (int f, float i);

void operator--(void); **//operator function declaration.**

void display();

};

Distance :: Distance (int f, float i)

{

feet = f ; inch = i;

}

void Distance :: display()

{

cout<<" Distance "<<endl;

cout<<"feet: "<<feet<<endl;

cout<<"inch: "<<inch<<endl;

}

void Distance :: operator --(void) **//operator function defined outside the class.**

{

feet--;

inch--;

}

int main()

{

Distance d(4,5);

d.display();

--d; **//calling operator function.**

d.display();

return 0;

}

##### Exercise: Redo above program using friend function.

### Operator Return Values

The operator--() function can return a value. If we use a statement like this

r1 = --r2;

For this we have to define the -- operator to have a return type object of a class in the operator--() function. That is the compiler is being asked to return whatever value r2 has after being operated on by the -- operator, and assign this value to r1.

##### Example 4: overloading for post-fix increment operator

class rectangle

{ int length;

int breadth;

public:

rectangle(){length=breadth=0;}

rectangle(int l, int b)

{

length = l;

breadth = b;

}

rectangle operator ++(int) **//operator function defined inside the class.**

{

Rectangle temp;

temp.length=length+1;

temp.breadth=breadth+1;

return temp;

}

void display()

{

cout<<"Length = "<<length<<endl<<"Breadth = "<<breadth<<endl;

}

};

int main()

{

rectangle r1(5, 6),r2;

cout<<"Before increment: "<<endl;

r1.display();

r2 = r1++;  **//calling operator function, post-fix notation.**

cout<<endl<<"After increment: "<<endl;

r2.display();

return 0;

}

## Overloading Binary Operators(+,-,\*,/,<,>,+= etc.)

Binary operators are those that work on two operands. Examples are +,-,\*, /, % for arithmetic operations, +=,-=,\*= and /= for assignment operations and >, <, <=,>=, == and! = for comparison operations.

Overloading a binary operator is similar to overloading unary operator except that a binary operator requires an additional parameter.

For binary operators, overloaded operator function can be invoked by;

X op Y, where X, Y are objects. It would be interpreted as X.operator op(Y) in case of member function and operator op(X, Y) in case of friend function by the compiler.

##### Example 5: Program for binary operator overloading for +.

class time

{

private:

int hour;

int min;

public:

time( ){ }

time(int a, int b)

{

hour =a;

min=b;

}

time operator +(time t2) //operator function

{

time temp;

temp.hour= hour + t2.hour;

temp.min=min + t2.min;

temp.hour =temp.hour + temp.min / 60;

temp.min=temp.min % 60;

return temp;

}

void display()

{

cout<<"("<<hour<<", "<<min<<")"<<endl;

}

};

int main()

{

time d1(5,78);

time d2(7,55);

time t;

t=d1 + d2; **//function call, left-hand operand d1 invokes the operator function, d2 is passed as argumnt**

d2.display();

d1.display();

t.display();

return 0;

}

**NOTE:** Remember left hand operand is always responsible for calling operator function. So in this example data members of d1 are accessed directly and the data members of d2 (that is passed as argument) are accessed using dot operator. Thus, both the objects are available for the function.

### 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.

##### Example 6: Overloading binary operator + using friend function.

class time

{

private:

int hour;

int min;

public:

time( ){ }

time(int a , int b)

{

hour =a;

min=b;

}

void display()

{

cout<<"("<<hour<<", "<<min<<")"<<endl;

}

friend time operator +(time t1, time t2); **//friend operator function**

};

time operator +(time t, time t2) **//operator function definition**

{

time temp;

temp.min=t.min+t2.min;

temp.hour= t.hour+t2.hour;

temp.hour =temp.hour+(temp.min)/60;

temp.min=(temp.min)%60;

return temp;

}

int main()

{

time d1(5,78);

time d2(7,55);

d1.display();

d2.display();

time t;

t=d1+d2; **//calling operator function.**

t.display();

return 0;

}

#### ---------------------------------------------Examples Programs--------------------------------------

##### A program for overloading + to add two C-Style strings:

#include<iostream>

#include<cstring>

using namespace std;

class String

{

private:

char str[80];

public:

String() { strcpy(str,"Rahul"); }

int main()

{

String s1("Happy "); **//calling one argument constructor**

String s2("New year"); **//calling argument constructor.**

String s3;

cout<<"Strings before overloading "<<endl;

s1.display();

s2.display();

s3=s1+s2;

cout<<"After overloading "<<endl;

s3.display();

getch();

return 0;

}

String(char s[]) { strcpy(str, s); }

void display() { cout<<str<<endl; }

String operator+(String );

};

String String:: operator+(String ss)

{

String temp;

if(strlen(str)+strlen(ss.str)<80)

{

strcpy(temp.str, str);

strcat(temp.str, ss.str);

}

else

{

cout<<"string overflow"<<endl;

}

return temp;

}

##### //Overloading < operator:

#include<iostream>

using namespace std;

class time

{

private:

int hour;

int min;

public:

time( ){}

time(int a , int b)

{

hour =a;

min=b;

}

int operator <(time t2)

{

int m = hour\*60 + min;

int n= t2.hour\*60+t2.min;

if(m<n)

return 1;

else

return 0;

}

void display()

{

cout<<"("<<hour<<", "<<min<<")"<<endl;

}

};

int main()

{

time d1(7, 41);

time d2(7, 44);

cout<<" d1: ";d1.display();

cout<<" d2: ";d2.display();

if(d1<d2)

{

cout<<" d1 is less than d2.";

}

else

{

cout<<" d2 is less than d1.";

}

return 0;

}

##### //Overloading += operator:

#include<iostream>

using namespace std;

class time

{

private:

int hour;

int min;

public:

time( ){ }

time(int a , int b)

{

hour =a;

min=b;

}

void operator +=( time t2)

{

hour+=t2.hour;

min+=t2.min;

hour+=min/60;

min=min%60;

}

time operator +=( time t2)

{

time temp;

hour+=t2.hour;

min+=t2.min;

hour+=min/60;

min=min%60;

temp.hour=hour;

temp.min =min;

return temp;

}

**OR**

void display()

{

cout<<"("<<hour<<", "<<min<<")"<<endl;

}

};

int main()

{

time d1(5,78);

time d2(7,55);

d1.display();

d2.display();

d1+=d2; **//operator function calling, d1 is calling object.**

d1.display();

return 0;

}

##### //Above program can be done using friend operator function:

#include<iostream>

using namespace std;

class time

{

private:

int hour;

int min;

public:

time( ){ }

time(int a , int b)

{

hour =a;

min=b;

}

friend void operator +=( time & t1, time & t2);

void display()

{

cout<<"("<<hour<<", "<<min<<")"<<endl;

}

};

void operator +=( time & t1, time & t2)

{

t1.hour+= t2.hour;

t1.min+= t2.min;

t1.hour+= t1.min/60;

t1.min= t1.min%60;

}

int main()

{

time d1(5,78);

time d2(7,55);

d1.display();

d2.display();

operator +=(d1,d2);

d1.display();

return 0;

}

##### //Overloading == operator:

class Distance

int Distance::operator==(Distance d)

{

if(feet == d.feet && inch == d.inch)

return 1;

else

return 0;

}

int main()

{

Distance d1(4,89.3);

Distance d2(4,89.2);

if (d1==d2)

cout<<"Both distances are of same length";

else

cout<<"Different";

return 0;

}

{

int feet;

float inch;

public:

Distance()

{}

Distance(int f, float i)

{

feet = f;

inch = i;

}

void display()

{

cout<<feet<<inch;

}

int operator ==(Distance);

};

##### //Overloading == operator to compare two C-style Strings:

#include<iostream>

int String::operator==(String d)

{

if(strcmp(Str, d.Str)==0)

return 1;

else

return 0;

}

int main()

{

String d1("Ajay ");

String d2("Pandey ");

if (d1==d2)

cout<<"Both Strings are same ";

else

cout<<"Different";

return 0;

}

#include<cstring>

using namespace std;

class String

{

char Str[200];

public:

String()

{}

String(char b[])

{

strcpy(Str, b);

}

void display()

{

cout<<"String "<<Str;

}

int operator ==(String);

};

##### Operator Overloading Restrictions

1. The precedence of the operator cannot be changed.
2. The number of operands that an operator takes cannot be altered.
3. The overloaded operators must have at least one user-defined operand.
4. Only the existing operators can be overloaded. New operators cannot be created.
5. Operator functions cannot have default arguments.
6. We cannot overload following C++ operators:

Class member access operators (**.\***) Scope resolution operator (**::**)

Size operator (**sizeof**) Conditional operator (**?:**).

1. We cannot use friend functions to overload certain operators. These operators are:

= assignment operator ( ) function call operator

[ ] subscripting operator -> class member access operator

#### Exercise:

1. WAP to overload == operator using friend function.
2. WAP to overload unary minus (-) operator to invert sign of data members of a distance object.
3. Write a program to overload arithmetic assignment operators -= for two time objects.
4. Write a program to overload <= and >= operators.
5. Write a program to concatenate two strings using + operator.
6. Write a program to compare two C-style strings using = = operator.(use strcmp())

## Type Conversion (Data Conversion)

Data conversion is the process of converting one data type to another data type. The type conversions are automatic as long as the data types involved are built-in types. If the data types are user defined, the compiler does not support automatic type conversion and therefore, we must design the conversion routines by ourselves.

There are four types of data conversions;

1. Conversion from one basic type to another basic type
2. Conversion from basic type to class type
3. Conversion from class type to basic type
4. Conversion from one class type to another class type

##### Syntax: data conversion between incompatible types

*target\_type* target\_type\_variable\_name = source\_type\_variable\_name;

Here target type may be user defined type or built-in type and also source type may be either user defined type or built-in type.

**Example:**

Time t;

int a=700;

t = a; **//here built-in type int is converted to user defined type Time.**

### Conversion from one basic type to another basic type

There are two types of conversion for built-in data type (int, float, char etc.).

Implicit data Conversion*:* Compiler is responsible for this type of data conversion. We do not need to write any conversion routine. Compiler automatically converts one basic type to another basic type.

For example:

int x;

float y=3.14 ;

x=y;

In above example, we are assigning the value of variable y to variable x. To achieve this, the compiler first converts y into integer and then assign it to x.

#### Explicit Conversion*:* In some cases automatic conversion may not work. Users have to write conversion routine explicitly. Explicit conversion is performed using type cast operator. Three forms of type cast operator;

* type\_name (expression) **//c++ notation**
* (type\_name) expression **//c notation**
* variable1 = static\_cast<target\_type> (variable2), where variable1 is target type variable and variable2 is source type variable.

**For example:**

int i, sum; float avg;

avg = sum/ (float)i; **//c notation**

avg = sum/ float (i); **//c++ notation**

avg = static\_cast<float>(sum)

### Conversion from basic type to class type(Basic type to User defined type)

The conversion from basic to class type can be done by using constructor. It is sometimes called *conversion constructor*. The constructor in this case takes single argument whose type is to be converted.

**Syntax:**

**class\_name(**an argument of basic type**)**

**{**

**//conversion routine.**

**}**

**Example:** Program to convert built-in type *int* to user-defined type *Time*.

#include<iostream>

using namespace std;

class Time

{

int hour;

int min;

public:

Time() **//default constructor.**

{}

Time(int t) **//Conversion constructor.**

{

hour = t/60;

min = t%60;

}

void show( )

{

cout<<hour<< " Hour and "<<min<<" Minutes";

}

};

int main()

{

int a = 789;

Time T1 = a; **// uses one-argument constructor to convert integer to time.**

T1.show();

return 0;

}

**Example:** Program to convert built-in type *float* to user-defined type *Distance*

class Distance

{

int feet;

float inches;

public:

Distance() **//default constructor.**

{}

Distance(float meters) **//conversion constructor.**

{

int main()

{

float meter;

cout<<"Enter a distance in meters: "<<endl;

cin>>meter;

Distance d1= meter; **//uses one argument constructor to convert float to Distance.**

d1.show();

return 0;

}

float f= meters\*3.28; //since,1 meter = 3.28 feet.

feet = int(f) ;

inches = (f-feet)\*12;

}

void show( )

{

cout<< " distance is: " <<endl ;

cout<<feet<< " Feet and "<<inches<<" Inches";

}

};

### Conversion from class type to basic type(User-Defined to Basic Type)

When class type data is converted into basic type data, it is called class to basic type conversion. The conversion constructor does not support this operation. This type of conversion is done through overloaded casting operator function, also called *conversion function*.

The general form of a conversion function is;

**operator** type\_name()

{

**//function statements.**

}

This function converts a class type data to type\_name. For example, the **operator** double() converts a class object to type double, **operator** int() converts a class type object to type int, and so on. Since it is a class member function, it is invoked by the object; therefore values inside the function belong to the object that invoked the function.

**Properties of Conversion Function:**

* It must be a class member.
* It must not specify a return type.
* It must not have any arguments.

**Example:** Program to convert user-defined type i.e, *Distance* to basic type *float* i.e., meters.

class Distance

{

private:

int feet;

float inches;

public:

Distance(int f, int i)

{

feet=f;

inches=i;

}

operator float() **//conversion function, converts *Distance* type to a basic type *float*.**

{

float ft=inches/12 ;

ft=ft+feet ;

return(ft/3.28) ;

}

};

int main()

{

Distance d(12, 6.56);

float x = d; **//conversion function calling, d is responsible for calling.**

cout<<"x = "<<x<<" meters.";

return 0;

}

**Example:** Program to convert user-defined type i.e., *Time* to basic type *int* i.e, seconds.

#include<iostream>

using namespace std;

class Time

{

private:

int hour;

int min;

public:

Time(int h, int m)

{

hour=h;

min=m;

}

operator int() **//conversion function, converts class type Time to a basic type int.**

{

int s = hour\*60\*60+min\*60;

return s;

}

};

int main()

{

Time T(10, 56);

int s = T; **//conversion function calling, T is responsible for calling.**

cout<<s<<" seconds.";

return 0;

}

### Conversion from one class type to another class type (one user-defined to another user defined type)

When a data of one class type is converted into data of another class type, it is called conversion of one class to another class type.

We can convert one class (object) to another class type as follows:

object of X = object of Y, X and Y both are different type of classes.

Here X is an object of class X and Y is object of class Y. The class Y type data is converted to the class X type data and converted value is assigned to the X. The conversion takes place from class Y to Class X. Therefore, Y is source class and X is destination class. This type of conversion is carried out by either constructor or a conversion function. It depends upon where we want the routine to be located – in the source class or in the destination class.

#### a. Routine in the source class:

When the conversion routine is in source class, it is implemented as a conversion function i.e. the overloaded casting operator function.

**Syntax:**

operator target\_type\_name() **//here type-name is the destination class name.**

{

**//function body.**

}

##### Example: Program to convert one user defined type Rupee to another user defined type Dollar.

//Here, conversion routine is in the source class.

#include<iostream>

using namespace std;

class Dollar **//Destination class**

{

float Dol;

float cent;

public:

Dollar(float a, float c)

{

Dol=a;

cent=c;

}

void display()

{

cout<<Dol<<" $ and "<<cent<<" Cents."<<endl;

}

};

class Rupee **//Source class**

{

float Rs;

float Paisa;

public:

Rupee(float r, float p)

{

Rs=r;

Paisa=p;

}

void display()

{

cout<<Rs<<"Rs. and "<<Paisa<<" Paisa."<<endl;

}

operator Dollar() **//this operator function is responsible for conversion.**

{

float a, b;

a=Rs/100;

b=Paisa/100;

return Dollar(a, b);

}

};

int main()

{

Rupee R(45.23,76.23);

R.display();

Dollar D = R; **//Here, R is a source type and D is a destination type.**

D.display();

return 0;

}

#### b. Routine in the destination class:

When the conversion routine is in destination class, it is commonly implemented as a constructor. This constructor takes one argument of source type i.e. an object of source type.

**Syntax:**

class-name(an object of source type)

{

**//conversion routine.**

}

**Example:** Program to convert one user defined type Rupee toanother user defined type Dollar..

##### //Here, conversion routine is in the destination class.

#include<iostream>

class Dollar

{

float Dol;

float cent;

public:

Dollar(){}

Dollar(float a, float c)

{

Dol=a;

cent=c;

}

Dollar(Rupee R)

{

Dol=R.getRs()/100;

cent=R.getPaisa()/100;

}

void display()

{

cout<<Dol<<" $ and "<<cent<<" Cents."<<endl;

}

};

int main()

{

Rupee R(45.23,700.23);

R.display();

Dollar D = R;

D.display();

return 0;

}

using namespace std;

class Rupee

{

float Rs;

float Paisa;

public:

Rupee(float r, float p)

{

Rs = r;

Paisa = p;

}

void display()

{

cout<<Rs<<"Rs. and "<<Paisa<<" Paisa.";

}

float getRs()

{

return Rs;

}

float getPaisa()

{

return Paisa;

}

};

##### Exercise:

1. WAP to convert Centigrade into Fahrenheit temperature. ( F= 9\*C/5 + 32)
2. Define two classes **Polar** and **Rectangle** to represents points in the polar and rectangle system, where;

x = r cos a

y = r sin a. Write a conversion routine to convert object of class **Polar** to object of class **Rectangle**.

1. Create classes called **Amount1**, **Amount2**, and **Amount3**. **Amount1** has data member Rs(float), Amount2 has data member Paisa(int) and Amount3 has data member Dollars(float). The classes must be defined in such a way that it is possible to convert **Amount2** into **Amount1** and **Amount1** into **Amount3**.