IB9N7 C++ for Quantitative Finance

Lecture 10: Operator overloading

J.F.Reizenstein@warwick.ac.uk

21 January 2015
(Week 16)

Operator overloading

Recall the available operators in C++ (e.g. from lecture 3):

- Different types of operators: arithmetic (+), logical (!), relational (>), etc.
- We only applied these to basic datatypes — what if we want to be able to use them with our own classes?

Operator overloading

Operator overloading is the (re)definition of standard operators acting in non-standard ways and/or on non-standard datatypes.

Operators vs. functions

Motivation: To add two numbers, operators provide a syntactic convenience.

You write

```
a + b
```

instead of

```
add_two_numbers(a, b)
```

(say) where `add_two_numbers` has overloads for all the fundamental arithmetic types.

Operations on classes

What if you want to add two `Point`s, or add a number of days to a `Date`, etc?

You can write `add_two_Points` (or just `add`) as a free function taking two `Point` arguments.

You can write `add` as a member function of `Point` taking one `Point` argument.

You can overload the + operator. (Better!)

The latter solution is clearly desirable if you want to add more than two `Point`s, allowing to avoid the `add(add(x, y), z)` notation, and instead write `x + y + z` and has the usual benefits:

- less to type
- less to remember
- less to go wrong

Warning about semantics

The semantics of the operators are not dictated by the standard.

It is not necessarily true that `a != b` and `!(a == b)` will behave in the same way or evaluate to the same values.

- It is not even required that `!=` and `==` evaluate to the same data types (which need not even be boolean).
- You could (though don't!) overload `!=` to mean addition, for example.
- You could also (though also don't!) overload it to write "Hello world" to the console, ignoring the operands.

Operator overloading best practice

It is good practice to overload operators in combinations that the users of your class will expect, and for them to behave in ways that users would expect.

- If you define `==`, C++ will not define a compatible `!=` automatically, so remember to do it yourself (can define one in terms of the other).

You should not overload operators to behave in unusual ways, unless there is a very good reason for doing so.

- For `cin` and `cout`, the bit-shift operators `>>` and `<<` are overloaded, but the semantics of these are stream extraction and stream insertion respectively.
- This is fine because bit-shifting would not be a usual operation to perform in these contexts.

How to overload an operator

Operator overloading

Overloading an operator is achieved by defining an **operator function**, a function with a special name given by the keyword `operator` followed by the symbol denoting the operator.

There are two ways of defining operator functions, either as class members or free functions. There are many important differences between the two approaches, the first being the syntax:

- In the former, the first operand is always taken as `this` (recall that (non-static) member functions are executed in the context of a particular object instance).
- In the latter, the first operand is an argument to the function.

Operator overloading syntax

Overloaded operators (but not the built-in operators) can be called using the function notation. e.g.:

```
1 Point a, b;
2 Point c = operator+(a, b); // free functional form
3 Point d = a.operator+(b); // member functional form
4 Point e = a + b; // operator form
```

Note: shouldn't define both

```
Point operator+(const Point& firstOperand, const Point
    & secondOperand); // free function
Point Point::operator+(const Point& secondOperand)
    const; // member function
```

because the standard says they are ambiguous.
Can therefore use either syntax on line 2 or on line 3, if defined relevant function, but not both.

Member or free function?

Given that there are two choices for implementing operator overloads, and that you cannot implement both, which should you choose?

Some operator overloads that you might want to provide cannot be defined as member functions (see examples on slides 25, 27 later).

Sometimes defining non-member functions means implicit conversion can happen where it could not with a member function (see example on slide 31 later).

For now, if you need access to the private data, declare as a member function.
If you don't need access to private data, prefer a non-member function.

What operators can/should be overloaded?

What operators *can* be overloaded?

- You can overload *almost every* operator, although there are a few exceptions.
- Eg, you cannot overload the scope operator `::`, `throw` operator, dot operator `.` or ternary operator `?:`.

What operators *should* be overloaded for a class?

- This depends entirely on the purpose of the class.
- Overload exactly those operators that make sense for the class.
- You should only overload operators to behave in an intuitive way!

Note that operator overloading does not affect precedence rules, nor number of operands!

Const

Many operators will not modify all/any of their operands (e.g. = will usually change the LHS only, but == and + usually won't change either of their operands).

- Usual pass by value vs. pass by reference semantics, with usual conventions.
- All should declare their invariant operands (of non-fundamental types) as `const` references.
- In the case of member operator functions, the operator member functions themselves should be declared `const` too where applicable (recall: essentially same as declaring hidden `this` parameter to be `const`).

Equality testing

The equality operator (==) is (usually) used for testing if two instances of a class are equal (in a sense to be decided by the programmer).

If we do not overload the equality operator for our `Point` class, then we cannot compare two `Point`s using the equality operator.

i.e. C++ does not implicitly generate any equality operator for us.

If `a` and `b` are `Point`s and there is no equality operator, then attempting to use `a == b` will result in the following compiler error:

```
[Error] no match for 'operator==' in 'a == b'
```

Testing if two `Point`s are equal (1)

Example: Equality operator for `Point`

```
1  // Point.hpp
2  class Point
3  {
4    bool operator==(const Point& that) const; // member
         syntax
5    // ...
6  };
7  // Point.cpp
8  bool Point::operator==(const Point& that) const
9  {
10   return (X_ == that.X_ && Y_ == that.Y_);
11 }
```


Testing if two `Point`s are equal (2)

Example: Equality operator for `Point`

```
1  // Point.hpp
2  class Point
3  {
4    // ...
5  };
6  bool operator==(const Point& a, const Point& b); //
       non-member syntax
7  // Point.cpp
8  bool operator==(const Point& a, const Point& b)
9  {
10   return (a.X_ == b.X_ && a.Y_ == b.Y_);
11 }
```


Testing if two `Point`s are equal (3)

- With either form, we can now do things like:

```
1  Point a, b;
2  // ...
3  if (a == b) {...};
```

- When evaluating the condition of the `if` statement, the compiler translates the expression `(a == b)` into

```
a.operator== ( b )
```

  (or `operator==(a, b)` for the non-member form) the return value of which is a `bool`.
- `a == b` is said to be an implicit call to `a.operator==`, while writing `a.operator==` directly constitutes an explicit call. We normally use the implicit syntax, unless we have a good reason to do otherwise.

Time-saving with comparison operators

In many classes, we can implement a dozen or more operators.

- Arithmetic-oriented classes are especially prone to this: `+ - / *` and also `+= -= /= *= ++ --`
- Plus a load of comparisons: `== != < > <= >=`

These basically duplicate the same core operations.

- Therefore, we can take advantage of this without excessively duplicating the code performing the core operations.
- The idea is to define some of the operations in terms of the others, as exemplified on the next slide.
- At the same time, this should make it more likely that we have implemented the semantics correctly (without unintended inconsistencies).

Comparison operator shortcuts

Example: Operator shortcuts

```
1  bool Point::operator!=(const Point& p) const
2  {
3      return !(operator==(p));
4  }
5  bool Point::operator<=(const Point& p) const
6  {
7      return operator<(p) || operator==(p); // assuming we
           have also implemented operator< directly
8  }
```


Operator return-values

The result of applying different operators will quite clearly have different results. Some easy and obvious examples:

- Comparison operators usually return `true` or `false`.
- Arithmetic operators usually return an object of the same class-type **by value**.

There are some circumstances which are not so obvious:

- The subscript operator (a.k.a. array indexing operator) `[]` and parentheses/function call operator `()`. Useful for creating classes representing vectors and matrices. May return arbitrary types.
- Operators involving *assignment* should usually return a reference to the object acted on (via the `this` keyword), see examples later. (Allows doing `a = b = c`, etc.)

Operator return-values: example

Example: `Point` addition

```
1  // Point.hpp
2  class Point
3  { // ...
4      Point operator+(const Point&) const;
5  };
6  // Point.cpp
7  Point Point::operator+(const Point& secondOperand)
       const
8  {
9      Point ret; // could use a parameterised constructor instead
10     ret.X_ = X_ + secondOperand.X_;
11     ret.Y_ = Y_ + secondOperand.Y_;
12     return ret;
13 }
```


Operator return-values (2)

After providing that function, we can then do stuff like:

```
1  Point a(1,1), b(2,2), c;
2  c = a + b;
```

This leads to some important questions about the assignment (=) operator.

Overloading assignment

Suppose you write `b = a;` (not necessarily `Point`s)

For a given `b`, what are the valid types of `a` such that this makes sense?

A natural case might be where `a` and `b` are of the same type (though this is not always valid!).

Assigning an rvalue to an lvalue of the same type is known as **copy assignment**. eg

```
1 int a = 5;
2 int b = 0;
3 b = a; // copy assignment
```

Here, `a` and `b` are both of the same type (`int`), so the type of assignment being done is called a copy assignment.

Other types of assignment

Assignments where the lvalue and rvalue are of distinct types are considered *conversion assignments*. eg

```
1 int a = 5;
2 double b = 0.0;
3 b = a; // conversion assignment
```

For fundamental types, the assignment operators behave (largely) as you would expect.

- In the above example, the value of `b` after the assignment is the value nearest to `a` that can be stored in `double` precision.
- As with the other operators, the semantics of overloading the assignment operator are not enforced.

For general types, depends if there is an (implicitly accessible) conversion constructor .

Implicitly generated operator overloads

If you do not declare any overloads of the assignment operator (=) for a given class, C++ tries to implicitly define two for you. This will fail e.g. if a member of a class is a reference.

- The copy assignment operator simply does a copy of all member variables (shallow copy).
  `operator=(const T& t)`
- The move assignment operator (`operator=(T&& t)`) is similar. It is called only in cases where the right hand side is a temporary object.
- Have to be careful with such operators if the class deals with raw resources, but this is usually not a good thing to do.

These, together with the other member functions that C++ will implicitly generate, are called **special member**

Copy assignment

Example: copy assignment in `Point`

```
1  // Point.hpp
2  class Point
3  {
4      // ...
5      Point& operator=(const Point& that);
6  };
7  // Point.cpp
8  Point& Point::operator=(const Point& that)
9  { // roughly equivalent to the implicitly generated one for this example
10     X_ = that.X_;
11     Y_ = that.Y_;
12     return *this;
13 }
```

Again, declare operand as `const` as we should not modify the RHS. Clearly the function itself cannot be `const`.

Stream insertion (1)

One can also overload the stream insertion operator, and this can be very convenient.

- << is overloaded in the stream classes for all primitive data types, and is easy to overload for user defined types.
- Stream operators should return the same stream as is passed in the argument, to allow combining multiple << operators into one expression.
- For this, we need to know that the object instance **std::cout** is of class type **std::ostream**.

Here, the class to which the member function "should" be added (**std::ostream**) is not available for modification, as it is one of the standard library classes.

Therefore, this has to be done as a non-member function.

Stream insertion (2)

Example: overloading stream insertion

```
1  // Point.hpp
2  #include <ostream> // or iostream, but only using output streams here
3  class Point
4  {
5      // ...
6  };
7  std::ostream& operator<< (std::ostream &out, const Point &p)
       ; // note the stream cannot be 'const' as we will be writing to it and thus it
       might change state
8
9  // Point.cpp
10 std::ostream& operator<< (std::ostream &out, const Point &p)
11 {
12     out << "(" << p.get_x() << ", " << p.get_y() << ")";
13     return out;
14 }
```


When is symmetric not symmetric? ☆

If we had a class **Complex** and defined the member function:

```
1  bool operator==(double that);
```

then we can do:

```
2  Complex a;
3  if (a == 5.0) {...};
```

But, even though it makes sense, the above doesn't allow us to do

```
4  if (5.0 == a) {...};
```

For binary operators, one limitation of the member function form of operator overloading is that the left object must be of the class type.

Different type first operators ☆

You can permit the previous example to work by defining an extra operator as a non-member function, with the operands in the opposite order. (Or, if convertible, will work as non-member function directly.)

Example: non-member operators

```
1  // Complex.hpp
2  class Complex
3  {
4      // ...
5      bool operator==(double that) const; // Complex == double
6  };
7  bool operator==(double lhs, const Complex& rhs); // double
       == Complex
8  // ...
```

Different type first operators (2) ☆

Example: non-member operators

```
9  // Complex.cpp
10 // ...(define the member operator as usual)...
11 // and now define the non-member operator:
12 bool operator==(double lhs, const Complex& rhs)
13 {
14     return (rhs == lhs); // reverse the two operands, and so
       use the previously defined member function
15 }
```

Then we can do

```
16 if (5.0 == a) {...};
```


Different type first operators (3) ☆

- When the compiler sees `(5.0 == a)`, it translates it to

```
17 operator==(5.0, a)
```

- This in turn uses our previously defined member function: `a.operator==(5.0)`

Non-member functions can be preferable ☆

Example: non-member preferred when type conversion relevant

1. Say we have a class `Complex`;
2. Suppose it can be implicitly converted from `double` (but not to `double`);
3. Suppose we have an `operator+(const & Complex, const & Complex);`
4. We want to add a `Complex` with a `double`.

If `a` and `b` are declared by

```
Complex a; double b;
```

then we want `a + b` and `b + a` to both work without having to define extra overloads.

If the `operator+` was instead a member function of `Complex`, then the second won't work. But as a free function, they both work.

Operator syntaxes ☆

If `x` is an object type then:

- `x(a, b, c)` is equivalent to `x.operator()(a, b, c)`,
- `x[a]` is equivalent to `x.operator[](a)`,
- `x->a` is equivalent to `x.operator->()->a`.

Operator special cases ☆

If you want to overload `++` or `--`:

- The prefix increment and decrement operators can be overloaded with `operator++()`, or as a non-member function with one argument.
- The postfix increment and decrement operators can be overloaded with `operator++(int)` where the argument is not of interest, or as a non-member function with two arguments.
  The additional argument is just a dummy parameter to ensure that the two signatures are different.

What do you need to know?

Operator overloading is a difficult topic.

- Using overloaded operators is relatively easy.
- Defining them can be complicated.

What you should take away:

- An appreciation of the fact that when you use operators, overloading may be going on behind the scenes.
- It is possible to enhance the interactions possible with objects by defining operator overloads.
- Being able to provide stream insertion overloads and simple arithmetic/relational operator overloads is sufficient for this course.

Lab objectives

- Understand and experiment with operator overloading