# CSCI 262
# Data Structures

16 – Operator Overloading

CS@Mines

## Motivation

Consider a new class for a number-like object:

```
class complex {
public:
      double real;
      double imaginary;
      complex(double r, double im);
};
```

We'd want to add, subtract, etc. complex numbers.

CS@Mines

## Without Operators

We could create functions for this job:

```
complex add(complex a, complex b)
complex subtract(complex a, complex b)
```

Etc.

```
complex x(0.0, 4.5);       // 4.5i
complex y(1.0, -7.0);      // 1 – 7i
complex z = add(x, y);     // 1 – 2.5i
```

CS@Mines

## With Operators

C++ let us do the following (with a little work):

```
complex x(0.0, 4.5);  // 4.5i
complex y(1.0, -7.0); // 1 – 7i
complex z = x + y;    // 1 – 2.5i
```

And even this (chaining):

```
complex c = x + z - y;
```

CS@Mines

## Operator Overloading

Here's how:

```
complex operator+(const complex &a, const complex &b) {
     return complex(
           a.real + b.real,
           a.imaginary + b.imaginary
     );
}
```

CS@Mines

## Member vs. Non-Member

- We wrote the previous method as a *non-member* function.
- You can do it the other way, too:

```
complex complex::operator +(const complex &b) const {
     return complex(
           real + b.real,
           imaginary + b.imaginary
     );
}
```

- Declare in the class declaration like any other method.
- Note, only one argument – for the RHS (Right Hand Side) argument.

CS@Mines

## Member vs. Non-Member

- Some operators (e.g., assignment) **must** be member functions
- Some operators (e.g., <<, >>) **cannot** be member functions
- Non-member operator functions may have to be declared as *friend* functions for private access
- Most binary operators can be either
  - Which you use partly a matter of style
  - For now, recommend using non-member functions



## Mixed Type

Suppose we want to add complex and real numbers:

```
complex a(1.0, 3.1); // 1 + 3.1i
double x = 0.5;
complex z = a + x;   // 1.5 + 3.1i
```



## More Overloading

To support mixed type operations, we just add more overloaded functions:

```
complex operator +(const complex &a, const complex &b);

complex operator +(const complex &a, const double &b);

complex operator +(const double &a, const complex &b);
```

Note this last one cannot be a member function!



## Stream Output

As another example consider the following:

```
complex c(4.0, 0.5);
cout << c << endl;  // error!
```

To make this work, we need to tell C++ how to print complex values on ostreams:

```
ostream& operator<<(ostream &out, const complex &c) {
    out << c.real << " + " << c.imaginary << 'i';
    return out;
}
```

Note: cannot be a member function of complex!



## Other Operators

- In all, C++ lets you overload some 50 different operators!
- Some you've seen:
  - <<, >> for stream operations (originally used for bitwise shift operations)
  - + for string concatenation (and complex addition)
  - [] for string and vector element access
  - = for assignment
- Besides these, a common set are the Boolean comparison operators: <, >, <=, >=, and ==.



## Rules

*These are not everyone's rules. But they're mine.*

I. Mostly, don't. Use clearly named member functions or static member functions instead.

II. If you must, then:
   a. Be consistent – use symbols that mimic their original use (or existing practice, as in + for concatenation)
   b. Be complete – if you overload one of a set, overload them all - e.g., <, >, <=, >=, and ==, not just <.

III. Exceptions to the above:
   a. = part of the "big 3"
   b. () used for "function objects"

# Up Next

- Reading for April 1: Chapter 13.2
- Friday, March 22
  - Lab 10 – Operator Overloading
  - APT 4 Due
  - Project 4 assigned
- Next week: S P R I N G  B R E A K!
- Monday, April 1
  - The Big 3
  - Lab 10 due

CS@Mines