# Operator Overloading & Type Conversion

In C++

# Operator Overloading

- Operator overloading is a compile-time polymorphism in which the operator is overloaded to provide the special meaning to the user-defined data type.
- Operator overloading is used to overload or redefines most of the operators available in C++.
- It is used to perform the operation on the user-defined data type.
- For example, C++ provides the ability to add the variables of the user-defined data type that is applied to the built-in data types.

**Operator that cannot be overloaded are as follows:**

- Scope operator (::)
- member selector(.)
- member pointer selector(*)
- ternary operator(?:)
- Size of (Sizeof)

The reason for not overloading these operators is because these operator take names (example class name) as their operand instead of values, as is the case with other normal operators.

# Defining Operator Overloading

- To Define an additional task, we must specify what it means in relation the class to which the operator is applied, This is done with the help of a special function called *Operator Function,* That describes the task.

e.g.

```
return type classname :: operator op(arglist)
                {
                    Function Body
                }
```

- Return type is the type of the value that returned back, op is the operator to be overloaded. Here operator op is the function name.

Operator functions will be either member functions or the friend functions.

Differences is that:

1. Friend Function will have one argument for unary operator and two arguments for binary operator.
2. While member function will have no argument for unary operator and one argument for binary operator.

# Defining Operator Overloading

- The Process of Overloading involves following steps:

1. Create a class to define a data type.
2. Declare the operator function ***operator op()*** in public part of class, It may be either member function or friend function.
3. Define operator function to implement required operation.

To invoke these operator functions we have to use the operators with the operands that are the variables of the class type.

e.g. For Unary Operator: **op x** OR x op

For Binary operator: **x op y**

# Rules for Operator Overloading

Though we can overload the operators for the user defined data types easily, Yet it has the rules to be followed:

1. Only existing operators can be overloaded, new operators can not be created.
2. The overloaded operator must have at least one operand that is of user defined data type.
3. We can not change the meaning of the operator, i.e. the plus(+) cannot be overloaded for subtraction.
4. Overloaded operators follow the syntax rules of the original operators, they cannot be overridden.
5. There are some operators that cannot be overloaded. E.g. .*, . , ::, ?::
6. There are some operators that cannot be overloaded using friend function, but they can be overloaded using the member functions. E.g. =, (), [], ->
7. Unary operators, overloaded by means of a member function, take no explicit arguments, and return no explicit value, but, those overloaded by means of a friend function take one reference argument (the object of relevant class).
8. Binary operators, overloaded through a member function, take one explicit argument and those which are overloaded through a friend function take two explicit arguments.
9. While overloading Binary operators through a member function, the left hand operand must be an object of the relevant class.
10. Binary arithmetic operators such as + , - , * , and / must explicitly return a value. They must not attempt to change their own arguments.

# Overloading Unary Operators

- Let us take an example of unary minus operator that changes the sign of the operand, when applied to basic data item, We are here going to learn that how this operator can be applied to the objects in the same way as It is applied to the in built data types(such as int, float etc.) When Unary operator minus(-) applied to an object it must change the sign of each of its data items.

Programming Example:

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

class space
{
   int x;
   int y;
   int z;
public:
   void getdata(int a, int b,int c);
   void display(void);
   void operator-();
 };
```

```
void space::getdata(int a,int b, int c)
{
   x=a;
   y=b;
   z=c;
}

void space::display(void)
{
      cout<<x <<"   ";
      cout<<y <<"   " ;
      cout<<z <<"\n";
}

void space::operator-()
{
   x= -x;
   y= -y;
   z= -z;
}
```

```
 int main()
{
    clrscr();
        space s;
        s.getdata(10,-20,30);
        cout<<"S : ";
        s.display();

        -s;
        cout<<"S : ";
        s.display();
        getch();
    return 0;

}
```

The same unary minus operator can be overloaded using a friend function.

```
friend void operator-(space &s);

void operator-(space &s)
   {
       s.x = -s.x;
       s.y = -s.y;
       s.z = -s.z;
    }
```

Here the argument has been passed by reference, if it is passed by value it will not work coz only a copy of the object will be passed to the function operator-(). Therefore the changes made inside the operator function will not reflect in the called object.

# Overloading Binary Operators

- Binary operator overloading will enable us to apply the operators on the objects (i.e. variables of user defined data types) in the same way they are applied to the in-built data types (i.e. int, float etc.)

Programming Example for Binary Operator +

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

class complex
 {
    float x; float y;
  public:
    complex(){}
    complex(float  real, float imag)
    {    x=real;  y=imag;
     }

     complex operator+(complex);
     void display(void);
};
```

```
complex complex::operator+(complex c)
  {
     complex temp;
     temp.x=x+c.x;
     temp.y=y+c.y;
      return(temp);
}

void complex::display(void)
 {

   cout<<x<<"  +j  "<<y<<"\n";

}
```

```
int main()
{

     clrscr();
    complex c1,c2,c3;

    c1=complex(2.5,3.5);
    c2=complex(1.6,2.7);
    c3=c1+c2;

 cout<<"C1 = "; c1.display();
 cout<<"C2 = "; c2.display();
 cout<<"C3 = "; c3.display();

return 0;
}
```

output:

```
5+10j
```

```
complex complex::operator+(complex c)
  {
     complex temp;
     temp.x=x+c.x;
     temp.y=y+c.y;
      return(temp);
}
```

Here two objects are being added and one is being returned, But we are sending only one object as an argument, where does the other value come from.

Actually the statement C3=C1+C2 acts as **C3=C1.operator+(C2)**

i.e. the C1 object is invoking the operator function and C2 is being sent as an argument.

Hence the C1 accessed directly and C2 sent as an argument.

Here members of C2 operator accessed through dot operator. Temp.x = x+c.x;
In overloading As a Rule, The left hand operand is used to invoke the operator function and the right hand operand is passed as an augment. The same can also be done as: return complex ((x+c.x),(y+c.y));

# Overloading Binary Operators Using Friend Functions

- As Stated Earlier Binary operators, overloaded through a member function, take one explicit argument and those which are overloaded through a friend function take two explicit arguments. However Overloading Binary Operator using Friend Function is done as follows:

1. Replace the member function declaration by Friend Function Declaration

   e.g. **Friend complex operator+(complex &,complex &);**

2. Redefine the operator function as follows:

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

In this case C3=C2+C1;

means

C3=operator +(C1,C2);

The Reasons why we have two methods to define operator functions:

1. Consider a situation where we need to use operands of different types for a binary operator say one is an object and other is a variable of built-in data type.

i.e. A= B+2; (or A=B*2 ;) Here A & B are Objects of same class.

Here member function will not work for A=2+B (or A=2*B;)

However it is possible using Friend Function.

Where a friend function cannot be used:

- Assignment operator : =
- Function call operator : ( )
- Subscripting operator : [ ]
- Class member access operator : ->

# Type Conversions

In C++

# Introduction

1. Whenever an expression appears with different types of operands the type of the operands is automatically converted.
2. Similarly in case of the assignment statement the type of the operand on the R.H.S is automatically converted to the type of the variable on the L.H.S.

e.g.
```
int m;
float x = 3.149519
m=x;
```

The value of the x is converted to int before being assigned to m.

There are mainly two types of type conversion:

**1. implicit**- This is also known as automatic type conversion. This is done by the compiler without any external trigger from the user. This is done when one expression has more than one datatype is present.

All datatypes are upgraded to the datatype of the large variable.

```
#include <iostream>
using namespace std;
int main() {
  int a = 10;
    char b = 'a';
    a = b + a;
  float c = a + 1.0;
    cout << "a : " << a << "\nb : " << b << "\nc : " << c;
}
```

**Output**

```
a : 107
b : a
c : 108
```

**2. Explicit**-This is also known as type casting. Here the user can typecast the result to make it to particular datatype. In C++ we can do this in two ways, either using expression in parentheses or using static_cast or dynamic_cast.

```
#include <iostream>
using namespace std;
int main() {
  double x = 1.574;
  int add = (int)x + 1;
    cout << "Add: " << add;
  float y = 3.5;
    int val = static_cast<int>(y);
    cout << "\nvalue: " << val;
}
```

**Output**

Add: 2

value: 3

This process is automatic until the data types are in-built.

- What happens when the data types are user defined.
- We may have the cases when we are trying to use the expressions where one operand is of user defined data type and other is in-built data type or the vice versa.
- Hence we have three type of conversions:

  (i) Conversion from basic type to class type

  (ii) Conversion from class type to basic type

  (iii) Conversion from one class type to the other class type

# Basic to Class Type

- Here the constructors can be used to convert the in-built data type variables to the class type variables.
- E.g.

```
string:: string(char *a)
 {
     length = strlen(a);
      P=new char[length+1];
     strcpy(P,a); }
```

Here constructor builds string type object from an array of char. Here length and P are the members of the class string. Once the constructor has been defined the type can be converted.

E.g.

```
  string s1,s2;
char* name1 = " Programming";
char* name2 = "Computers";
   s1=string(name1);
   s2=name2;      /* Here the constructor is invoked implicitly*/
```

# Basic to Class Type

- Here is another example:

```
class time
    {
      int hrs;
      int mins;
    public:
        ::::::
      time(int t)
      {
       hours= t/60;
        mins=t%60;
      }
};
  time T1;
 int duration = 85;
    T1=duration;
```

Imp: The constructor used for type conversion takes single argument whose type is to be converted.

# Class Type to Basic Type

- Here constructors does not help, here we use Overloaded Casting Operator, that will convert the class type data to the basic type. The General syntax for *overloaded casting operator function also called conversion function* .

```
operator typename()
    {
    :::::::::::::   (Function statements)
    }
```

This will convert the class type data to typename. E.g. Operatror int() convert class type data to int.

e.g

```
class time
{     public:
            int hours , min;
            time (int x, int y)
            {
             hours=x;
             min=y;
            }
  operator int();
};
```

# Class Type to Basic Type

```
time::operator int()
{
 int totalmin;
 totalmin=hours*60+min;
 return totalmin ;
}
```

This function converts the time to corresponding total no. minutes.

e.g. **int totalmin = int (t);** or **int totalmin = t;**

here t is the object of class time;

Whenever this type of the statements appears the compiler automatically calls the casting operator functions to do the job. The conditions for the casting operator function are:

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

As it is a member function, It has to be invoked by an object, and hence it will automatically use the values of the object for the conversion i.e it doesn't need any argument.

# One Class Type to Another Class Type

This kind of situations occur when we have **objX=objY**;

objects of different classes.

Here first the class Y type data is to be converted to Class X type data and then converted values are to assigned to the objX.

Here Y is known as *source class* and X is called *Destination Class.*

Here we can use either Constructor or Conversion Function.

The selection depends on where we want to place the conversion function in source class or in destination class.

Here we know that operator typename() converts class object *of which it is a member to typename.* Here typename may be a user defined data type that is class i.e destination class.

*Here the conversion is done in the source class and the result is assigned to the destination class.*

In case we are using a single argument constructor to convert then the argument belongs to the source class and is passed to the destination class for conversion. Here the conversion constructor must be placed in the destination class for conversion.

# Summary

| Conversion Required | Conversion takes place in | |
|---|---|---|
| | Source Class | Destination Class |
| Basic -> class | Not Applicable | Constructor |
| Class - > basic | Casting Operator | Not Applicable |
| Class - > class | Casting Operator | Constructor |

Whenever we are using the constructor in the destination class, we must be able to access the data members of the objects sent, If the data members of the class are private then we have to use the special access function in the source class to facilitate its data flow to the destination.

Program Example

# One Class to another Class Type

```
class timeS
{
        int hours;
        int min;
         public:
        timeS (int h, int min)    ;
        {                hours=h;
                                min=m;       }
              void displayS()
        {      cout<<hours<<"\n"<<min;     }
               int gethours()
               {      return hours;   }
                int getmin()
               {      return min;      }

operator timeD()
};
```

```
/* timeS::operator timeD()
{
timeD temp;
temp.totalmin= hours*60+min;
 return temp;
} */
Class timeD
{
        int totalmin;
          public:
        timeD (int t)
                {
                   totalmin=t;
                }
        void displayD()
        {
           cout<<totalmin;
        }
```

```
timeD( timeS p)
{
  h=p.gethours();
  m=p.getmin();
  totalmin= h*60+m;
}
};
void main()
{
timeS tS(10, 20);
tS.dislayS();
timeD tD;
tD= tS;
tD.displayD();
getch();
}
```

# Abstract Classes

- struct A { virtual void f() = 0; }; struct B : A { virtual void f() { } }; // Error: // Class A is an abstract class // A g(); // Error: // Class A is an abstract class // void h(A); A& i(A&); int main() { // Error: // Class A is an abstract class // A a; A* pa; B b; // Error: // Class A is an abstract class // static_cast<A>(b); } Class A is an abstract class. The compiler would not allow the function declarations A g() or void h(A), declaration of object a, nor the static cast of b to type A.
- Virtual member functions are inherited. A class derived from an abstract base class will also be abstract unless you override each pure virtual function in the derived class.
- For example:
- class AB { public: virtual void f() = 0; }; class D2 : public AB { void g(); }; int main() { D2 d; } The compiler will not allow the declaration of object d because D2 is an abstract class; it inherited the pure virtual function f()from AB. The compiler will allow the declaration of object d if you define function D2::g().
- Note that you can derive an abstract class from a nonabstract class, and you can override a non-pure virtual function with a pure virtual function.
- You can call member functions from a constructor or destructor of an abstract class. However, the results of calling (directly or indirectly) a pure virtual function from its constructor are undefined. The following example demonstrates this:
- struct A { A() { direct(); indirect(); } virtual void direct() = 0; virtual void indirect() { direct(); } }; The default constructor of A calls the pure virtual function direct() both directly and indirectly (through indirect()).
- The compiler issues a warning for the direct call to the pure virtual function, but not for the indirect call.

# Empty Classes

- You can declare empty classes, but objects of such types still have nonzero size.
- // empty_classes.cpp

```
// compile with: /EHsc
#include <iostream>
  class NoMembers { };
    using namespace std;
       int main()
           {
           NoMembers n; // Object of type NoMembers.
       cout << "The size of an object of empty class is: “ << sizeof n << endl;
           }
```

- Output

```
The size of an object of empty class is: 1
```

- The memory allocated for such objects is of nonzero size; therefore, the objects have different addresses. Having different addresses makes it possible to compare pointers to objects for identity. Also, in arrays, each member array must have a distinct address.