Operator Overloading

Today’s topic is operator overloading. Let me start with a slightly different topic, however, one that is not in your text: testing, and in particular unit testing. Several of your peers have to come to me with a similar question. It goes something like this: I think my program is good, but then I loose points on a test case I didn’t anticipate. How can I improve this?

There are two things going on here. One is a “rush to program”. You get an assignment, and immediately start working on it. Sounds good, but before you start to solve a problem you need to define it. In CS, that means think of the boundary cases. And you can’t think do this effectively after you have solved the problem, because then you are thinking in terms of the solution you have already written. You will do much better if you write down the test cases first, and then develop your program, not the other way around.

Really. Try it. It will help.

There is also something else going on. Your programs are getting more complex. They are now bigger than most of the assignments you have written in previous classes. And they will get bigger, significantly bigger, before the end of the semester.

Note that in practical terms, they are still small. When you get to the 400 level classes, you will be expected to produce programs that are an order of magnitude bigger. And most industrial programs are two or three orders of magnitude bigger (and written in teams, of course).

As your program becomes more complex, more things can go wrong, and more things need to be tested. Worse still, the boundary cases interact. Just think of the example of abysmal… The number of cases you have to test for increases exponentially with the complexity of the program.

What do you do? It’s not coincidental that we are starting to look at OO programming again. You know we are going to encourage you to encapsulate related code into modular objects to manage complexity. That is what OO programming is about.

But right now I am talking about object oriented testing. We call it unit testing. Not only should you break up your code into independent objects, you should write an independent test program for every single object. In fact, you should write a test for every method of every object. Think about the boundary cases of every single method. Test them. If every method is right, your overall program will (generally) be pretty good.

People complain that this means writing a lot of test code. Yes it does: it also takes a lot of scaffolding to build a tall building. It’s worth it, if it means the building won’t fall down when you are done. It’s the same with programming.

Trust me, with complex programs is to faster to write all the test programs and debug the little pieces, then to skip the unit tests and then debug the resulting large program.

And we have tools that can help. This is why we are teaching you GoogleTest in the recitations. It is a free tool for helping you write unit test programs. It works. Trust me, I use it.
OK, let’s start talking about object oriented programming, and let’s start with a brief review: what does “polymorphism” mean? (in Java). Why do you use it, and how do you use it? Can you give me an example?

OK, let’s talk about one of the simplest (and most popular) features of C++ not shared by Java: operator overloading. The basic idea is as simple as its name implies: C++ allows you to overload the definition of operators in order to define how they operate on non-primitive data types.

Now think about this for a second: Java supports polymorphism, but only for classes. We can redefine the move method of a class, but we can’t use polymorphism if some of the options are primitives. We just can’t define methods for primitives.

Think about the complex number class we defined in the second week. We would like to treat it like any other number. But primitive numbers support operations like “+” and “-”. Making complex use named methods like “Add()” is non-symmetric. It breaks polymorphism.

So the idea of operator overloading is to support polymorphism across classes and primitives. Operators are defined for primitives, so they should be defined for classes, too.

In fact, operators are overloaded already (in both C++ and Java) for various types of primitives. Is ‘+’ defined for integers? Or doubles? (answer: yes). How about the input and output operators?

You have already seen an example of operator overloading. Go back to the Complex number class I gave you near the beginning of the semester, and look at the Complex class definition in the header file. It looks like:

class Complex {
public:
    Complex(double real_part = 0.0, double imaginary_part = 0.0)
        : real(real_part), imaginary(imaginary_part) {} 
    inline double Real() const {return real;} 
    inline double& Real() {return real;} 
    inline double Imaginary() const {return imaginary;} 
    inline double& Imaginary() {return imaginary;} 
    void Add(const Complex& add_in);
    void Subtract(const Complex& sub_in);
    void Multiply(const Complex& mult_in);
    bool Divide(const Complex& div_in);
    Complex operator + (const Complex& in) const;
    Complex operator – (const Complex& in) const;
    Complex operator * (const Complex& in) const;
    Complex operator / (const Complex& in) const;

protected:
    ... 
};

Look at the declaration of operator +. This is an overload of the plus operator. It is followed by overloads of -, *, and /. What do the implementations look like? Go back to the Complex.cpp file:
Complex Complex::operator +
    (const Complex& in) const
{
    Complex out(Real() + in.Real(),
                Imaginary() + in.Imaginary);
    return out;
}

This implements the plus operator for adding two Complex numbers. Note that mathematics plus
is both symmetric (A+B) = (B+A) and non-side-effecting: C = A+B does not alter A or B. The
same is true of how we overloaded plus for the Complex object class.

OK, so that was an example. Now let’s talk a little more generally. What operators can you
overload?

- Math: + - * / == < > *= - /= += %=
- I/O << >>
- Logic: & | ! ~ <<>> <<= >>= ^ ^= && ||
- Indexing: () [] ++ -- > * new delete new[] delete[]

There are a few you cannot overload (. .* ? sizeof)

Should operators be methods or stand alone functions? Your choice. The way I defined ‘+’ in our
complex number example, it was a method. In this case, the compiler translates “a + b” (if a and b
are instances of complex) into a.operator+(b).

Had I defined complex ‘+’ as a stand-alone function (i.e. outside of the complex class), then it
would have translated it into operator+(a, b).

Syntactically, if ‘+’ is a function, it takes two arguments. If it is a method, it only takes one (why?
Because of the hidden argument…)

This is a general rule: remember the hidden argument. So ++ as a method takes no arguments, but
as a function it takes 1. Make sense?

OK, so you have a choice. When to make an overloaded operator a method, and when to make it
a function. Sometimes either answer is OK, but there are cases where a method is called for:

- When the operator needs to access private data
- When the operator alters one argument (+ is usually a method). Note: you can define the
  operator as taking in a reference and alter an explicit argument. But not good style

I tend to use functions when (a) the operator is symmetric (e.g. add two complex numbers), (b)
the data is public, and (c) there are no side-effects (the arguments aren’t altered). Seems better to
me somehow (more like true ‘+’). But some people make all operators methods. It’s a matter of
taste.

Some limitations: you can’t change operator precedence. If you write a*b+c, * takes precedence
over +, even if they are overloaded. Operator overloading doesn’t change how your program is
parsed.
You can’t define a syntactically new operator. You cannot, for example, overload **. It isn’t an operator.

You cannot change the number of arguments. ++ takes only one argument (usually hidden), the data being incremented. You cannot define a two argument version such that a ++ b is legal.

The instructions above explain how to overload operators. The more important question is when to overload them. The short answer is that you should overload an operator when you would otherwise have to write a method that exactly mimics what expect of an operator. The Complex number class is an easy example. It’s a number; we know how operators should be defined for it. We know how to add, subtract, multiple and divide numbers, and what the semantics of those operators should be (including symmetry, no side-effects, etc.). So go ahead and overload them.

The case for overloading mathematical operators for mathematical classes is straightforward, but also kind of rare. How often do you define a mathematical class? When else should you overload operators?

Two rules of thumb:
1. When a method exactly matches the expectations that an average programmer would have for an operator, overload the operator.
2. If there is any doubt about point #1 above, don’t overload the operator. Overloading an operator when the semantics are not obvious creates confusion. I would rather have you miss an opportunity to overload an operator than to overload one when you shouldn’t.

Some common situations where overloading an operator is the right thing to do:
1. I/O. The most common operators to overload are << and >>. The read or write objects. Even big objects, like complete data bases.
2. Functors. If you overload the () operator, the class will act like a Function. Think about it: if I overload the () operator for class Foo, and f is a Foo, then I can call f(). This is how we avoid passing pointers to functions in C++. We make classes that are functors instead, i.e. classes with () overloaded, and then pass functor objects. More about this when we get to templates.
3. Numerics (like Complex).
4. Sometimes logic operators. For example, the stream classes overload !. But this only works when the semantics of object are binary, for example a stream is an a fail state or it isn’t.

When shouldn’t you overload an operator? Think of the following code:

```cpp
Company c;
Person p;
cout << c + p;
```

What the heck is going on? Did company c hire person p? Did person p buy company c? Are we combing the assets of person p and company c? This is totally confusing, and an example of the misuse of operator overloading.

Also, think about reflexive and symmetric properties. Never overload a reflexive or symmetric operator with a non-reflexive or non-symmetric method.