

CSCI-1200 Computer Science II — Fall 2008

Lecture 19 – Operators & Friends and Hash Tables, Part I

Announcements: Test 3 Information

- Test 3 will be held on **Tuesday, November 11th, 2008 from 2-3:50pm in West Hall Auditorium**. No make-ups will be given except for emergency situations, and even then a written excuse from the Dean of Students office will be required.
- Coverage: Lectures 1-19, Labs 1-11, HW 1-8.
- Closed-book and closed-notes *except for 1 sheet of notes on 8.5x11 inch paper (front & back) that may be handwritten or printed.*

Review from Lecture 18

- Longest & shortest paths, depth-first vs. breadth-first, increment and decrement operations on iterators, red-black trees

Today's Lecture

- Operators as non-member functions, as member functions, and as friend functions.
- Hash Tables, Hash Functions, and Collision Resolution
- Hash Table Performance, Binary Search Trees vs. Hash Tables

Ford & Topp Sections 12.1-12.2.

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

19.2 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
```

19.3 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;
}
```

19.4 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!

19.5 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 reference to this object.

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

19.6 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`.

19.7 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.

19.8 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.

19.9 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 this function a member function, it would have to be a member function of the `ostream` class because this is the first argument. *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).

19.10 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.

19.11 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<
```

19.12 Definition: What's a Hash Table?

- A table implementation with *constant time access*.
 - Like a map, we can store key-value pair associations in the hash table.
But it's even faster to do find, insert, and erase with a hash table!
However, hash tables *don't* store the data in sorted order.
- A hash table is implemented with an array at the top level.
- Each key is mapped to a slot in the array by a *hash function*.

19.13 Definition: What's a Hash Function?

- A simple function of one argument (the key) which returns an index (a bucket or slot in the array).
- Ideally the function will "uniformly" distribute the keys throughout the range of legal index values ($0 \rightarrow k-1$).
- **What's a collision?**
When the hash function maps multiple (different) keys to the same index.
- **How do we deal with collisions?**
One way to resolve this is by storing a linked list of values at each slot in the array.

19.14 Example: Caller ID

- We are given a phonebook with 50,000 name/number pairings. Each number is a 10 digit number. We need to create a data structure to lookup the name matching a particular phone number. Ideally, name lookup should be $O(1)$ time expected, and the caller ID system should use $O(n)$ memory ($n = 50,000$).
- We'll review how we solved this problem in Lab 9 with an STL `vector` then an STL `map`. Finally, we'll implement the system with a hash table.
- Note: In the toy implementations that follow we use small datasets, but we should evaluate the system scaled up to handle the large dataset.

19.15 Caller ID with an STL Vector

```
void add(vector<string> &phonebook, int number, string name) {
    phonebook[number] = name;
}

void identify(const vector<string> &phonebook, int number) {
    if (phonebook[number] == "UNASSIGNED") cout << "unknown caller!" << endl;
    else cout << phonebook[number] << " is calling!" << endl;
}

int main() {
    // create the phonebook, initially all numbers are unassigned
    vector<string> phonebook(10000, "UNASSIGNED");
    // add several names to the phonebook
    add(phonebook, 1111, "fred");
    add(phonebook, 2222, "sally");
    add(phonebook, 3333, "george");
    // test the phonebook
    identify(phonebook, 2222);
    identify(phonebook, 4444);
}
```

Exercise: What's the memory usage for the vector-based Caller ID system?
What's the expected running time for find, insert, and erase?

19.16 Caller ID with an STL Map

```
void add(map<int,string> &phonebook, int number, string name) {
    phonebook[number] = name;
}

void identify(const map<int,string> &phonebook, int number) {
    map<int,string>::const_iterator tmp = phonebook.find(number);
    if (tmp == phonebook.end()) cout << "unknown caller!" << endl;
    else cout << tmp->second << " is calling!" << endl;
}

int main() {
    // create the phonebook, initially all numbers are unassigned
    map<int,string> phonebook;
    // add several names to the phonebook
    add(phonebook,1111,"fred");
    add(phonebook,2222,"sally");
    add(phonebook,3333,"george");
    // test the phonebook
    identify(phonebook,2222);
    identify(phonebook,4444);
}
```

Exercise: What's the memory usage for the map-based Caller ID system?
What's the expected running time for find, insert, and erase?

19.17 Now let's implement Caller ID with a Hash Table

```

#define PHONEBOOK_SIZE 10

class Node {
public:
    int number;
    string name;
    Node* next;
};

// corresponds a phone number to a slot in the array
int hash_function(int number) {

}

// add a number, name pair to the phonebook
void add(Node* phonebook[PHONEBOOK_SIZE],
         int number, string name) {

}

// given a phone number, determine who is calling
void identify(Node* phonebook[PHONEBOOK_SIZE], int number) {

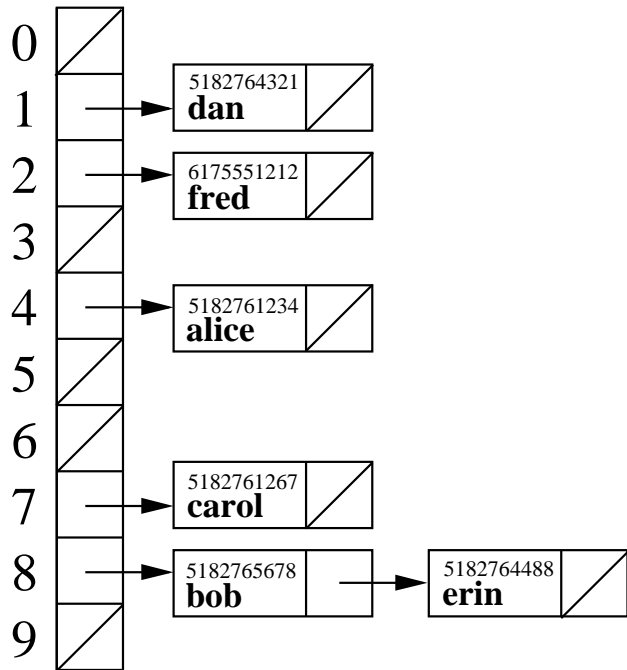
}

int main() {
    // create the phonebook, initially all numbers are unassigned
    Node* phonebook[PHONEBOOK_SIZE];
    for (int i = 0; i < PHONEBOOK_SIZE; i++)
        phonebook[i] = NULL;

    // add several names to the phonebook
    add(phonebook, 1111, "fred");
    add(phonebook, 2222, "sally");
    add(phonebook, 3333, "george");

    // test the phonebook
    identify(phonebook, 2222);
    identify(phonebook, 4444);
}

```



19.18 Exercise: Choosing a Hash Function

- What's a good hash function for this application?
- What's a bad hash function for this application?

19.19 Exercise: Hash Table Performance

- What's the memory usage for the hash-table-based Caller ID system?
- What's the expected running time for find, insert, and erase?