Operator Overloading in C++

In C++, we can make operators work for user-defined classes. This means C++ has the ability to provide the operators with a special meaning for a data type, this ability is known as operator overloading.

For example, we can overload an operator '+' in a class like String so that we can concatenate two strings by just using +.

Other example classes where arithmetic operators may be overloaded are Complex Number, Fractional Number, Big Integer, etc. An example for the case of complex numbers addition:

```
#include <bits/stdc++.h>
using namespace std;

class Complex {
    private:
        int real, imag;
    public:
        Complex(int r = 0, int i =0)  {real = r;   imag = i;}

        // This is automatically called when '+' is used with
        // between two Complex objects
        Complex operator + (Complex const &obj) {
             Complex res;
             res.real = real + obj.real;
             res.imag = imag + obj.imag;
             return res;
        }
        void print() { cout << real << " + i" << imag << endl
};

int main()
{
    Complex c1(10, 5), c2(2, 4);
    Complex c3 = c1 + c2; // An example call to "operator+"
```

```
    c3.print();
}
```

Output:

```
12 + i9
```

What is the difference between operator functions and normal functions?

Operator functions are the same as normal functions. The only difference is that the name of an operator function is always the operator keyword followed by the symbol of operator and operator functions are called when the corresponding operator is used.

Following is an example of global operator function.

```
#include <bits/stdc++.h>
using namespace std;

class Complex
{
    private:
        int real, imag;
    public:
        Complex(int r = 0, int i =0)  {real = r;   imag = i;}
        void print() { cout << real << " + i" << imag << endl

        // The global operator function is made friend of thi
        // that it can access private members
        friend Complex operator + (Complex const &, Complex c
};

Complex operator + (Complex const &c1, Complex const &c2)
{
    return Complex(c1.real + c2.real, c1.imag + c2.imag);
}


int main()
{
```

```
    Complex c1(10, 5), c2(2, 4);
    Complex c3 = c1 + c2; // An example call to "operator+"
    c3.print();
    return 0;
}
```

Output:

```
12 + i9
```

Can we overload all operators?

Almost all operators can be overloaded except few. Following is the list of operators that cannot be overloaded.

- **.** (dot)
- **::** (scope-resolution)
- **?:** (ternary-operator)
- **sizeof**

Important points about operator overloading

1. For operator overloading to work, at least one of the operands must be an user-defined class object.
2. *Assignment Operator* - Compiler automatically creates a default assignment operator with every class. The default assignment operator does assign all members of right side to the left side and works fine most of the cases (this behavior is same as copy constructor).
3. *Conversion Operator* - We can also write conversion operators that can be used to convert one type to another type. As an example:

```
#include <bits/stdc++.h>
using namespace std;

class Fraction
{
    int num, den;
```

```
    public:
        Fraction(int n,  int d) : num(n), den(d) {}

        // conversion operator: return float value of frac
        operator float() const {
            return float(num) / float(den);
        }
};

int main()
{
    Fraction f(2,5);
    float val = f;
    cout << val;
    return 0;
}
```

Output:

```
0.4
```

4. Overloaded conversion operators must be a member method. Other operators can either be a member method or a global method.
5. Any constructor that can be called with a single argument works as a conversion constructor, which means it can also be used for implicit conversion to the class being constructed.

```
#include <bits/stdc++.h>
using namespace std;

class Point
{
    private:
        int x, y;
    public:
        Point(int i=0, int j=0) : x(i), y(j) {}
        void print() {
            cout << endl << " x = " << x << ", y = " << y;
```

```
        }
};

int main() {
    Point t(20, 20);
    t.print();
    t = 30;   // Member x of t becomes 30
    t.print();
    return 0;
}
```

Output:

```
 x = 20, y = 20
x = 30, y = 0
```