Data structure cheat sheet
Final

Lecture 19 – Operators & Friends

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.

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).

• 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.

• 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. 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.

Lecture 20 – Hash Tables

• A table implementation with constant time access.
• A hash table is implemented with an array at the top level.
• Each element or key is mapped to a slot in the array by a hash function.
• Goals: fast O(1) computation and a random, uniform distribution of keys throughout the table, despite the actual distribution of keys that are to be stored.

• In open addressing, when the chosen table location already stores a key (or key-value pair), a different table location is sought in order to store the new value (or pair).
• Here are three different open addressing variations to handle a collision during an insert operation:
– Linear probing: If i is the chosen hash location then the following sequence of table locations is tested ("probed") until an empty location is found: $(i+1)\%N, (i+2)\%N, (i+3)\%N, ...$
– Quadratic probing: If i is the hash location then the following sequence of table locations is tested: $(i+1)\%N, (i+2*2)\%N, (i+3*3)\%N, (i+4*4)\%N, ...$ More generally, the jth "probe" of the table is $(i+c_1 j+c_2 j^2) mod N$ where c1 and c2 are constants.
– Secondary hashing: when a collision occurs a second hash function is applied to compute a new table location. This is repeated until an empty location is found.
• For each of these approaches, the find operation follows the same sequence of locations as the insert operation. The key value is determined to be absent from the table only when an empty location is found.
• When using open addressing to resolve collisions, the erase function must mark a location as "formerly occupied". If a location is instead marked empty, find may fail to return elements in the table. Formerly-occupied locations may (and should) be reused, but only after the find operation has been run to completion.
• Problems with open addressing:
– Slows dramatically when the table is nearly full (e.g. about 80% or higher). This is particularly problematic for linear probing.
– Fails completely when the table is full.
– Cost of computing new hash values.

Lecture 21 – Functors & Hash Tables, part II

• 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)

Given a pointer to the root node in a binary tree:
• Use an STL stack to print the elements with a pre-order traversal ordering. This is straightforward.
• Use an STL stack to print the elements with an in-order traversal ordering. This is more complicated.
• Use an STL queue to print the elements with a breadth-first traversal ordering.

• Priority queues are used in prioritizing operations. Examples include a personal "to do" list, what order to do homework assignments, 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).

Lecture 22 – Priority Queues

• 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!
• Alternatively, the heap could be organized such that the value stored at each internal node is greater than the values at its children.

Implementing Pop (a.k.a. Delete Min):
The value at the top (root) of the tree is replaced by the value stored in the last leaf node. This has echoes of the erase function in binary search trees. The last leaf node is removed. The value now at the root likely breaks the heap property. We use the percolate_down function to restore the heap property.

```
percolate_down(TreeNode<T> * p) {
    while (p->left) {
        TreeNode<T>* child;
        if (p->right && p->right->value < p->left->valu
            child = p->right;
        else child = p->left;
        if (child->value < p->value) {
            swap(child, p);
            p = child;
```

```
        } else break;
    }
}
```

Implementing Push (a.k.a. Insert): To add a value to the heap, a new last leaf node in the tree is created to store that value. Then the 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);
            p = p->parent;
        } else break;
    }
}
```

• In the vector implementation, the tree is never explicitly constructed. Instead the heap is stored as a vector, and the child and parent "pointers" can be implicitly calculated.
• To do this, number the nodes in the tree starting with 0 first by level (top to bottom) and then scanning across each row (left to right). These are the vector indices. Place the values in a vector in this order. • As a result, for each subscript, i,
– The parent, if it exists, is at location $[(i-1)/2]$.
– The left child, if it exists, is at location $2i+1$.
– The right child, if it exists, is at location $2i+2$.
• For a binary heap containing n values, the last leaf is at location $n-1$ in the vector and the first node with less than two children is at location $[(n-1)/2]$.

Lecture 23 — C++ Inheritance and Polymorphism

• Constructors of a derived class call the base class constructor immediately, before doing ANYTHING else. The only thing you can control is which constructor is called and what the arguments will be.
• The reverse is true for destructors: derived class constructors do their jobs first and then base class destructors are called at the end, automatically. Note: destructors for classes which have derived classes must be marked virtual for this chain of calls to happen.
• A derived class can redefine member functions in the base class. The function prototype must be identical, not even the use of const can be different (otherwise both functions will be accessible).
• protected and private inheritance are other options:
– With protected inheritance, public members becomes protected and other members are unchanged
– With private inheritance, all members become private.

• Some of these functions are marked virtual, which means that when they are redefined by a derived class, this new definition will be used, even for pointers to base class objects.
• Some of these virtual functions, those whose declarations are followed by = 0 are pure virtual, which means they must be redefined in a derived class.
– Any class that has pure virtual functions is called "abstract".
– Objects of abstract types may not be created — only pointers to these objects may be created.
• Functions that are specific to a particular object type are declared in the derived class prototype.

Lecture 24 — C++ Exceptions

• When you detect an error, throw an exception.
• You can throw a value of any type (e.g., int, std::string, an instance of a custom class, etc.)
• When the throw statement is triggered, the rest of that block of code is abandoned.
• If you suspect that a fragment of code you are about to execute may throw an exception and you want to prevent the program from crashing, you should wrap that fragment within a try/catch block
• The logic of the try block may throw more than one type of exception.
• A catch statement specifies what type of exception it catches (e.g., int, std::string, etc.)
• You may use multiple catch blocks to catch different types of exceptions from the same try block.
• You may use catch (...) /* code */ to catch all types of exceptions. (But you don't get to use the value that was thrown!)
• If an exception is thrown, the program searches for the closest enclosing try/catch block with the appropriate type. That try/catch may be several functions away on the call stack (it might be all the way back in the main function!).
• If no appropriate catch statement is found, the program exits.

Resource Acquisition Is Initialization (RAII): • Because exceptions might happen at any time, and thus cause the program to abandon a partially executed function or block of code, it may not be appropriate to rely on a delete call that happens later on in a block of code.
• RAII describes a programming strategy to ensure proper deallocation of memory despite the occurrence of exceptions. The goal is to ensure that resources are released before exceptions are allowed to propagate.
• Variables allocated on the stack (not dynamically-allocated using new) are guaranteed to be properly destructed when the variable goes out of scope (e.g., when an exception is thrown and we abandon a partially executed block of code or function).
• Special care must be taken for dynamically-allocated variables (and other resources like open files, mutexes, etc.) to ensure that the code is exception safe.

Lecture 26 — Concurrency & Asynchronous Computing

• No two operations that change any shared state variables may occur at the same time.
– Certain low-level operations are guaranteed to execute atomic-ly (from start to finish without interruption), but this varies based on the hardware and operating system. We need to know which operations are atomic on our hardware.
– In the bank account example we cannot assume that the deposit and withdraw functions are atomic.
• The concurrent system should produce a result of the threads/processes running sequentially in some order.
– We do not require that the threads/processes run sequentially, only that they produce results as if they had run sequentially.

• We can serialize the important interactions using a primitive, atomic synchronization method called a mutex.
• Once one thread has acquired the mutex (locking the resource), no other thread can acquire the mutex until it has been released.
• In the example below we use the STL mutex object (#include <mutex>). If the mutex is unavailable, the call to the mutex member function lock() blocks (the thread pauses at that line of code until the mutex is available).

• So how exactly do we get multiple streams of computation happening simultaneously? There are many choices (may depend on your programming language, operating system, compiler, etc.). • We'll use the STL thread library (#include <thread>). The new thread begins execution in the provided function (student thread, in this example). We pass the necessary shared data from the main thread to the secondary thread to facilitate communication.

DS_SET.H

```
// PRIVATE HELPER FUNCTIONS
TreeNode<T>*  copy_tree(TreeNode<T>* old_root, TreeNode
    if (old_root == NULL) return NULL;
    TreeNode<T> *answer = new TreeNode<T>();
    answer->value = old_root->value;
    answer->left = copy_tree(old_root->left,answer);
    answer->right = copy_tree(old_root->right,answer);
    answer->parent = the_parent;
    return answer;
}

void destroy_tree(TreeNode<T>* p) {
    if (!p) return;
    destroy_tree(p->right);
    destroy_tree(p->left);
    delete p;
}

iterator find(const T& key_value, TreeNode<T>* p) {
    if (!p) return end();
    if (p->value > key_value) return find(key_value, p-
    else if (p->value < key_value) return find(key_valu
    else return iterator(p,this);
}

std::pair<iterator,bool> insert(const T& key_value, Tre
    if (!p) {
        p = new TreeNode<T>(key_value);
        p->parent = the_parent;
```

```
        this->size_++;
        return std::pair<iterator,bool>(iterator(p,this), true);
    }
    else if (key_value < p->value)
        return insert(key_value, p->left, p);
    else if (key_value > p->value)
        return insert(key_value, p->right, p);
    else
        return std::pair<iterator,bool>(iterator(p,this), false);
}

int erase(T const& key_value, TreeNode<T>* &p) {
    if (!p) return 0;

    // look left & right
    if (p->value < key_value)
        return erase(key_value, p->right);
    else if (p->value > key_value)
        return erase(key_value, p->left);

    // Found the node.  Let's delete it
    assert (p->value == key_value);
    if (!p->left && !p->right) { // leaf
        delete p;
        p=NULL;
        this->size_--;
    } else if (!p->left) { // no left child
        TreeNode<T>* q = p;
        p=p->right;
        assert (p->parent == q);
        p->parent = q->parent;
        delete q;
        this->size_--;
    } else if (!p->right) { // no right child
        TreeNode<T>* q = p;
        p=p->left;
        assert (p->parent == q);
        p->parent = q->parent;
        delete q;
        this->size_--;
    } else { // Find rightmost node in left subtree
        TreeNode<T>* q = p->left;
        while (q->right) q = q->right;
        p->value = q->value;
        // recursively remove the value from the left subtree
        int check = erase(q->value, p->left);
        assert (check == 1);
    }
    return 1;
}
```