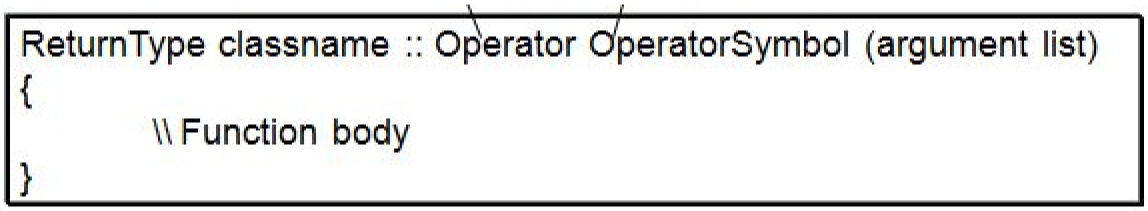
Introduction

The mechanism of giving special meaning to an operator is known as operator overloading. Operator overloading provides flexible option for the creation of new definitions for most of the C++ operators. We can overload (give additional meaning) to all the C++ operators except the following:

- scope operator (::)
- sizeof
- member selector (.)
- member pointer selector (*)
- ternary operator (?:)

To define an additional task to an operator, we must specify what it means in relation to the class to which the operator is applied. This is done with the help of a special function called **operator function.** The general form of operator function is:



```
ReturnType classname :: Operator OperatorSymbol (argument list)
{
        \\ Function body
}
```

Example:

```
int employee :: operator + (employee e)
{
        // function definition
}
```

Implementing Operator Overloading in C++

Operator overloading can be done by implementing a function which can be :

1. Member Function
2. Non-Member Function
3. Friend Function

Operator overloading function can be a member function if the Left operand is an Object of that class, but if the Left operand is different, then Operator overloading function must be a non-member function.

Operator overloading function can be made friend function if it needs access to the private and protected members of the class.

Rules for operator overloading

There are following rules of operator overloading given below:

1. The first and basic rule of operator overloading is: we can overload unary operator as only unary operator, it cannot be overload as binary operator and vice versa.
2. We cannot overload those operators that are not a part of C++ language like '$'.
3. We can perform operator overloading in only user defined classes. We cannot change the operators existing functionality.
4. Using operator overloading we cannot change the presidency and associatively of operators.
5. There are some operators cannot be overloaded that are given below:
    - :: Scope resolution operator.
    - . Class membership operator.
    - ?: ternary or conditional operator.
    - .* pointer to member operator.

- ->* pointer to member operator.

Overloading unary operator

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.
- Address of Operator (*).
- Cast operator ().
- Complement operator (~) and so on.

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 an overloading of an operator operating on a single operand.

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

Example:

```
#include <iostream>
using namespace std;

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 minus (-) operator
      Distance operator- () {
        feet = -feet;
        inches = -inches;
        return Distance(feet, inches);
      }
};

int main() {
   Distance D1(11, 10), D2(-5, 11);

   -D1;                     // apply negation
   D1.displayDistance();    // display D1
   -D2;                  // apply negation
   D2.displayDistance();    // display D2

   return 0;
}
```

Output:

```
F: -11 I:-10

F: 5 I:-11
```

Overloading binary Operator

In binary operator overloading function, there should be one argument to be passed. It is overloading of an operator operating on two operands.

Following is an example to overload binary (+) operator .

Example:

```
#include<iostream>
#include<string.h>
using namespace std;

class test
{
   private:
          char name[40];
```

```
    public:
            void getData(char s[])
            {
                    strcpy(name,s);
            }

            void display()
            {
                    cout<<"Full Name:"<<name<<endl;
            }

            test operator+ (test s){
                    test temp;
                    strcpy(temp.name,name);
                    strcat(temp.name," ");
                    strcat(temp.name,s.name);
                    return temp;

            }
};

int main()
{
    test t1,t2,t3;
    t1.getData("Donald");
    t2.getData("Trump");
    t3=t1+t2 ; // Equivalent to t3=t1.operator+(t2)
    t3.display();
    return 0;
}
```

Output

Full name: Donald Trump

Overloading I/O operator (Overloading insertion << and extraction >> operator)

```
#include <iostream>
using namespace std;

class Complex
{
private:
        int real, imag;
public:
        Complex(int r = 0, int i =0)
        {  real = r;   imag = i; }
        friend void  operator << (ostream &out, const Complex &c);
        friend void  operator >> (istream &in,  Complex &c);
};

void  operator << (ostream &out, const Complex &c)
{
         out << c.real<< "+i" << c.imag << endl;
}

void  operator >> (istream &in,  Complex &c)
{
        cout << "Enter Real Part :";
        in >> c.real;
        cout << "Enter Imaginary Part :";
        in >> c.imag;
}

int main()
{
  Complex c1;
  cin >> c1;
  cout << "The complex object is ";
  cout << c1;
  return 0;
}
```

Output:

```
Enter Real Part :12
Enter Imaginary Part :13
The complex object is 12+i13
```

Example: Program to overload compliment (~) operator.

```
#include <iostream>
using namespace std;

class test{
    private:
          int num;
    public:
          void setData(int x)
          {
           num=x;
          }

          void display()
          {
           cout<<"Num:"<<num<<endl;
          }

          test operator~()
          {
           test temp;
           temp.num=~num;
           return temp;
          }
};



int main()
{
          test t;
          t.setData(10);
          test t1=~t;
          t.display();
          cout<<"After operator overloading:"<<endl;
          t1.display();
          return 0;
}
```

Output:

```
Num:10
After operator overloading:
Num:-11
```

Example: Program to overload ! operator.

```
#include <iostream>
using namespace std;
class test{

   private:
           bool status;
   public:
           void setStatus(bool x)
           {
            status=x;
           }

           void display()
           {
             cout<<"Status:"<<status<<endl;
           }

           test operator!()
           {
            test temp;
            temp.status=!status;
            return temp;
           }
};



int main()
{
           test t;
           t.setStatus(false);
           test t1=!t;
```

```
        t.display();
        cout<<"After operator overloading:"<<endl;
        t1.display();
        return 0;
}
```

Output

```
Status:0
After operator overloading:
Status:1
```

Operator overloading using friend function

Example: Program to overload arithmetic operator (*) using friend function.

```
#include <iostream>
using namespace std;
class test{

        private:
        int num1, num2;
        public:
        void setData(int x, int y)
        {
        num1=x;
        num2=y;
        }

        void display()
        {
        cout<<"First Number:"<<num1<<endl;
        cout<<"Second Number:"<<num2<<endl;
        }
        friend test operator*(test,test);
};

test operator*(test t1, test t2)
{
   test temp;
   temp.num1=t1.num1*t2.num1;
   temp.num2=t1.num2*t2.num2;
   return temp;
}

int main()
{
        test t1,t2;
        t1.setData(10,20);
        t2.setData(30,40);
        test t3=t1*t2;
        cout<<"First Object"<<endl;
        t1.display();
        cout<<"Second Object"<<endl;
        t2.display();
        cout<<"After operator overloading:"<<endl;
        t3.display();
        return 0;
}
```

Output:

```
First Object

First Number:10
Second Number:20

Second Object

First Number:30
Second Number:40
```

```
After operator overloading:

First Number:300
Second Number:800
```

Restrictions on Operator Overloading in C++

Following are some restrictions to be kept in mind while implementing operator overloading.

1. Precedence and Associativity of an operator cannot be changed.
2. Arity (numbers of Operands) cannot be changed. Unary operator remains unary, binary remains binary etc.
3. No new operators can be created, only existing operators can be overloaded.
4. Cannot redefine the meaning of a procedure. You cannot change how integers are added.

Data Type Conversion

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

Example 1:

```
float a;
int x , y;
x=5;
y=6;
a=x+y; (Automatic/ implicit type conversion)    // 11 will be output
a=x/float(y)   (Explicit type conversion)   // 0.83333 will be output
```

Example 2

```
int a;        // basic type

class student
{
        ......
} s1,s2,s3;

s1=a;
a=s1;
```

In this case, we need to design conversion routine.

Example 3

```
class student
{
 ......
} s1, s2;


class employee
{
...
} e1, e2,e3;

e1=s1;
s1=e1;
```

In this case also, we need to design conversion routine.

Hence, from above illustration three types of situations might arise in the data conversion which are;

1. Conversion from basic type to class type (object type).
2. Conversion from class type to basic type.

3. Conversion between one class type to another class type.

Conversion from basic type to class type

In this case, it is necessary to use the constructor. The constructor in this case takes a single argument whose type is to be converted.

For Example:

Conversion from class to basic type

C++ allows us to define an overloaded operator casting function that could be used to convert a class type data to basic type.

General Form

```
operator type_name()
{
        //conversion routine
}
```

```
Operator int()
{
  return (a*b);
}
```

The casting operator function should specify following conditions (characters)

- It must be a class member.
- It mustn't have any return type.
- It mustn't have any arguments.

Example:

Conversion from class to class type

There is another situation where we've to convert one class type data to another class type data. For example:

```
class student
{
 ......
} s1;


class employee
{
...
} e1;

s1=e1;
```

Where s1 is an object of class student and e1 is an object of class employee. Class employee is converted into class student. Here, employee is known as source class and student is known as destination class. Conversion between objects of different classes can be carried out by either a **constructor** function or **operator casting** function. It depends upon where we want the conversion function to be located in the source or in the destination class. If we want to place in destination class then constructor function is used and if we want to place in source class then operator casting function is used.

Example: Using a constructor function in destination class.

```
#include<iostream>
using namespace std;

class employee
{
   char name[20];
   public:
          void getData()
          {
                   cout<<"Enter employee Full Name:"<<endl;
```

```
                        cin.getline(name,20);

            }

            char* giveName()
            {
                        return name;
            }
};

class student
{
    char name[20];
    public:
            student()
            {
            }

            student(employee e)
            {
                        cout<<"Student Full Name is :"<<e.giveName();
            }

};

int main()
{
    student s1;
    employee e1;
    e1.getData();
    s1=e1;   // s1=student(e1);
    return 0;

}
```

output:

```
Enter employee Full Name:
Hari Thapa
```

```
Student Full Name is :Hari Thapa
```

Example : Using operator casting function in source class

```
#include<iostream>
using namespace std;

class student
{
   char name[20];
   public:
         student()
         {
         }

         student(char* name)
         {
                 cout<<"Student full name is :"<<name;
         }

};

class employee
{
   char name[20];
   public:
         void getData()
         {
                 cout<<"Enter employee Full name:"<<endl;
                 cin.getline(name,20);

         }

         operator student()
         {
                 return student(name);
         }
};
```

```
int main()
{
   student s1;
   employee e1;
   e1.getData();
   s1=e1;   // checks for operator function with student type
   return 0;

}
```

Output:

```
Enter employee Full name:
sanjay Kumar Chaudary
Student full name is :sanjay Kumar Chauda
```