Shamit Bhatia

Feb 18th, 2017

Lecture Notes up to Midterm

**MIDTERM STUDY GUIDE**

**Lecture 1:**

* #include <iostream>
  + std::cout, std::cin
* main always has to return an int, cannot return void
* using namespace std
  + pros – less typing
  + cons – can mess with functions in standard library
* DON’T USE NAMESPACE STD, instead
  + using std::cout;
  + using std::endl;
* Enumeration - Allows you to make custom data type
  + enum Coin

{

* + - Penny = 1,
    - Nickel = 5,
    - Dime = 10,

};

* Must write function headers:
  + Function: Add
  + Purpose: Adds two integers
  + Input: Two integers to add
  + Output: Returns the integral result
* Function Overloading
  + Only differs by parameters either in number or return type
  + CANNOT just change the return type of the entire function
* #pragma once
  + Protect against accidentally including same header file twice

**Lecture 2:**

Std::strings:

* “Nathan” + “Greenfield” are string literals and not technically std::string classes
  + Need to cast one -> std::string(“Nathan”) + “Greenfield” works
* **4 basic data types**
  + **char**
  + **Boolean**
  + **Int**
  + **Double**
* Can access string characters by doing varName[3] sort of like arrays
* since std::string is a class, can access member functions
  + name.length();
* String member functions
  + Clear – clears entire string so its empty
  + Empty – tells you whether or not string is empty
    - Returns bool true/false
  + Front – gets first character in string
    - Returns char
  + Back – gets last character in string (before null terminator)
    - Returns char
  + Find – searches for the first appearance of another string
    - Parameters (string searching for, index you want to start at)
    - Returns index at which the thing appears
      * But since indexes are never negative, you can’t have the thing return like -1 if it doesn’t exist bc its an unsigned int so you have to use std::string::npos
      * If you don’t find the string then you get the value that std::string::npos is equal to (some large ass #)
      * Combat this by doing this:
      * If (title.find(“hello”) == std::string::npos){
        + Cout << “Didn’t find string << endl;}
  + **Substring – returns a new string that’s a substring**
    - **Parameters (index to start at, the number of characters to include)**
    - **Returns a new string**
  + **Overloading a function: when function named same but can have different number of parameters, like substring bc it has optional second parameter**

**Function parameters:**

* Pass by value (makes copy, changes don’t persist)
* Pass by reference (int& x, int& y)
* **\*\*\*\*Practice this stuff\*\*\*\***
* Another reason to pass by reference is that it makes things much faster
* Basic types, usually pass by value unless you DO want parameter modifications to persist
* Non-basic types, usually pass by reference to avoid the copy (string, ifstream, ofstream)

**Stanford C++ Library:**

* Convert to/from strings
* Graphical drawing
* Collections (data structures)
* GWindow class
  + GWindow gw(500, 500)
    - Creates instance of GWindow that’s 500x500
  + Origin is top left corner (0,0)
* drawRect function to draw rectangle
  + gw.drawRect(x, y, width, height)
    - x/y are top left coordinates, width/height of rect.
* fillRect function to fill rectangle
  + gw.fillRect(x, y, width, height)
* setColor
  + gw.setColor(color)
  + Ex. gw.setColor(“blue”);
  + gw.fillRect(50, 50, 200, 100) 🡪 will be in blue
  + gw.setColor(“red”);
  + gw.fillRect(50, 50, 200, 100) 🡪 will be in red
* draw a text string (label)
  + gw.drawLabel(text, x, y)
    - text is the std::string you want to draw
    - x/y are coordinates of the center of the text

**Collections – type of data structure that can store elements:**

* Arrays, lists, dictionaries,
* **Strings ARE collections (of char’s)**
* **Enums ARE NOT collections (can only hold one type of enum at a time)**
* Array
  + **Problem 1: C++ arrays are created with a fixed size**
  + **Problem 2: Arrays don’t know their own size. Can go too far in a for loop trying to access array elements**
  + **Problem 3: If we don’t initialize the array to values, it will have random garbage data**
* Vector is a good fix to an array
  + Automatically grows (and shrinks)
  + Allows to insert and remove elements whenever you want to
  + Can check to make sure you aren’t accessing invalid index
  + Stanford Vector:
    - #include “vector.h”
    - Vector<int> myVector;
  + Size function
    - myVector.size()
  + Add an element to the end of the vector
    - myVector.add(element) -> the element type must match type of Vector
  + Remove an element from vector at specified index
    - myVector.remove(index)
    - If you were to remove beginning element, vector has to copy/shift all indices to the left by 1
  + Insert an element **before** the specified index
    - myVector.insert(index, element)
  + Check if a vector is empty
    - myVector.isEmpty()
    - returns a Boolean true or false
  + Removes everything from the vector
    - myVector.clear()

**Lecture 3:**

* Cin only reads up to the first whitespace (space, tab, enter)
* Std::getline solves this
  + Parameters: input stream, string to store the line in
* If you do a cin and then a getline
  + It won’t work well -> c++ stops reading as soon as it hits whitespace
  + Fix by doing cin.ignore() –> skips everything until the newline
  + Ex. 5 ABC -> it will not put ABC in getline it will stop at whitespace
  + If you do cin.ignore(), then it will be something like
    - 5
    - ABC
* File stream
  + #include <fstream>
  + std::ofstream fileStream(“Output.txt”)
  + fileStream << “Hello”; // writes to the file Output.txt
  + fileStream.close() // close bc only can have 1 stream connected to a file
* Input file stream
  + Std::ifstream fileStream(“Input.txt”)
    - Int num = 0;
    - fileStream >> num;
  + Check if file didn’t open properly
    - If (fileInput.is\_open()){
      * Do stuff
    - Else{
      * Cout << “Error: file not there”;
  + Check if you’re at the end of file
    - Eof returns bool when ur at end of file or not
    - While (fileInput.eof() != true){
      * Int num
      * FileInput >> num;
      * Cout << num << endl;
* **String streams**
  + **#include <sstream>**
  + **std::istreamstream**
  + **std::ostringstream**
  + **Basically lets you take apart each number in a stream**
    - **Istringstream stream stream(“1 2 3”)**
    - **While (!stream.eof()){**
      * **int x;**
      * **stream >> x;**
      * **std::cout << x << std::endl;**

**Lecture 4:**

* **STRINGS ARE COLLECTIONS OF CHARS**
* Vector example – Average
  + Pass by reference because more efficient even if avg. doesn’t change vector
* Queue – collection
  + Allows to add elements to end (tail) and remove from front (head)
  + **Cannot read from any arbitrary element unlike arrays**
  + FIFO (first in, first out)
  + Ex. Printer
    - Queue to a movie
    - Can only access beginning of a queue cant access middle
  + #include “queue.h”
  + Queue<int> myQueue;
* Queue member functions
  + isEmpty - returns bool
  + clear – clears everything from queue
  + size – returns number of elements in queue
  + **Enqueue** – adds an element to the tail (end)
    - myQueue.enqueue(element);
  + **Dequeue** – removes element at the head (front) of queue – **cannot read from the tail**
    - Cout << myQueue.dequeue() << endl;
  + **Peek** – returns element at the head (front) but does NOT remove item
* Stacks – collection
  + Allows you to add elements to top of stack
  + Allows you to remove elements from the top of the stack
  + **Cannot read from any arbitrary element unlike arrays**
  + LIFO (last in, first out)
  + Ex. Stack of plates at parkside
    - the last plate put on stack is first to be washed
  + #include “stack.h”
  + Stack<int> myStack;
* Stack member functions
  + Push – push an element onto top of the stack
    - myStack.push(element)
  + pop – removes element form the top of the stack and returns it
    - type OldTop = mystack.pop();
* **Infix notation – binary operators appear in between operands**
  + **5 + 10**
  + **not efficient way for a computer to perform arithmetic with numbers**
* **Postfix notation – appears after operands**
  + **5 10 +**
  + **5 10 + 2 \* means (5 + 10) \* 2**
* **Physical calculator examples**

**Lecture 5:**

Sets:

* Set is a collection of distinct objects
  + P = {2, 3, 5, 7, 11}
* Empty set is a set that contains nothing
  + O = {}
* Union between sets A and B = A U B -> contains all objects that are in either A or B
  + P = {2, 3, 5, 7, 11}
  + C = {4, 6, 8, 9, 10}
  + P U C = {2, 3, 4, 5, 6, 7, 8, 9, 10, 11}
* Intersection
  + Set that contains all objects that are in BOTH A and B
* Subset
  + A is a subset of B if all the elements in A are also in B
  + C is a subset of C++ -> anything you can do in C you can do in C++ but doesn’t work the other way around
* Stanford C++ set -> **DON’T ALLOW RANDOM ACCESS LIKE ARRAYS, CAN ONLY SEE WHETHER IN OR OUT OF SET**
  + #include “set.h”
  + Set<int> mySet;
* Set member functions
  + Add – add element to set
    - mySet.add(element)
    - **IMPORTANT: Stored in ascending alphabetical order IN STANFORD SETS**
  + Contains
    - Check if set contains requested value -> if so, returns true
  + Remove
    - mySet.remove(element)
  + Union
    - Use the + operator to perform a union between two sets
    - mySet = set1 + set2
  + Intersection
    - Use the \* operator to perform a intersection between two sets
    - mySet = set1 \* set2
  + isSubsetOf
    - mySet.isSubsetOf(set2) -> retrusn true if set is a subset of the set passed as a parameter

**Maps:**

* Type of collection allows you to associate **a unique key with a value**
* Also called associative arrays (known as dictionary in python)
* Stanford C++ Map
  + #include “map.h”
  + **Map<int, std::string> myMap;**
    - **Keys are integers, and values are strings**
* Map member functions
  + Put – associates specified key with a value
    - myMap.put(key, value)
    - Ex. Map<int, std::string> months;
      * Months.put(1, “January”)
      * Months.put(2, “February”)
  + ContainsKey
    - myMap.containsKey(key) -> returns true if key is associated with a value in map
  + Get – returns the value associated with the key, provided that key is in Map
    - String value = myMap.get(key)
  + [] operator
    - Instead of put/get you can use []
    - Ex. Map<int, std::string> months
    - months[1] = “January”
    - months[2] = “February”
  + Remove
    - myMap.remove(key) -> removes key and associated value
* Range-based For loop – used to iterate over sets
  + For (type i : collection)
    - Visits each element in a collection
  + Modification in range-based
    - For (int& number : fibNumbers
      * Number \*= 2

If you use range-based for loops to iterate maps, you will only get keys

**Lecture 6:**

* Struct (structure) allows us to group together related variables into a single variable
* Classes can have both member variables (properties) and member functions
  + AND can prevent other code from directly modifying member variables
  + Private variables – cannot be directly accessed from outside the class
    - **All member variables should be private – known as data encapsulation**
  + Public – can be accessed outside class
  + Getter function
    - Takes no input (return it by value – NOT reference)
    - Ex. int getHours()
      * Return mHours;
  + Setter functions – allow to set member variables
    - Returns void and takes some input
    - Ex. Void setHours (int newHours)
* Splitting up into files
  + Clock.h
    - #Pragma once
    - Maybe #include “string”
  + Clock.cpp – the guts and implementation of the “.h” file
    - #include “Clock.h”
    - #include <iostream>
    - void Clock::reset()
    - **There are 4 parts to every function that belong to a class – return type, class to which it belongs, function name, input**
* Constructors
  + Special type of member function that’s automatically called when an instance of the class is created
  + **Default constructor takes no parameters**
    - Ex. Clock(); -> in “.h”
      * **Function named same as class**
      * **No return type – NOT even void**
    - In “.cpp”
      * Clock::Clock()
        + mHours = 0
        + mMinutes = 0
        + mSeconds = 0
  + **Constructor with parameters**
    - “h.”
      * Clock(int hours, int minutes, int seconds)
    - “.cpp”
      * Clock::Clock(int hours, int minutes, int seconds)
        + mHours = hours
        + mMinutes = minutes
        + mSeconds = seconds
  + Operator Overloading – allows us to define operators for classes/structs we’ve declared
    - friend bool operator==(Clock& left, Clock& right)
      * **friend means that it’s a standalone func. That’s allowed to access private data of the class -> therefore not a class function but since you write friend you have access to private data of class**
      * bool returns boolean

operator == -> overloading the operator

**Lecture 7:**

* a + b is the same as operator+(a, b)
  + Not a friend function
* a += b
* a.operator+=(b)
* (a+=3)++ -> adds 4 to add
* **Recursion**
  + **Sometimes called divide-and-conquer**
  + **Factorial**
    - **If (num == 0)**
      * **Return**
    - **Else**
      * **Return num \* Factorial(num – 1)**
  + **Base case – what prevents recursive function from going on forever**
  + **Recursive call**
* Lab
  + 235.12
    - To get the whole number on left side you truncate using int(235.12)
    - To get the 12 after the decimal, do int (235.12 \* 100) % 100

**Lecture 8:**

* Come up with a recursive solution to a problem you only need to convince
  + Base case is correct
  + Recursive call is provided parameters that are simpler in a useful way
* **Stack overflow – occurs if there are too many functions active at once**
  + **Usually happens when your base case is wrong**
  + To get past this, can use iterative solution using loop
    - But Fact(1000000) will give you 0 if using iterative solution because integer size will be greater than 4270 number or w/e -> luckily doesn’t crash program
* **Any recursive solution can be done iteratively and vice versa**
* Example Fibonacci Recursion
  + If n < 2, n
  + If n > 1, Fn-1 + Fn-2

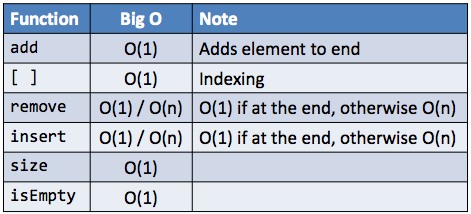
**Lecture 9:**

* Can use recursion to draw squares over and over again
* Base case -> width <= 0
* Sum #’s of vector
  + Iteratively
    - Int sum = 0;
    - For (int I = 0; I < numbers.size(); i++)
      * Sum += numbers[i]
    - Return sum;
  + **Recursively**
    - Vector of numbers
    - Summing it all up means adding 1st number + vector of numbers after that
    - Then take 2nd number + vector of numbers after that, etc.
    - Base case?
      * If startIndex == endIndex
    - **Need to add a “helper” function when you need to keep track of indices to do the recursion but you don’t need indices for overall program**
      * **So the recursion actually happens in helpfer function**
    - Int sumOfVectorPart(Vector<int>& numbers, unsigned start, unsigned end)
      * // Base case: if start == end
      * if (start == end){
        + //return number at end index
        + return numbers[end];
      * else{
        + // add number at start index + a new part of the remaining indices
        + return numbers[start] + sumOfVectorPart(numbers, start+1, end)
    - Then add this helper function to original function
      * Int sumOfVector(Vector<int>& numbers)

Return SumOfVectorPart(numbers, 0, numbers.size() – 1)

**Lecture 10:**

* Searching for a movie in a vector
  + Simplest solution: start at index – and keep incrementing until we find movie we are looking for (Top Gun)
  + **Linear Search**
    - Loop from index 0 to size
      * Check at each index if movie = Top Gun?
      * If so, we found movie and return index
      * If we reach end of vector and we don’t find movie, return -1
    - Linear Search: Worst Case
      * If we want to see if a movie is NOT in a list, we have to go through all elements in the list: n elements
  + Another way: BINARY SEARCH
    - Only works on vectors that are already sorted
    - What if we have an alphabetized (sorted) list
      * If we start at middle of list (Inception) then we check and since Inception is NOT Top gun, we know top gun has to be in 2nd half of list IF it is there.
      * Keep repeating
    - Algorithm:
      * Start with a range of indices
      * Check index at middle of range
      * If contains value we’re looking for, return it!
      * Otherwise, is the value we’re looking for greater than the middle value?
        + If so, cut the range into the first half, and do another binary search
        + Otherwise, cut the range into the second half, and do another binary search
    - Binary search, worst case
      * Log2(n) -> which means 2^x = n
      * For log2(10) we visit roughly 3.3 elements
  + Linear Vs. Binary Search
    - Linear much steeper
    - Worst case: How many steps do you have to check list if you are 100% sure it isn’t in list. For binary search it is log2(n). For linear search it is (n)
  + Big O Notation
    - Convention used to describe the worst-case performance of a particular algorithm
    - Linear search: O(n)
    - Binary search: O(log n)
    - Constant time: if performance is independent of the size of collection
      * O(1)
      * Ex. Accessing an index in a Vector
        + bigVec[0] = 5
    - Random accessing – constant O(1)
    - Insertion or removal of elements at the end – amortized constant O(1)
    - Insertion or removal of elements – linear in distance to the end of the vector O(n)



* Queue and Stack all functions are O(1)
* Set – behind the scenes is in ascending order so is using binary search
  + Add is O(log n)
  + Contains is O (log n)
* Map – behind the scenes sorted with keys in ascending order
  + Put – O(log n)
  + containsKey - O(log n)

get - O(log n)

**Midterm Study Guide:**

* Types of Collections (7 so far)
  + Array
    - Index ranges: 0 -> size -1
  + Vector
    - Big O time and table
    - Iteration over RANGE BASED FOR LOOP
  + Map
    - 2 step process -> check if it has they key, then access it
  + Queue
    - Enqueue and dequeue
  + Stack
    - Push, pop, and peek
  + String
    - Found, substring
  + Set
    - After added to set -> check with contains (don’t use push/pop)
* Enums: know about it
* Input/Output: is\_open, .eof, getline, string stream KNOW STRING STREAMS
* Big O’s to know
  + O(n)
  + O(1)
  + O(log n)
  + O(N^2)
* Classes:
  + Normal functions, getters, setters, etc.
  + Operator overloads, .h, and write cpp’s
  + Friend functions (no member functions, but have access) -> lab 4/5
* Know recursion (write and use)
  + Fibonacci sequence
* Random
  + Function flow (stack of papers)
  + Stack overflow (when you call too many functions) -> bad base case or factorial of a billion
* By reference or value
  + Const reference -> can only call const functions in those functions
* Range based for loops!!