CSCI-1200 Computer Science II — Fall 2012
Lecture 20 – Operators & Friends *and* Priority Queues, Part I

Review from Lecture 19

- A hash table is implemented with a array at the top level. Each key is mapped to a slot in the array by a *hash function*, a simple function of one argument (the key) which returns an index (a bucket or slot in the array).
- CallerID Performance: Vectors vs. Binary Search Trees vs. Hash Tables
- Hash Table Collision Resolution
- Using a Hash Table to Implement a `set`.
  - Function objects, Iterators, Fundamental operations: find, insert and erase.

Today's Lecture

- Finish material from Lecture 19: Hashset Implementation
- Operators as non-member functions, as member functions, and as friend functions.
- Queues and Stacks, What's a Priority Queue?
- A Priority Queue as a Heap, `percolate_up` and `percolate_down`
- HW6 Contest Results

Ford & Topp Sections 8.1,8.5-8.6.

20.1 Complex Numbers — A Brief Review

- Complex numbers take the form $z = a + bi$, where $i = \sqrt{-1}$ and a and b are real. a is called the real part, b is called the imaginary part.
- If $w = c + di$, then
  - $w + z = (a + c) + (b + d)i$,
  - $w - z = (a - c) + (b - d)i$, and
  - $w \times z = (ac - bd) + (ad + bc)i$
- The magnitude of a complex number is $\sqrt{a^2 + b^2}$.

20.2 `Complex` Class declaration (`complex.h`)

```
class Complex {
public:
  Complex(double x=0, double y=0) : real_(x), imag_(y) {}  // default constructor
  Complex(Complex const& old) : real_(old.real_), imag_(old.imag_) {}  // copy constructor
  Complex& operator= (Complex const& rhs); // Assignment operator
  double Real() const { return real_; }
  void SetReal(double x) { real_ = x; }
  double Imaginary() const { return imag_; }
  void SetImaginary(double y) { imag_ = y; }
  double Magnitude() const { return sqrt(real_*real_ + imag_*imag_); }
  Complex operator+ (Complex const& rhs) const;
  Complex operator- () const; // unary operator- negates a complex number
  friend istream& operator>> (istream& istr, Complex& c);
private:
  double real_, imag_;
};

Complex operator- (Complex const& left, Complex const& right); // non-member function
ostream& operator<< (ostream& ostr, Complex const& c);  // non-member function
```

20.3 Implementation of `Complex` Class (`complex.cpp`)

```
// Assignment operator
Complex& Complex::operator= (Complex const& rhs) {
  real_ = rhs.real_;
  imag_ = rhs.imag_;
  return *this;
}

// Addition operator as a member function.
Complex Complex::operator+ (Complex const& rhs) const {
  double re = real_ + rhs.real_;
  double im = imag_ + rhs.imag_;
  return Complex(re, im);
}

// Subtraction operator as a non-member function.
Complex operator- (Complex const& lhs, Complex const& rhs) {
  return Complex(lhs.Real()-rhs.Real(), lhs.Imaginary()-rhs.Imaginary());
}

// Unary negation operator.  Note that there are no arguments.
Complex Complex::operator- () const {
  return Complex(-real_, -imag_);
}

// Input stream operator as a friend function
istream& operator>> (istream & istr, Complex & c) {
  istr >> c.real_ >> c.imag_;
  return istr;
}

// Output stream operator as an ordinary non-member function
ostream& operator<< (ostream & ostr, Complex const& c) {
  if (c.Imaginary() < 0)  ostr << c.Real() << " - " << -c.Imaginary() << " i ";
  else                    ostr << c.Real() << " + " <<  c.Imaginary() << " i ";
  return ostr;
}
```

20.4 Operators as Non-Member Functions and as Member Functions

- We have already written our own operators, especially `operator<`, to sort objects stored in STL containers and to create our own keys for maps.
- We can write them as non-member functions (e.g., `operator-`). When implemented as a non-member function, the expression: `z - w` is translated by the compiler into the function call: `operator- (z, w)`
- We can also write them as member functions (e.g., `operator+`). When implemented as a member function, the expression: `z + w` is translated into: `z.operator+ (w)`

  This shows that `operator+` is a member function of `z`, since `z` appears on the left-hand side of the operator. Observe that the function has **only one** argument!

  There are several important properties of the implementation of an operator as a member function:

  - It is within the scope of class `Complex`, so private member variables can be accessed directly.
  - The member variables of `z`, whose member function is actually called, are referenced by directly by name.
  - The member variables of `w` are accessed through the parameter `rhs`.
  - The member function is `const`, which means that `z` will not (and can not) be changed by the function. Also, since `w` will not be changed since the argument is also marked `const`.
- Both `operator+` and `operator-` return `Complex` objects, so both must call `Complex` constructors to create these objects. Calling constructors for `Complex` objects inside functions, especially member functions that work on `Complex` objects, seems somewhat counter-intuitive at first, but it is common practice!

20.5 Assignment Operators

- The assignment operator: `z1 = z2;` becomes a function call: `z1.operator=(z2);`

  And cascaded assignments like: `z1 = z2 = z3;` are really: `z1 = (z2 = z3);`
  which becomes: `z1.operator= (z2.operator= (z3));`

  Studying these helps to explain how to write the assignment operator, which is usually a member function.

- The argument (the right side of the operator) is passed by constant reference. Its values are used to change the contents of the left side of the operator, which is the object whose member function is called. A reference to this object is returned, allowing a subsequent call to `operator=` (`z1`'s `operator=` in the example above).

  The identifier `this` is reserved as a pointer inside class scope to the object whose member function is called. Therefore, `*this` is a a reference to this object.

- The fact that `operator=` returns a reference allows us to write code of the form: `(z1 = z2).real();`

20.6 Exercise

Write an `operator+=` as a member function of the `Complex` class. To do so, you must combine what you learned about `operator=` and `operator+`. In particular, the new operator must return a reference, `*this`.

20.7 Returning Objects vs. Returning References to Objects

- In the `operator+` and `operator-` functions we create new `Complex` objects and simply return the new object. The return types of these operators are both `Complex`.

  Technically, we don't return the new object (which is stored only locally and will disappear once the scope of the function is exited). Instead we create a copy of the object and return the copy. This automatic copying happens outside of the scope of the function, so it is *safe* to access outside of the function. *Note: It's important that the copy constructor is correctly implemented!* Good compilers can minimize the amount of redundant copying without introducing semantic errors.

- When you change an existing object inside an operator and need to return that object, you must return a **reference** to that object. This is why the return types of `operator=` and `operator+=` are both `Complex&`. This avoids creation of a new object.

- A common error made by beginners (and some non-beginners!) is attempting to return a reference to a locally created object! This results in someone having a pointer to stale memory. The pointer may behave correctly for a short while... until the memory under the pointer is allocated and used by someone else.

20.8 Friend Classes vs. Friend Functions

- In the example below, the `Foo` class has designated the `Bar` to be a **friend**. This must be done in the `public` area of the declaration of `Foo`.

```
class Foo {
public:
  friend class Bar;
  ...
};
```

  This allows member functions in class `Bar` to access *all of* the private member functions and variables of a `Foo` object as though they were public (but not vice versa). Note that `Foo` is giving friendship (access to its private contents) rather than `Bar` claiming it. What could go wrong if we allowed friendships to be claimed?

- Alternatively, within the definition of the class, we can designate specific functions to be "`friend`"s, which grants these functions access similar to that of a member function. The most common example of this is operators, and especially stream operators.

20.9 Stream Operators as Friend Functions

- The operators `>>` and `<<` are defined for the `Complex` class. These are binary operators.
  The compiler translates: `cout << z3` into: `operator<< (cout, z3)`
  Consecutive calls to the `<<` operator, such as: `cout << "z3 = " << z3 << endl;`
  are translated into: `((cout << "z3 = ") << z3) << endl;`
  Each application of the operator returns an `ostream` object so that the next application can occur.

- If we wanted to make one of these stream operators a regular member function, it would have to be a member function of the `ostream` class because this is the first argument (left operand). *We cannot make it a member function of the* `Complex` *class.* This is why stream operators are never member functions.

- Stream operators are either ordinary non-member functions (if the operators can do their work through the public class interface) or friend functions (if they need non public access).

20.10 Summary of Operator Overloading in C++

- Unary operators that can be overloaded: `+ - * & ~ ! ++ -- -> ->*`

- Binary operators that can be overloaded: `+ - * / % ^ & | << >> += -= *= /= %= ^= &= |= <<= >>= < <= > >= == != && || , [] () new new[] delete delete[]`

- There are only a few operators that can not be overloaded: `. .* ?: ::`

- We can't create new operators and we can't change the number of arguments (except for the function call operator, which has a variable number of arguments).

- There are three different ways to overload an operator. When there is a choice, we recommend trying to write operators in this order:

  - Non-member function
  - Member function
  - Friend function

- The most important rule for clean class design involving operators is to **NEVER change the intuitive meaning of an operator**. The whole point of operators is lost if you do. One (bad) example would be defining the increment operator on a `Complex` number.

20.11 Extra Practice

- Implement the following operators for the `Complex` class (or explain why they cannot or should not be implemented). Think about whether they should be non-member, member, or friend.

  `operator*` `operator==` `operator!=` `operator<`

20.12 Additional STL Container Classes: Stacks and Queues

- We've studied STL vectors, lists, maps, and sets. These data structures provide a wide range of flexibility in terms of operations. One way to obtain computational efficiency is to consider a simplified set of operations or functionality. 2 examples are:

- **Stacks** allow access, insertion and deletion from only one end called the *top*

  - There is no access to values in the middle of a stack.
  - Stacks may be implemented efficiently in terms of vectors and lists, although vectors are preferable.
  - All stack operations are $O(1)$

- **Queues** allow insertion at one end, called the *back* and removal from the other end, called the *front*

  - There is no access to values in the middle of a queue.
  - Queues may be implemented efficiently in terms of a list. Using vectors for queues is also possible, but requires more work to get right.
  - All queue operations are $O(1)$

20.13 What's a Priority Queue?

- Priority queues are used in prioritizing operations. Examples include jobs on a shop floor, packet routing in a network, scheduling in an operating system, or events in a simulation.
- Among the data structures we have studied, their interface is most similar to a queue, including the idea of a `front` or `top` and a `tail` or a `back`.
- Each item is stored in a priority queue using an associated "priority" and therefore, the `top` item is the one with the lowest value of the priority score. The `tail` or `back` is never accessed through the public interface to a priority queue.
- The main operations are `insert` or `push`, and `pop` (or `delete_min`).

20.14 Some Data Structure Options for Implementing a Priority Queue

- Vector or list, either sorted or unsorted
  - At least one of the operations, `push` or `pop`, will cost linear time, at least if we think of the container as a linear structure.
- Binary search trees
  - If we use the priority as a `key`, then we can use a combination of finding the minimum key and erase to implement `pop`. An ordinary binary-search-tree insert may be used to implement `push`.
  - This costs logarithmic time in the average case (and in the worst case as well if balancing is used).
- The latter is the better solution, but we would like to improve upon it — for example, it might be more natural if the minimum priority value were stored at the root.
  - We will achieve this using a binary *heap*, giving up the complete ordering imposed in the binary *search tree*.

20.15 Definition: Binary Heaps

- A binary heap is a complete binary tree such that at each internal node, p, the value stored is less than the value stored at either of p's children.
  - A complete binary tree is one that is completely filled, except perhaps at the lowest level, and at the lowest level all leaf nodes are as far to the left as possible.
- Binary heaps will be drawn as binary trees, but implemented **using vectors**! (more on this next lecture)
- Alternatively, the heap could be organized such that the value stored at each internal node is greater than the values at its children.

20.16 Exercise: Drawing Binary Heaps

Draw two different binary heaps with these values: 52 13 48 7 32 40 18 25 4

20.17 Implementing Pop (a.k.a. Delete Min)

- The top (root) of the tree is removed.
- It is replaced by the value stored in the last leaf node.
  - This has echoes of the erase function in binary search trees.
  - We have not yet discussed how to find the last leaf.
- The last leaf node is removed.
- The (following) `percolate_down` function is then run to restore the heap property. This function is written here in terms of tree nodes with child pointers (and the priority stored as a `value`), but later it will be written in terms of vector subscripts.

```
percolate_down(TreeNode<T> * p) {
  while (p->left) {
    TreeNode<T>* child;
    //  Choose the child to compare against
    if (p->right && p->right->value < p->left->value)
      child = p->right;
    else
      child = p->left;
    if (child->value < p->value) {
      swap(child, p); // value and other non-pointer member vars
      p = child;
    }
    else
      break;
  }
}
```

20.18 Push / Insert

- To add a value to the heap, a new last leaf node in the tree is created and then the following `percolate_up` function is run. It assumes each node has a pointer to its parent.

```
percolate_up(TreeNode<T> * p) {
  while (p->parent)
    if (p->value < p->parent->value) {
      swap(p, parent);  // value and other non-pointer member vars
      p = p->parent;
    }
    else
      break;
}
```

20.19 Analysis

- Both `percolate_down` and `percolate_up` are $O(\log n)$ in the worst-case. Why?
- But, `percolate_up` (and as a result `push`) can be $O(1)$ in the average case. Why?
  (The full answer is beyond the scope of this course.)

20.20 Exercise

Suppose the following operations are applied to an initially empty binary heap of integers. Show the resulting heap after each `delete_min` operation. (Remember, the tree must be **complete**!)

```
push 5, push 3, push 8, push 10, push 1, push 6,
pop,
push 14, push 2, push 4, push 7,
pop,
pop,
pop
```