Operators overloading in C++:

tutorialspoint.com/cplusplus
en.wikibooks.org/wiki/C%2B%2B_Programming
http://www.learncpp.com
www3.ntu.edu.sg

Operators such as =, +, and others can be redefined when used with objects of a class

The name of the function for the overloaded operator is `operator` followed by the operator symbol, *e.g.*,

- `operator+` to overload the + operator, and
- `operator=` to overload the = operator

Prototype for the overloaded operator goes in the declaration of the class that is overloading it

Overloaded operator function definition goes with other member functions

Like any other function, an overloaded operator has a return type and a parameter list.

```
Box operator+(const Box&);
```

declares the addition operator that can be used to **add** two Box objects and returns final Box object.

Binary operators overloading in C++

Following is the example to show the concept of operator overloading using a member function.

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

class Box
{
   public:

      double getVolume(void)
      {
         return length * breadth * height;
      }
      void setLength( double len )
      {
          length = len;
      }

      void setBreadth( double bre )
      {
          breadth = bre;
      }

      void setHeight( double hei )
      {
          height = hei;
      }
```

Inline Member Functions

```
      // Overload + operator to add two Box objects.
      Box operator+(const Box& b)
      {
         Box box;
         box.length = this->length + b.length;
         box.breadth = this->breadth + b.breadth;
         box.height = this->height + b.height;
         return box;
      }
   private:
      double length;      // Length of a box
      double breadth;     // Breadth of a box
      double height;      // Height of a box
};
```

Here an object is passed as an argument whose properties will be accessed using this object, the object which will call this operator can be accessed using **this** operator as explained below:

```
// Main function for the program
int main( )
{
   Box Box1;                // Declare Box1 of type Box
   Box Box2;                // Declare Box2 of type Box
   Box Box3;                // Declare Box3 of type Box
   double volume = 0.0;     // Store the volume of a box here

   // box 1 specification
   Box1.setLength(6.0);
   Box1.setBreadth(7.0);
   Box1.setHeight(5.0);

   // box 2 specification
   Box2.setLength(12.0);
   Box2.setBreadth(13.0);
   Box2.setHeight(10.0);

   // volume of box 1
   volume = Box1.getVolume();
   cout << "Volume of Box1 : " << volume <<endl;

   // volume of box 2
   volume = Box2.getVolume();
   cout << "Volume of Box2 : " << volume <<endl;
```

```
   // Add two object as follows:
   Box3 = Box1 + Box2;

   // volume of box 3
   volume = Box3.getVolume();
   cout << "Volume of Box3 : " << volume <<endl;

   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Volume of Box1 : 210
Volume of Box2 : 1560
Volume of Box3 : 5400
```

Most overloaded operators may be defined as class member functions

```
Box operator+(const Box&);
```

Or as ordinary non-member functions .

In case we define above function as non-member function of a class then we would have to pass two arguments for each operand as follows:

```
Box operator+(const Box&, const Box&);
```

3.2 "friend" Functions

A regular non-member function cannot directly access the private data of the objects given in its arguments. A special type of function, called friends, are allowed to access the private data.

A "friend" function of a class, marked by the keyword friend, is a function defined outside the class, it has unrestricted access to all the class members (private, protected and public data members and member functions).

As member:

```
public:
        …….
        Box  operator+(const Box& b);
};

Box  Box::operator+(const Box& b)
{
    Box box;
    box.length = this -> length + b.length;
    box.breadth = this -> breadth + b. breadth;
    box.height = this -> height + b. height;
    return box;
}
```

You can view the code:
box3 = box1 + box2;

As written by the compiler as:
box3 = box1.operator+(box2);

As friend:

```
private:
        ……..
        friend Box operator+(Box  a, Box  b);
};

Box operator+(Box  b1, Box  b2)
{
    Box box;
    box.length = a.length + b.length;
    box.breadth = a.breadth + b. breadth;
    box.height = a.height + b. height;
    return box;
}
```

private member variables

You can view the code:
box3 = box1 + box2;

As written by the compiler as:
box3 = operator+(box1, box2);

Relational operators overloading in C++

There are various relational operators supported by C++ language like (<, >, <=, >=, ==, etc.) which can be used to compare C++ built-in data types.

You can overload any of these operators, which can be used to compare the objects of a class.

Following example explains how a < operator can be overloaded and similar way you can overload other relational operators.

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

class Distance
{
   private:
      int feet;             // 0 to infinite
      int inches;           // 0 to 12
   public:
      // required constructors
      Distance(){
         feet = 0;
         inches = 0;
      }
      Distance(int f, int i){
         feet = f;
         inches = i;
      }
      // method to display distance
      void displayDistance()
      {
         cout << "F: " << feet << " I:" << inches <<endl;
      }
```

Inline Member Functions

```
      // overloaded minus (-) operator
      Distance operator- ()
      {
         feet = -feet;
         inches = -inches;
         return Distance(feet, inches);
      }
      // overloaded < operator
      bool operator <(const Distance& d)
      {
         if(feet < d.feet)
         {
            return true;
         }
         if(feet == d.feet && inches < d.inches)
         {
            return true;
         }
         return false;
      }
};
```

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

   if( D1 < D2 )
   {
      cout << "D1 is less than D2 " << endl;
   }
   else
   {
      cout << "D2 is less than D1 " << endl;
   }
   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
D2 is less than D1
```

Input/Output operators overloading in C++

C++ is able to input and output the built-in data types using the stream extraction operator >> and the stream insertion operator <<.

The stream insertion and stream extraction operators also can be overloaded to perform input and output for user-defined types like an object.

Here, it is important to make operator overloading function a friend of the class because it would be called without creating an object.

Following example explains how extraction operator >> and insertion operator << work.

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

class Distance
{
   private:
      int feet;             // 0 to infinite
      int inches;           // 0 to 12
   public:
      // required constructors
      Distance(){
         feet = 0;
         inches = 0;
      }
      Distance(int f, int i){
         feet = f;
         inches = i;
      }
      friend ostream &operator<<( ostream &output,
                                  const Distance &D )
      {
         output << "F : " << D.feet << " I : " << D.inches;
         return output;
      }
```

```
      friend istream &operator>>( istream  &input, Distance &D )
      {
         input >> D.feet >> D.inches;
         return input;
      }
};
int main()
{
   Distance D1(11, 10), D2(5, 11), D3;

   cout << "Enter the value of object : " << endl;
   cin >> D3;
   cout << "First Distance : " << D1 << endl;
   cout << "Second Distance :" << D2 << endl;
   cout << "Third Distance :" << D3 << endl;


   return 0;
}
```

```
$./a.out
Enter the value of object :
70
10
First Distance : F : 11 I : 10
Second Distance :F : 5 I : 11
Third Distance :F : 70 I : 10
```

Overloading <<

For classes that have multiple member variables, printing each of the individual variables on the screen can get tiresome fast. For example, consider the following class:

```
class Point
{
private:
    double m_dX, m_dY, m_dZ;

public:
    Point(double dX=0.0, double dY=0.0, double dZ=0.0)
    {
    m_dX = dX;
    m_dY = dY;
    m_dZ = dZ;
    }

    double GetX() { return m_dX; }
    double GetY() { return m_dY; }
    double GetZ() { return m_dZ; }
};
```

If you wanted to print an instance of this class to the screen, you'd have to do something like this:

```
Point cPoint(5.0, 6.0, 7.0);
cout << "(" << cPoint.GetX() << ", " <<
    cPoint.GetY() << ", " <<
    cPoint.GetZ() << ")";
```

It would be much easier if you could simply type:

```
Point cPoint(5.0, 6.0, 7.0);
cout << cPoint;
```

By overloading the << operator, you can!

Overloading operator<< is similar to overloading operator+ (they are both binary operators), except that the parameter types are different.

Consider the expression cout << cPoint. If the operator is <<, what are the operands?

The left operand is the cout object, and the right operand is your Point class object.

cout is actually an object of type ostream. Therefore, our overloaded function will look like this:

```
friend ostream& operator<< (ostream &out, Point &cPoint);
```

Implementation of operator<< is fairly straightforward -- because C++ already knows how to output doubles using operator<<, and our members are all doubles, we can simply use operator<< to output the member variables of our Point.

Here is the above Point class with the overloaded operator<<.

```
class Point
{
private:
    double m_dX, m_dY, m_dZ;

public:
    Point(double dX=0.0, double dY=0.0, double dZ=0.0)
    {
    m_dX = dX;
    m_dY = dY;
    m_dZ = dZ;
    }

    friend ostream& operator<< (ostream &out, Point &cPoint);

    double GetX() { return m_dX; }
    double GetY() { return m_dY; }
    double GetZ() { return m_dZ; }
};

ostream& operator<< (ostream &out, Point &cPoint)
{
    // Since operator<< is a friend of the Point class, we can acce
ss
    // Point's members directly.
    out << "(" << cPoint.m_dX << ", " <<
        cPoint.m_dY << ", " <<
        cPoint.m_dZ << ")";
    return out;
}
```

This is pretty straightforward -- note how similar our output line is to the line we wrote when we were outputting our members manually. They are almost identical, except cout has become parameter out! The only tricky part here is the return type. Why are we returning an object of type ostream? The answer is that we do this so we can "chain" output commands together, such as cout << cPoint << endl;

Consider what would happen if our operator<< returned void. When the compiler evaluates cout << cPoint << endl;, due to the precedence/associativity rules, it evaluates this expression as (cout << cPoint) << endl;. cout << cPoint calls our void-returning overloaded operator<< function, which returns void. Then the partially evaluated expression becomes: void << endl;, which makes no sense!

By returning the out parameter as the return type instead, (cout << cPoint) returns cout. Then our partially evaluated expression becomes: cout << endl;, which then gets evaluated itself!

Any time we want our overloaded binary operators to be chainable in such a manner, the left operand should be returned.

Just to prove it works, consider the following example, which uses the Point class with the overloaded operator<< we wrote above:

```
int main()
{
    Point cPoint1(2.0, 3.0, 4.0);
    Point cPoint2(6.0, 7.0, 8.0);

    using namespace std;
    cout << cPoint1 << " " << cPoint2 << endl;

    return 0;
}
```

This produces the following result:

```
(2.0, 3.0, 4.0) (6.0, 7.0, 8.0)
```

Overloading >>

It is also possible to overload the input operator. This is done in a manner very analogous to overloading the output operator. The key thing you need to know is that cin is an object of type istream. Here's our Point class with an overloaded operator>>:

```
class Point
{
private:
    double m_dX, m_dY, m_dZ;

public:
    Point(double dX=0.0, double dY=0.0, double dZ=0.0)
    {
    m_dX = dX;
    m_dY = dY;
    m_dZ = dZ;
    }

    friend ostream& operator<< (ostream &out, Point &cPoint);
    friend istream& operator>> (istream &in, Point &cPoint);

    double GetX() { return m_dX; }
    double GetY() { return m_dY; }
    double GetZ() { return m_dZ; }
};

```

```
ostream& operator<< (ostream &out, Point &cPoint)
{
    // Since operator<< is a friend of the Point class, we can acce
ss
    // Point's members directly.
    out << "(" << cPoint.m_dX << ", " <<
        cPoint.m_dY << ", " <<
        cPoint.m_dZ << ")";
    return out;
}

istream& operator>> (istream &in, Point &cPoint)
{
    in >> cPoint.m_dX;
    in >> cPoint.m_dY;
    in >> cPoint.m_dZ;
    return in;
}
```

Here's a sample program using both the overloaded operator<< and operator>>:

```
int main()
{
    using namespace std;
    cout << "Enter a point: " << endl;

    Point cPoint;
    cin >> cPoint;

    cout << "You entered: " << cPoint << endl;

    return 0;
}
```

Assuming the user enters `3.0 4.5 7.26` as input, the program produces the following result:

```
You entered: (3, 4.5, 7.26)
```

Following is the list of operators, which can not be overloaded:

::	.*	.	?: