Benjamin Fenster
CS-100 (Damon)
3 May 2003

Notes – Operator Overloading

- I. Function Overloading
  - a. Can have similar functions with the same name, but different arguments.
  - b. The compiler picks one at compile time based on the declared types of arguments.
  - c. Everything else we've seen is based on runtime types, but this is not.
- II. Operator Overloading
  - a. Arithmetic operators are almost always overloaded for ints, floats, etc.
  - b. Java also overloads + for strings, and |, ||, &, and && for int and Boolean.
  - c. All this seems pretty natural.
  - d. Languages like C++ allow programmers to define operators too.
    - i. Makes the programmer's code look just like built-in code.
    - ii. In C++ can define a global function that takes one or two arguments (depending on the nature of the operator).
    - iii. Can also define a member function on a class, with one fewer argument (the first will always be an instance of the class.
  - e. Finding the right function or method to interpret a unary operator is pretty easy.
  - f. How does C++ find the right function to interpret a binary operator?
    - i. Look at the left-hand operand
      - 1. Try to resolve as an operator member function.
      - 2. If it's not an object skip this, and move on. If there is no function, move on.
    - ii. Look for an appropriate global function.
  - g. There's no assumption that any operator is commutative
    - i. Need every possible permutation to make the code easy to write.
    - ii. Makes for a LOT of code!
  - h. C++ allows almost every operator to be overloaded
    - i. We discussed `new` and `delete` already.
    - ii. Can define globally or in a class.
    - iii. Define both or none – don't define one and assume the default implementation of the other will suffice.
  - i. The tricky ones are those using reference types. (Remember that reference types are just pointers that get dereferenced automatically).
- III. Specific Operators
  - a. ++, --
    - i. One for pre, one for post.
    - ii. operator++(int) is post
    - iii. operator++() is pre
    - iv. The post-fix version doesn't actually use its integer argument – it's just there to indicate which function is which.
  - b. Subscripting, operator[]
    - i. Makes list access look like array access
    - ii. For getting, return a reference to where the data is stored (dangerous!)
    - iii. For setting, you still have to return such a reference so you may need to create some space first and return a reference to that.
  - c. Can overload function calls
    - i. Make an object and "call it"
    - ii. Weird
  - d. Surprising: The `->` dereference operator, along with *
    - i. Allows "smart pointers"
    - ii. Means you can make a "pointer" that doesn't really refer to memory.
    - iii. May really access stuff off a disk or something else, but will look and act like a pointer.
    - iv. *Almost* allows persistent objects, save a few quirks.
  - e. Casting

    - i. Casting to a built-in type is simple.
    - ii. Casting to another object is quite different.
    - iii. Converts cast to constructor calls.
    - iv. C++ tries to cast wherever it’s appropriate (3 to double, for example, in double x = 3;)
    - v. Makes nice code in terms of simplicity.
    - vi. Makes it even harder to find the code that’s executing.
    - vii. More time is spent reading code than writing it.
- IV. The Rub
  - a. There are many cases where this is great, but others where it’s horrible.
  - b. Three problems.
  - c. Need to get all the operators right. That’s not too bad.
  - d. Need to get every permutation right.
    - i. This is really hard.
    - ii. Adding a new class that can interact with an existing class means you need to add tons of overloaded operator functions just to get the two to work together.
  - e. Readability
    - i. It’s hard for the compiler to find the right function to use.
    - ii. It’s even harder for a human to find it.
    - iii. It’s *extremely* hard to understand code using overloaded operators when you need to know where the supporting code is.
    - iv. You can’t even guarantee that the operator does what’s expected.
    - v. The `cout << value;` command *looks* pretty, but has nothing to do with left shift!
    - vi. Someone thought that was a good idea, so someone else may similarly create something weird.
  - f. Use it Sparingly
    - i. `x + y` isn’t that much better than `x.plus(y);`
    - ii. Keep in mind the future maintenance you’ll have to do.
  - g. Sometimes overloading operators is cheaper than allowing the automatic conversion when you write Fraction f2 = f1 * 3.
    - i. Would first convert 3 to a Fraction
    - ii. That’s expensive. VERY expensive (takes twice the time)
    - iii. Better in that case to overload + and get an efficiency boost.
  - h. Performance
    - i. On that note…
    - ii. It takes about a 10% change in performance for anybody to notice.
    - iii. It’s a question of choosing the right algorithms.
    - iv. Too many people focus on the speed of low-level code, but that rarely matters.
    - v. The design matters. If it’s a good design, you can focus on the performance issues that need attention later rather than focusing on *all* potential problems as you write code.
    - vi. Take CS209 next Spring for more information on performance.