Motivation

Consider a new class for a number-like object:

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

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

Without Operators

We could create functions for this job:

```cpp```
complex add(complex a, complex b)
complex subtract(complex a, complex b)
Etc.
```

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

With Operators

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

```cpp```
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):

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

Operator Overloading

Here's how:

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

Member vs. Non-Member

- We wrote the previous method as a non-member function.
- You can do it the other way, too:
  ```cpp```
  complex complex::operator +(const complex &b) const {
      return complex(
          real + b.real,
          imaginary + b.imaginary
      );
  }
  ```cpp```
- Declare in the class declaration like any other method.
- Note, only one argument – for the RHS argument.
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:

\[
\text{complex } a(1.0, 3.1); \quad // \quad 1 + 3.1i \\
\text{double } x = 0.5; \\
\text{complex } z = a + x; \quad // \quad 1.5 + 3.1i
\]

More Overloading

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

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

Stream Output

As another example consider the following:

\[
\text{complex } c(4.0, 0.5); \\
\text{cout } << c << endl; \quad // \text{error!}
\]

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

```cpp
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 44 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 <.