CSCI-1200 Data Structures — Spring 2013
Lecture 20 – Operators & Friends

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.

19.1 Hash Table in STL?

The Standard Template Library standard and implementation of hash table is slowly evolving. Depending on your system, you will use: `unordered_set` and `unordered_map` (proposed revisions to STL in TR1/C++0x/C++11) *or* `hash_set` and `hash_map` (an older, temporary collection of extensions).

19.2 Our Copycat Version: A Set As a Hash Table

- The class is templated over both the key type and the hash function type.

```
template  < class KeyType, class HashFunc >
class ds_hashset {
  ...
```

- We use separate chaining for collision resolution. Hence the main data structure inside the class is:

```
std::vector< std::list<KeyType> > m_table;
```

- We will use automatic resizing when our table is too full. Resize is expensive of course, so similar to the automatic reallocation that occurs inside the vector `push_back` function, we at least double the size of underlying structure to ensure it is rarely needed.

19.3 Function Objects, a.k.a. *Functors*

- The hash function (both in the STL hash table implementation, and our copycat version) is implemented as an object with a function call operator (a "functor").
- The basic form of the function call operator is shown below. Any specific number of arguments can be used.

```
class my_class_name {
public:
  // ... normal class stuff ...
  my_return_type operator() ( *** args *** );
};
```

- Here is an example of a templated function object implementing the less-than comparison operation. This is the default 3rd argument to `std::sort`.
  ```
  template <class T> class less_than {
  public:
    bool operator() (const T& x, const T& y) { return x<y; }
  };
  ```
- More interestingly... Constructors of functions objects can be used to specify *internal data* for the functor that can then be used during computation of the function call operator! For example:
  ```
  class between_values {
  private:
    int x, y;
  public:
    between_values(int in_x, int in_y) : x(in_x), y(in_y) {}
    bool operator() (int z) { return x <= z && z <= y; }
  }
  ```
  `x` & `y` are specificed when the functor is created. The functor accepts a single argument `z` that it compares against the internal data `x` & `y`. This can be used in combination with `find_if`. An example use with a vector of integers, `v`:
  ```
  between_values low_range(-99, 99);
  if (std::find_if(v.begin(), v.end(), low_range) != v.end())
    std::cout << "A value between -99 and 99 is in the vector.\n";
  ```

19.4 Our Hash Function Object

```
class hash_string_obj {
public:
  unsigned int operator() (std::string const& key) const {
    //  This implementation comes from   http://www.partow.net/programming/hashfunctions/
    unsigned int hash = 1315423911;
    for(unsigned int i = 0; i < key.length(); i++)
      hash ^= ((hash << 5) + key[i] + (hash >> 2));
    return hash;
  }
};
```

The type `hash_string_obj` is one of the template parameters to the declaration of a `ds_hashset`. E.g.,

```
ds_hashset<std::string, hash_string_obj> my_hashset;
```

19.5 Hash Set Iterators

- Iterators move through the hash table in the order of the storage locations rather than the ordering imposed by (say) an `operator<`. Thus, the visiting/printing order depends on the hash function and the table size.
  - Hence the increment operators must move to the next entry in the current linked list or, if the end of the current list is reached, to the first entry in the next non-empty list.
- The declaration is nested inside the `ds_hashset` declaration in order to avoid explicitly templating the iterator over the hash function type.
- The iterator must store:
  - A pointer to the hash table it is associated with. This reflects a subtle point about types: even though the `iterator` class is declared inside the `ds_hashset`, this does not mean an iterator automatically knows about any particular `ds_hashset`.
  - The index of the current list in the hash table.
  - An iterator referencing the current location in the current list.
- Because of the way the classes are nested, the `iterator` class object must declare the `ds_hashset` class as a friend, but the reverse is unnecessary.

19.6 Implementing `begin()` and `end()`

- `begin()`: Skips over empty lists to find the first key in the table. It must tie the iterator being created to the particular `ds_hashset` object it is applied to. This is done by passing the `this` pointer to the iterator constructor.
- `end()`: Also associates the iterator with the specific table, assigns an index of -1 (indicating it is not a normal valid index), and thus does not assign the particular list iterator.
- **Exercise:** Implement the `begin()` function.

19.7 Iterator Increment, Decrement, & Comparison Operators

- The increment operators must find the next key, either in the current list, or in the next non-empty list.
- The decrement operator must check if the iterator in the list is at the beginning and if so it must proceed to find the previous non-empty list and then find the last entry in that list. This might sound expensive, but remember that the lists should be very short.
- The comparison operators must accommodate the fact that when (at least) one of the iterators is the `end`, the internal list iterator will not have a useful value.

19.8 Insert & Find

- Computes the hash function value and then the index location.
- If the key is already in the list that is at the index location, then no changes are made to the set, but an iterator is created referencing the location of the key, a pair is returned with this iterator and `false`.
- If the key is not in the list at the index location, then the key should be inserted in the list (at the front is fine), and an iterator is created referencing the location of the newly-inserted key a pair is returned with this iterator and `true`.
- **Exercise:** Implement the `insert()` function, ignoring for now the `resize` operation.
- Find is similar to insert, computing the hash function and index, followed by a `std::find` operation.

19.9 Erase

- Two versions are implemented, one based on a key value and one based on an iterator. These are based on finding the appropriate iterator location in the appropriate list, and applying the list erase function.

19.10 Resize

- Must copy the contents of the current vector into a scratch vector, resize the current vector, and then re-insert each key into the resized vector. **Exercise:** Write `resize()`

19.11 Hash Table Iterator Invalidation

- Any insert operation invalidates *all* `ds_hashset` iterators because the insert operation could cause a resize of the table. The erase function only invalidates an iterator that references the current object.

20.12 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 $\sqrt{a^2 + b^2}$.

20.13 `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
```

20.14 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;
}
```

20.15 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!

20.16 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();`

20.17 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`.

20.18 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.

20.19 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.

20.20 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).

20.21 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.

20.22 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<`