

## Operator Overloading

“overloading” means that we can give different meaning (implementation) to common operators. C++ defines lots of operators, most of which can be overloaded to work with user-defined class types (e.g., Complex).

Binary arithmetic operators are either global **friend** functions (two args) or class member functions (one arg). Both do the same thing, but it is often the case that overloaded operators are implemented as friend functions.

```
class T {
public:
    ...
    friend T& operator+ (const T&, const T&); // or T& operator+ (const T&)
    friend T& operator- (const T&, const T&); // or T& operator- (const T&)
    friend T& operator* (const T&, const T&); // or T& operator* (const T&)
    friend T& operator/ (const T&, const T&); // or T& operator/ (const T&)
    friend T& operator% (const T&, const T&); // or T& operator% (const T&)
    ...
};

Complex a, b(1.0, 1.0), c(2.0); // a is (0.0,0.0), b is (1.0,1.0), c is (2.0,1.0)

a = b + c; // Complex::operator+(b, c) or b.operator+(c)
a = b - c; // Complex::operator-(b, c) or b.operator-(c)
a = b * c; // Complex::operator*(b,c) or b.operator*(c)
a = b / c; // Complex::operator/(b,c) or b.operator/(c)

a = b + 1.0; // Complex::operator(b, ?) or b.operator+(?)
a = b + Complex(1.0, 0.0);
```

## Operator Overloading

Relational operators:

```
bool operator==(const T&, const T&);
bool operator!=(const T&, const T&);
bool operator<(const T&, const T&);
bool operator>(const T&, const T&);
bool operator<=(const T&, const T&);
bool operator>=(const T&, const T&);
```

What minimal set of the relational operators do you need to implement for any type T, such as the Complex type?

### HOMEWORK ASSIGNMENT:

Assuming that Complex is defined as:

```
class Complex {
private:
    double real, imag;
public:
    Complex(double r = 0.0, double i = 0.0) : real(r), imag(i);
    ...
};
```

Implement each of the relational operators for the Complex class.

## Operator Overloading

Pre/Post increment and decrement operators:

```
T& operator++(); // prefix increment
T& operator++(int); // postfix increment
T& operator--(); // prefix decrement
T& operator--(int); // postfix decrement
```

```
Complex x;
x++; // x.operator++(int)
++x; // x.operator++()
x--; // x.operator--(int)
--x; // x.operator--()
```

Assignment operators:

```
T& operator= (const T&);
T& operator+= (const T&);
T& operator-= (const T&);
T& operator*= (const T&);
T& operator/= (const T&);
```

Complex x(1.0, 1.0), y(2.0);

```
x = y; // x.operator=(y)
x += Complex(2.0, 2.0); // x.operator+=(Complex(2.0, 2.0))
y -= Complex(1.0); // y.operator-=(Complex(1.0, 0.0))
...
```

## Operator new and delete (again)

The global new and delete operators apply to all class types that do not overload new and delete. However, new and delete can be overloaded on a class by class basis to provide customized allocation and deallocation of objects.

```
class String {
public:
    void* operator new(unsigned long size) { /* your own memory allocator */ }
    void operator delete(void* p){ /* your own memory deallocator */ }
    ...
};
```

```
String *s = new String("hello, world");
```

Usually, the compiler translates this single statement into the equivalent of the following two statements:

```
String* s = ::operator new(sizeof(String)); // allocate storage using global new
String::String(s, "hello, world"); // call String(const char*) constructor
```

If you overload operator new for a specific class, you get the following instead:

```
String *s = String::operator new(sizeof(String));
String::String(s, "hello, world");
```

**Note that compiler arranges to pass the String\* pointer 's' as the first argument implicitly? In C++, the address of any object is called its “this” pointer. Every object has a “this” pointer, and every member function is passed the “this” pointer implicitly as the first argument of the function.**