CS106X Course Syllabus

I don't even try to promise a day-by-day lecture schedule, since even the most disciplined and organized instructors have a difficult time staying on track, and I'm far outside the set of disciplined and organized instructors. I do, however, think it's reasonable to give a sense of what topics I'll be covering, and how much time we'll be dedicating to each of them.

Week 1: **Basic C++ syntax, control idioms, program structure, `strings`, and libraries.** *Beyond the introductory remarks, the week will feel like a transition course designed to get Java programmers to emulate their craft using C++ instead. The overarching lesson here is that C++, like Java, JavaScript, and many other programming languages, has* **`int`***s,* **`float`***s,* **`for`** *loops,* **`switch`** *statements,* **`return`** *values, functions, classes, and methods.*

Week 2: **Templates, abstract data types, containers, vectors, stacks, queues, sets, maps, scanners, and lexicons.** *You're already familiar with templates and containers, even if you didn't call them that when you took AP Java or CS106A. Containers are data structures that are designed to store other data structures, and you already have plenty of practice with them—specifically, the Java* **`ArrayList`** *and the* **`HashMap`***. C++—Stanford's version of it, anyway—antes up its own versions of the* **`ArrayList`** *and* **`HashMap`** *(the* **`Vector`** *and* **`Map`***, respectively). We'll invest a lot of energy teaching the new container classes and the metaphors you subscribe to when coding with them (a stack of cafeteria plates, a queue of customers at Harrods in London, etc.)*

Week 3: **Recursion, recursive backtracking, and memoization.** *Recursion didn't originate with computer science or programming. Mathematics gets all the credit there. But virtually all programming languages support recursion, which is a function's ability to either directly or eventually call itself. Many practical programming problems are inherently recursive, and the ability to code using recursion makes it easier to exploit their recursive structure. Recursive backtracking is a form of recursion where you recognize one particular recursive call turned out to be a dead end of sorts, but that some other recursive call could pan out with something useful. Memoization is a technique used when there are a finite number of recursive sub-problems that will otherwise be repeatedly called an exponential number of times unless some form of caching (aka memoization) is used.*

Week 4: **Memory, Memory Addresses, Pointers, and Dynamic Memory Allocation.** *One of the most powerful features of the C++ language—and admittedly, the feature that makes C++ a terribly difficult language to master—is that it grants* *you the ability to share the physical memory locations of your data with other functions and to manually manage dynamically allocated computer memory. We'll discuss a new type of variable—the pointer—that's designed to store the location of other figures in memory, whether those figures are integers, arrays,* ***`ArrayList`****s, records, or even other pointers. For the next several weeks, we focus less on application-focused programming and more so on the use of pointers and advanced memory management to implement all of the ADTs you've come to use and appreciate since Week 2.*

Week 5: **Linked Lists and Their Variations, Hash Tables and Trees.** *The majority of you are familiar with arrays and the* ***`Vector`****—so familiar, in fact, that you should recognize that insertion and deletion from the front can be a very time-consuming operation if the size of the array or* ***`Vector`*** *is large. The* ***`Queue`****'s* ***`dequeue`*** *operation, which is supposed to be fast regardless of queue length, couldn't possibly be backed by anything that's truly array-like. The linked list makes a few sacrifices in the name of fast insertion at and deletion from both the front and the back of the sequence. We'll see the linked list and a few of its variations as the most basic linked structure in the series of linked structures we learn about over the remainder of the course, and we'll understand why it (and not anything array-like) backs the* ***`Queue`****.*

Week 6: **Hash Tables and Trees.** *Once you're fluent in the construction and manipulation of the basic linked list, we'll be in a position to build and talk about more advanced linked structures like hash tables, binary search trees, tries, and skip lists. The hash table is the backbone of your* ***`Map`*** *container, the binary search tree is more or less the core of your* ***`Set`****, and the trie is a simplified version of what backs your* ***`Lexicon`****. (The skip list is a fairly recent randomized data structure that could back the* ***`Set`*** *if we opted for it over the BST.) We could spend 40 lectures talking about data structures. We'll try to get as much of those 40 lectures into the two or three we have.*

Week 7: **Graphs and Fundamental Graph Algorithms.** *The graph is the Holy Grail of all linked data structures, allowing us to model generic peer-to-peer systems like road, rail, and airline systems, computer intranets, social networks, and the WWW. There are a number of fascinating and fairly well-understood graph algorithms (Dijkstra's shortest path algorithm is the most important we'll study), as well as a number of other algorithms that we're not 100% convinced are the most efficient ones possible. We'll study as many of them as time permits, and without stealing the thunder of later theory classes, explain why some algorithms appear to be the best we can do even though they're exceptionally slow for large graphs.*

Week 8: **C++ Interfaces, Inheritance, and Class Hierarchies.** *There are a good number of scenarios where multiple classes are intentionally implemented to the same interface. Inheritance is a unique form of sub-typing and code sharing that* *recognizes common implementation patterns across multiple classes and works to unify them, so their public interfaces overlap as much as possible. You've already seen and benefited from inheritance to some degree if you've done any significant coding in Java, as all Java classes extend the core* ***`Object`*** *class, and therefore respond to a small core of methods like* ***`equals`*** *and* ***`hashcode`****. We'll extend that basic understanding, build collections of related classes that exhibit even more aggressive sharing of interface and implementation, and demonstrate how fundamental inheritance is to large, scalable, object-oriented systems.*

Week 9/10: I'll leave this unscheduled, since I suspect the majority of you will want to crank on your final projects without feeling an obligation to attend lecture and learn new material.