Assignment #2: ADT[1] Client Extravaganza

Inspiration credit to Joe Zachary, University of Utah (random writer).

Due Wed Jan 30th 2:15pm

You've been introduced to the handy CS106 class library, now it's time to put these objects to work! In the role of client with the low-level details abstracted away, you can put your energy toward solving more interesting problems. Your next assignment is to write two short client programs that heavily leverage our classes to do nifty things. The tasks may sound a little daunting at first, but given the power tools in your arsenal, each requires only a page or two of code. Let's hear it for abstraction!

The assignment has several purposes:

1. To more fully experience the joy of using pre-written classes. Most of the heavy-lifting is handled by objects from our class library.
2. To stress the notion of abstraction as a mechanism for managing data and providing functionality without revealing the representational details.
3. To become familiar with using C++ class templates.
4. To gain practice with classic data structures such as the stack, queue, vector, map, and set.

Part A— Random writing and Markov models of language

In the past few decades, computers have revolutionized student life. In addition to providing no end of entertainment and distractions, computers also have also facilitated much productive student work. However, one important area of student labor that has been painfully neglected is the task of filling up space in papers, Ph.D. dissertations, grant proposals, and recommendation letters with important sounding and somewhat sensible random sequences. Come on, you know the overworked TA/professor/reviewer doesn't have time to read too closely…

To address this burning need, the random writer is designed to produce somewhat sensible output by generalizing from patterns found in the input text. When you're coming up short on that 10-page paper due tomorrow, feed in the eight pages you already have written into the random writer, and voila! another couple of pages appear. You can even feed your own .cpp files back into your program and have it build a new random program on demand.

How does this work?

Random writing is based on an idea advanced by Claude Shannon in 1948 and subsequently popularized by A.K. Dewdney in his Scientific American column in 1989. Shannon's famous paper introduces the idea of a Markov model for English text. A *Markov model* is a statistical model that describes the future state of a system based on the current state and the conditional probabilities of the possible transitions. Markov models have a wide variety of uses, including recognition systems for handwriting and speech, machine learning, bioinformatics, even Google's PageRank algorithm has a Markov component to it. In the case of English text, the Markov model is used to describe the possibility of a particular character appearing given the sequence of characters seen so far. The sequence of characters within a body of text is clearly not just a random rearrangement of letters and the Markov model provides a way to discover the underlying patterns and, in this case, to use those patterns to generate new text that fits the model.

[1] ADT = "Abstract Data Type". This is the nerdy computer scientist term for types used with respect to their public interface, without knowledge of or dependence on any details of the underlying representation.

First, consider generating text in total randomness. In my house, I can round up a toddler, turn him loose on my keyboard where he's just as likely to hit any key as another, and get output of a truly random nature. As cute as a 2-year-old typing is, the gibberish produced makes pretty unconvincing English:

No model aowk fh4.s8an zp[q;1k ss4o2mai/

A simple improvement is to gather information on character frequency and use that to weight the probability of choosing. In English text, not all characters occur equally often. Better random text would mimic the expected character distribution. Read some input (say, *Tom Sawyer*) and count the character frequencies. You'd find that spaces are the most common, that the character 'e' is fairly common, and that the character 'q' is rather uncommon. Suppose that space characters represent 16% of all characters, 'e' just 9%, and 'q' a mere .04% of the total. Using these weights, you could produce random text that exhibited these same probabilities. It wouldn't have a lot in common with the real *Tom Sawyer*, but at least the characters would tend to occur in the proper proportions. In fact, here's an example of what you might produce:

Order 0 rla bsht eS ststofo hhfosdsdewno oe wee h .mr ae irii ela iad o r te u t mnyto onmalysnce, ifu en c fDwn oee iteo

This is an *order 0 Markov model*, which predicts that each character occurs with a fixed probability, independent of previous characters.

We're getting somewhere, but most events occur in context. Imagine randomly generating a year's worth of Fahrenheit temperature data. A series of 365 random integers between 0 and 100 wouldn't fool the average observer. It would be more credible to make today's temperature a random function of yesterday's temperature. If it is 85 degrees today, it is unlikely to be 15 degrees tomorrow. The same is true of English words: If this letter is a 'q', then the following letter is quite likely to be a 'u'. You could generate more realistic random text by choosing each character from the ones likely to follow its predecessor.

For this, process the input and build an order 1 model that determines the probability with which each character follows another character. It turns out that 's' is much more likely to be followed by 't' than 'y' and that 'q' is almost always followed by 'u'. You could now produce some randomly generated *Tom Sawyer* by picking a starting character and then choosing the character to follow according to the probabilities of what characters followed in the source text. Here's some output produced from an order 1 model:

Order 1 "Shand tucthiney m?" le ollds mind Theybooure He, he s whit Pereg lenigabo Jodind alllld ashanthe ainofevids tre lin--p asto oun theanthadomoere

This idea extends to longer sequences of characters. An order 2 model generates each character as a function of the two-character sequence preceding it. In English, the sequence "sh" is typically followed by the vowels, less frequently by 'r' and 'w', and rarely by other letters. An order 5 analysis of *Tom Sawyer* would reveal that "leave" is often followed by 's' or space but never 'j' or 'q' and that "Sawye" is always followed by 'r'. Using an order *k* model, you generate random output by choosing the next character based on the probabilities of what followed the previous *k* characters (the *seed*) in the input text.

At only a moderate level of analysis (say, orders 5 to 7), the randomly generated text begins to take on many of the characteristics of the source text. It probably won't make complete sense, but you'll be able to tell that it was derived from *Tom Sawyer* as opposed to, say, *Pride and Prejudice*. At even higher levels, the generated words tend to be valid and the sentences start to scan. Here are some more examples:

Order 2 "Yess been." for gothin, Tome oso; ing, in to weliss of an'te cle -- armit. Papper a comeasione, and smomenty, fropeck hinticer, sid, a was Tom, be suck tied. He sis tred a youck to themen

Order 4 en themself, Mr. Welshman, but him awoke, the balmy shore. I'll give him that he couple overy because in the slated snufflindeed structure's kind was rath. She said that the wound the door a fever eyes that WITH him.

Order 6 Come -- didn't stand it better judgment; His hands and bury it again, tramped herself! She'd never would be. He found her spite of anything the one was a prime feature sunset, and hit upon that of the forever.

Order 8 look-a-here -- I told you before, Joe. I've heard a pin drop. The stillness was complete, how- ever, this is awful crime, beyond the village was sufficient. He would be a good enough to get that night, Tom and Becky.

Order 10 you understanding that they don't come around in the cave should get the word "beauteous" was over-fondled, and that together" and decided that he might as we used to do -- it's nobby fun. I'll learn you."

A sketch of the random writer implementation

Your program is to read a source text, build an order *k* Markov model for it, and generate random output that follows the frequency patterns of the model.

First, you prompt the user for the name of a file to read for the source text and re-prompt as needed until you get a valid name. (And you have a beautiful and correct version you can crib from your solution to assignment #1, no?) Now ask the user for what order of Markov model to use (a number from 1 to 10). This will control what seed length you are working with.

Use simple character-by-character reading on the file. As you go, track the current seed and observe what follows it. Your goal is to record the frequency information in such a way that it will be easy to generate random text later without any complicated manipulations.

Once the reading is done, your program should output 2000 characters of random text generated from the model. For the initial seed, choose the sequence that appears most frequently in the source (e.g. if doing an order 4 analysis, the four-character sequence that is most often repeated in the source is used to start the random writing). If there are several sequences tied for most frequent, any of them can be used as the initial seed. Output the initial seed, then choose the next character based on the probabilities of what followed that seed in the source. Output that character, update the seed, and the process repeats until you have 2000 characters.

For example, consider an order 2 Markov model built from this input:

```
As Gregor Samsa awoke one morning from uneasy dreams he found himself
transformed in his bed into a gigantic insect.
```

Here is how the first few characters might be chosen:

- The most commonly occurring sequence is "in" which appears four times. It is the initial seed.
- The next character is chosen based on the probability that it follows the seed "in" in the source. The source contains four occurrences of "in", one followed by 'g', one followed by a space, one followed by 't', and one followed by 's'. Thus, there should be a 1/4 chance of choosing 'g', 1/4 chance of choosing space, a 1/4 chance of choosing 't', and a 1/4 chance of choosing 's'. Suppose space is chosen this time.
- The seed is updated to "n ". The source contains one occurrence of "n ", which is followed by 'h'. Thus the next character chosen is 'h'.

- The seed is now " h". The source contains three occurrences of " h", once followed by 'e', and twice followed by 'i'. Thus there is a 1/3 chance of choosing 'e' and 2/3 for 'i'. Imagine 'i' is chosen this time.
- The seed is now "hi". The source contains two occurrences of "hi", once followed by 'm', the other by 's'. For the next character, there is 1/2 chance of choosing 'm' and 1/2 chance for 's'.

If your program ever gets into a situation in which there are no characters to choose from (which can happen if the only occurrence of the current seed is at the exact end of the source), your program can just stop writing early.

A few implementation hints

Although it may sound daunting at first glance, this task is supremely manageable with the bag of power tools you bring to the job site.

- **`Map`** and **`Vector`** are just what you need to store the model information. The keys into the map are the possible seeds (e.g. if order is 2, each key is a 2-character sequence found in the source text). The associated value is a vector of all the characters that follow that seed in the source text. That vector can, and likely will, contain a lot of duplicate entries. Duplicates represent higher probability transitions from this Markov state. Explicitly storing duplicates is the easiest strategy and makes it simple to choose a random character from the correct frequency distribution. A more space-efficient strategy would store each character at most once, with its frequency count. However, it's a bit more awkward to code this way. You are welcome to do either, but if you choose the latter, please take extra care to keep the code clean.
- Determining which seed(s) occurs most frequently in the source can be done by iterating or mapping over the entries once you have finished the analysis.
- For reading a file one character at a time, use the **`get`** member function for **`ifstream`**.

Random writer task breakdown

A suggested plan of attack that breaks the problem into the manageable phases with verifiable milestones:

- *Task 1— Try out the demo program*. Play with the demo just for fun and to see how it works from a user's perspective.
- *Task 2— Review our provided classes*. It's Vector and Map you need. Once you're familiar with what functionality our classes provide, you're in a better position to figure out what you'll need to do for yourself.
- *Task 3— Design your data structure*. Think through the problem and map out how you will store the analysis results. It is vital that you understand how to construct the nested arrangement of string/vector/map objects that will properly represent the information. Since the Map will contain Vectors, a nested template type is in your future.
- *Task 4— Implement analyzing source text*. Implement the reading phase. Be sure to develop and test incrementally. Work with small inputs first. Verify your analysis and model storage is correct before you move on. There's no point in trying to generate random sentences if you don't even have the data read correctly!
- *Task 5— Implement random generation*. Now you're ready to randomly generate. Since all the hard work was done in the analysis step, generating the random results is straight-forward.

You can run the random writer program on any sort of input text, in any language! The web is a great place to find an endless variety of input material (blogs, slashdot, articles, etc.) When you're ready for a large test case, Project Gutenberg maintains a library of thousands of full-length public-domain books. At higher orders of analysis, the results it produces can be surprisingly sensible and

often amusing. When your program generates something particularly entertaining, send it to me to share with the class.

References

A.K. Dewdney. A potpourri of programmed prose and prosody. *Scientific American,* 122-TK, June 1989.

C.E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27, 1948.

http://en.wikipedia.org/wiki/Markov_chain Wikipedia entry on Markov models

http://www.gutenberg.org Project Gutenberg, public domain e-books

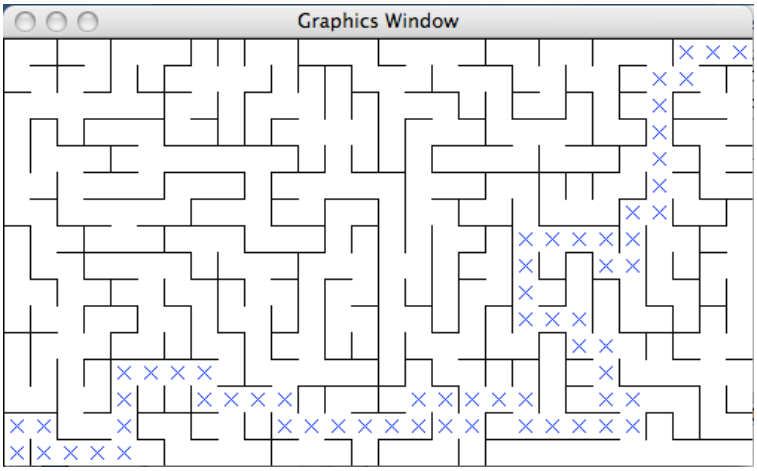
Part B — Mazes

Next up, you will deploy a whole slew of container classes (Stack, Queue, Vector, Set, and more) and our simple Maze class (itself built on Grid) in writing a program to generate and solve mazes.

Labyrinths and mazes have fascinated humans since ancient times (remember Theseus and the Minotaur?) and so it comes as no surprise that there are many known approaches to generating and solving mazes. What you may not realize is that it is more than just fun and games. Many of the same fundamental issues arise in tasks such as designing circuit boards, routing network traffic, motion planning, social networking, and graph algorithms, making this an active area of research, even today.

For this program, you will write a program to generate and solve a maze. We will focus on a particular type of maze known as a *perfect* maze. A perfect, or simply-connected, maze has no loops and no inaccessible areas. There is exactly one path connecting any pair of points in a perfect maze. You can arbitrarily designate a start and goal point and are guaranteed one solution path exists.

Here is a sample run showing a perfect maze and its solution:



Generating a perfect maze

Your first task is to generate a perfect maze. There are a wide variety of algorithms for maze generation. The clever, but simple, algorithm we will use is attributed to D. Aldous and A. Z. Broder, who each independently came up with it. The maze is initially configured with all walls intact; each cell in its own enclosed room. The Aldous-Broder algorithm takes a random walk around the maze, moving around from neighbor to neighbor and removing walls where appropriate to connect all the cells into a perfect maze.

Here is a bit more detail. All cells start "excluded" from the maze. Choose a cell at random and mark it as "included" in the maze. This cell is the current point.

1. Randomly choose one of the neighbors of the current point
2. If neighbor is "excluded", remove wall between current point and neighbor and mark neighbor as "included"
3. Neighbor becomes current point (whether or not wall was removed)
4. Repeat steps 1-3 until all cells in the maze have been included

Aldous-Broder isn't the fastest maze generation algorithm (toward the end, when fewer excluded cells remain, it may take many iterations for the random walk to eventually reach them and include them) but it's simple to understand and implement and produces lovely perfect mazes.

We provide some help in the form of a **`Maze`** class that internally uses a Grid to manage the cells and walls. The Maze class provides functionality to set and access wall information as well as draw the maze in the graphics window. Read the header file for more information on its features. You will need some additional data storage beyond the maze while you are generating it. For example, what container might you use to track which cells are currently included? Would using a container help when randomly choosing a neighbor? Keep in my mind you have access to the full arsenal (Vector, Stack, Set, Grid, etc.) and may find it handy to use more than one.

Solving a perfect maze using bread-first search

Just as with generating a maze, there are numerous algorithms for solving one. Solving a maze can be seen as a specific instance of a *shortest path* problem, where the challenge is to find the shortest route from the start to the goal. Shortest path problems come up in a variety of situations such as packet routing, robot motion planning, social networks, studying gene mutations, and more. In the case of the perfect maze, the shortest path is also the only path, but the general process is the same regardless.

One classic path-finding algorithm is *breadth-first search*. A breadth-first search reaches outward from the start in a radial fashion until it hits the goal. For the maze, this means first examining those paths that take one step away from the start. If any of these reach the destination, you're done. If not, the search now examines all paths that add one more step. The search expands radially, examining all paths of length three, then of length four, etc.) until it finds the goal.

Breadth-first search is typically implemented using a queue. The queue is used to store partial paths that represent possibilities to explore. The paths are enqueued in order of increasing length. The first elements enqueued are all one-step paths, followed by the two-step paths, and so on. Due to FIFO handling, paths will be dequeued in order of increasing length. The algorithm operates by dequeueing the front path and determining if it reaches the goal. If it does, you have a complete path, and it is the shortest. If not, you take that partial path and extend it to reach points that are one more step away, and enqueue those extended paths to be examined later. To represent each path, the stack is a natural choice, since you typically add/examine only the end of the path, which is easily accessible at the top of the stack. For simplicity, your program can assume the start location is always the lower-left corner, and the goal is always the upper-right corner.

Here are the steps of the breadth-first search. It works on a queue of paths (stack objects).

1. Create path with just start location and enqueue it
2. Dequeue shortest path
3. If path ends at goal you're done, otherwise
   a. For each neighbor accessible from path end, make copy of path, extend by adding neighbor and enqueue it
4. Repeat steps 2-3 until at goal (also stop if queue is empty, won't happen for perfect maze)

One issue that is a bit subtle is that you need to take care to avoid circular paths. A perfect maze doesn't have cycles, but if you're not watching you can accidentally double-back within a path. For example, if the path leads from point 0-0 to 1-0 you should not extend the path by moving back to 0-0. There are various ways you can watch for this, if you think it through, I'm sure you'll come up with a strategy. Some of approaches uses—surprise!—yet another container!

Once you have found the path, visually mark the solution on the graphics window to trace the steps from the start to the goal. Now offer the user a chance to start over, generating and solving more mazes until they are done.

A few implementation hints

It's all about leveraging the class library — you'll find your job is mostly coordinating objects.

- The ever handy **`Vector`** can be used for any sort of list that comes up in your strategy. A **`Set`** is particularly handy if you need to be able to quickly determine membership (is this element contained in the set?). Remember that if a Set is storing elements of user-defined type (which cannot be compared using the built-in relational operators), you will need to write the appropriate comparison callback function for that type and pass it to the Set constructor.
- For solving the maze, the **`Stack`** with its LIFO behavior is ideal for storing a path. The first entry pushed on the stack is the starting point and each subsequent point in the path is pushed on top. The FIFO **`Queue`** is just what's needed to track those partial paths under consideration. The paths are enqueued (and thus dequeued) in order of increasing length.
- We also supply **`Maze`**, a special-purpose class for managing the maze configuration (walls). Read the interface file **`maze.h`** in the starter project for details on how to create a new maze, draw it, change wall s, check if a wall exists, and so forth. Internally, the maze is implemented using a Grid. Examine the code in **`maze.cpp`** if you're curious to learn more.
- When you need to make a copy of a stack (or any container for that matter), remember that the assignment operator works as expected for all our container classes. Assigning one stack to another will create a new stack with the same contents.

Maze task breakdown

Amazingly (I couldn't resist!), this program requires just over a page of code, but like all programming tasks, it benefits from a step-by-step development plan to keep things moving along smoothly.

- *Task 1— Try out the demo program*. Play with the demo just for fun and to see how it works from a user's perspective.
- *Task 2— Get familiar with our provided classes*. You've seen the containers in lecture, but Maze is new to you. Reading the header file comments will acquaint you with its features.
- *Task 3— Conceptualize generation algorithm*. Review our description of Aldous-Broder. Think carefully about what data you need during the generation and how you will store and access it.
- *Task 4— Generate maze*. Code up the algorithm and use the drawing features of the Maze class to animate the progress of the algorithm. Visually verify the resulting meet the criteria for perfect mazes.
- *Task 5— Conceptualize solve algorithm*. Be sure you understand the breadth-first algorithm and what objects you will be using. Note that since the items in the Queue are Stacks, you have a nested template here— be careful!
- *Task 6— Solve maze*. Now you're ready for the meaty part. The code is not long, but it is dense and all those templates will conspire to trip you up. We recommend generating small mazes to

make it easier for you to test and trace your algorithm while you are in development. Once it works in the small, scale up to bigger mazes.

There are many interesting facets to mazes and much fascinating mathematics underlying them. Mazes will come up again several times this quarter. Chapter 6 of the reader develops a different path-finding algorithm called depth-first search, implemented using recursion. At the end of the quarter when we talk about graphs, we'll explore the equivalence between mazes and graphs and note how many of the interesting results in mazes are actually based on work done in terms of graph theory.

References

A. Aldous. The random walk construction for spanning trees. *SIAM J. Discrete Math,* 1990.

A.Z. Broder. Generating random spanning trees. *30thIEEE Symp Foundations of Computer Science*, 1989.

http://www.astrolog.org/labyrnth/algrithm.htm Very complete page on mazes and maze algorithms

Requirements and hints for all programs

- The programs will be a hybrid of the procedural and object-oriented paradigms. You will be creating and messaging lots of objects, but will write your code in the procedural style, starting with the main function and decomposing into ordinary global functions.
- Be on your toes about making sure that your template types are always properly specialized and that all types match. Using **`Vector`** without specialization just won't fly, and a **`Vector<int>`** is not the same thing as a **`Vector<double>`**. The error messages you receive when you have mismatches can be cryptic and hard to interpret. Look carefully at the types and see if you can spot where they don't quite match. Also remember that a nested template type requires a space between the two closing angle bracket, e.g. **`Vector<Stack<int> >`**. Check our new course wiki for help and advice on compiler issues and please use it to share your own lessons learned with the rest of the class.
- You can run our demo program to observe the expected behavior and/or compare your results with ours. Our code doesn't do anything tricky or special, it uses the same classes and algorithms suggested in the handout. Students sometimes notice that their program runs more slowly than the demo. If you're generally in the same ballpark, this is nothing to worry about but a larger discrepancy could suggest that your code has issues such as using an incorrect algorithm, computing things unnecessarily, doing redundant work, or passing large objects by value instead of by reference.
- Our programs average 80 lines of code (not including whitespace and comments), decomposed into about 7-10 functions of 5-10 lines each.
- Since you have two different programs to write, you might want to create separate projects, one for each. You can re-use the same project, but it requires some extra steps. A project cannot contain more than one file that declares a **`main`** function, so you will need to either remove the previous file (ie remove **`randomwriter.cpp`** before adding **`mazework.cpp`**), temporarily rename the unused **`main`** function(s) in the program you're not working on, or patch the programs together with some sort of über-main that can invoke the appropriate one by request.

Extensions

We design the assignments to be fairly challenging, but we know a few of you may not be content to stop there. Be sure to first concentrate your energy on doing an excellent job on the standard requirements but if you've nailed those parts and want to go deeper, we'd love to have you explore ways to extend the assignment. It's your chance to challenge yourself further, use your creativity, and earn some CS106 Gold Stars of Valour. Below we give some ideas for possible explorations; you're also free to strike out in your own direction.

If you do attempt some fancy features, please be sure they don't interfere with your section leader's ability to test the standard portion of the assignment. If necessary, you can turn in separate programs if it's not possible for the standard and extended versions to peacefully co-exist.

- Random writing could work off word patterns, instead of characters. An order *k* word model would report the probability of a word following the previous sequence of *k* words. A word-based model tends to produce more sensible sounding things, even at low orders, but the results aren't quite as creative since it tends to reproduce large sequences lifted from the source text. If your original program is well-written, this can be a rather small change.
- A Markov model can also be used in the other direction, to recognize an author by his or her characteristic patterns. This requires building multiple Markov models, one for each candidate author, and comparing them to an unattributed text to find the best match. This sort of "literary forensic analysis" has been used to try to determine the correct attribution for texts where the authorship is unknown or disputed, as in the unveiling of the author of the anonymous bestseller *Primary* Colors or fingering who is behind the "fake Steve Jobs" blog. Markov analysis can also be used in plagiarism detection.
- If you have way too much time on your hands: consider Markov models for images. A texture synthesis algorithm can fill holes within images by generating replacement pixels that "match" the surrounding context using a Markov model (this is the basis for the standard algorithm used by Photoshop to remove elements). The algorithm creates amazingly realistic continuations of a scene—ocean waves, clouds, forests, pebbles, etc. The idea is to match existing pixels in a neighborhood of a target pixel with tiles from elsewhere in the same image. To learn more about this, check out this nifty 1999 paper by Efros and Leung on the web at `http://graphics.cs.cmu.edu/people/efros/research/EfrosLeung.html`.
- Try out other maze generation algorithms. How do they compare to Aldous-Broder in terms of coding ease? Are they more or less efficient at runtime? Does they result in different kind of mazes? Here are a few names to get you started: depth-first, growing tree, algorithms named for their inventor: Eller, Prim, Kruskal, Wilson, and many others.
- Try out other maze solving algorithms. How does BFS stack up against the options in terms of solving power or runtime efficiency? Take a look at random mouse, wall following, depth-first search (either manually using a Stack or using recursion once you get the hang of it), Bellman-Ford, or others.
- There are many other neat maze-based games out there that make fun extensions. You might gather ideas from Robert Abbott's nifty `http://www.logicmazes.com/` or the Mathematical Association's `http://www.maa.org/editorial/mathgames/mathgames_11_24_03.html`.

Accessing files http://see.stanford.edu/see/materials/icspacs106b/assignments.aspx

On the class web site, you'll find a starter folder containing these files:

maze.cpp/h	Interface/implementation for Maze class (**add .cpp to project**)
ADTDemo	Our demo application (both programs combined)

To get started, create your own starter project and add the necessary .cpp files.

Deliverables

As always, you should turn in both a paper copy and an electronic submission of your files. Both are due before the beginning of lecture on the due date. Please refer to our handout on electronic submission for the details on e-submit. You only need submit/print those files you modified.