Assignment 2: ADT Client Applications

Inspiration credit to Joe Zachary from University of Utah (random writer) and Owen Astraction from Duke University (word ladder). Jerry developed the maze generator assignment.

Now that you've been introduced to the 106 container classes, it's time to put these objects to use. In the role of client, the low-level details have already been dealt with and you can focus your attention on solving more interesting problems. Having a library of well-designed and debugged classes vastly extends the range of tasks you can easily take on. Your next assignment has you write three short client programs that heavily leverage the standard classes to do nifty things. The tasks may sound a little daunting at first, but given the power tools in your arsenal, each requires less than a hundred lines of code. Let's hear it for abstraction!

The assignment has several purposes:

1. To more fully explore the notion of using objects.
2. To stress the notion of abstraction as a mechanism for managing data and providing functionality without revealing the representational details.
3. To become more familiar with C++ class templates.
4. To gain practice with classic data structures such as the queue, vector, set, lexicon, and map.

Due: Wednesday, October 12th at 5:00 p.m.

Part I: 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, letters to mom, grant proposals, statements of purpose, and the like with important sounding and somewhat sensible random sequences.

You are to take your newly acquired knowledge of data abstraction and put our **`Vector`** and **`Map`** classes to work in constructing a program that addresses this burning need. The random writer is a marvelous bit of technology that generates 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 and voila: another two pages comes right up. Sheesh, you can even feed your own **`.cpp`** files back into your program and have it build you 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 (handwriting, speech, etc), 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 quite obviously not just any 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.

An *order 0 Markov model* predicts that each character occurs with a fixed probability, independent of previous characters. Imagine taking a book (say, *Tom Sawyer*) and counting 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. The order 0 Markov model reports that space characters represent 16% of all characters, 'e' just 9%, and 'q' a mere .04% of the total. Using this model, 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

Now imagine doing a slightly more sophisticated order 1 analysis 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 an example:

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

Now extend this to an order *k* Markov model that determines the probability with which each character follows a sequence of *k* characters. 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'd be able to produce

random *Tom Sawyer* 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*. 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, choose any one you want. 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 value will be a vector of all the characters that follow seed in the source text. That vector can, and likely will, contain a lot of duplicate entries. Those duplicates represent the higher probability transitions from this Markov state. This is the easiest strategy and makes it simple to choose a random character when needed. 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, check out the **`get`** member function for **`ifstream`** which is discussed on page 3-26 in the reader. In particular, don't use the **`Scanner`** class, even though you'll be tempted to, being this assignment is about containers and all.

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— Become familiar with our provided classes.* Be sure you have a solid understanding of the provided classes. Carefully read over the interfaces so you understand how to be a proper client of these classes. Once you know what functionality our classes provide, you'll be 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. Don't shortchange this step! 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`** contains **`Vector`**s, you will use a nested template type— careful with the syntax!
- *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 should be pretty straightforward. Pat yourself on the back! You have successful completed your first mission as a client of the 106 classes!

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 appropriate and surprisingly 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 II: Word ladders

Leveraging the vector, queue, and lexicon abstractions, you'll find yourself well equipped to write a program to build word ladders. A word ladder is a connection from one word to another formed by changing one letter at a time with the constraint that at each step the sequence of letters still forms a valid word. For example, here is a word ladder connecting "code" to "data".

code → cade → cate → date → data

You will ask the user to enter a start and a destination word and then your program is to find a word ladder between them if one exists. By using an algorithm known as breadth-first search, you are guaranteed to find the shortest such sequence. The user can continue to request other word ladders until they are done.

Here is some sample output of the word ladder program:

```
Enter start word (RETURN to quit): work
Enter destination word: play
Found ladder: work fork form foam flam flay play

Enter start word (RETURN to quit): awake
Enter destination word: sleep
Found ladder: awake aware sware share shire shirr shier sheer sheep sleep

Enter start word (RETURN to quit): airplane
Enter destination word: tricycle
No ladder found.
```

A sketch of the word ladder implementation

Finding a word ladder is a specific instance of a *shortest path* problem, where the challenge is to find the shortest path from a starting position to a 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. One approach for finding a shortest path is the classic algorithm known as *breadth-first search*. A breadth-first search searches outward from the start in a radial fashion until it hits the goal. For word ladder, this means first examining those ladders that represent "one hop" (i.e. one changed letter) from the start. If any of these reach the destination, we're done. If not, the search now examines all ladders that add one more hop (i.e. two changed letters). By expanding the search at each step, all one-hop ladders are examined before two-hops, and three-hop ladders only considered if none of the one-hop nor two-hop ladders worked out, thus the algorithm is guaranteed to find the shortest successful ladder.

Breadth-first is typically implemented using a queue. The queue is used to store partial ladders that represent possibilities to explore. The ladders are enqueued in order of

increasing length. The first elements enqueued are all the one-hop ladders, followed by the two-hop ladders, and so on. Due to FIFO handling, ladders will be dequeued in order of increasing length. The algorithm operates by dequeueing the front ladder from the queue and determining if it reaches the goal. If it does, you have a complete ladder, and it is the shortest. If not, you take that partial ladder and extend it to reach words that are one more hop away, and enqueue those extended ladders onto the queue to be examined later. If you exhaust the queue of possibilities without having found a completed ladder, you can conclude that no ladder exists.

Let's make the algorithm a bit more concrete with some pseudo-code:

```
create initial ladder (just start word) and enqueue it
   while queue is not empty
      dequeue first ladder from queue (this is shortest partial ladder)
      if top word of this ladder is the destination word
         return completed ladder
      else for each word in lexicon that differs by one char from top word
           and has not already been used in some other ladder
               create copy of partial ladder
               extend this ladder by pushing new word on top
               enqueue this ladder at end of queue
```

A few of these tasks deserve a bit more explanation. For example, you will need to find all the words that differ by one letter from a given word. A simple loop can change the first letter to each of the other letters in the alphabet and ask the lexicon if that transformation results in a valid word. Repeat that for each letter position in the given word and you will have discovered all the words that are one letter away.

Another issue that is a bit subtle is the restriction that you not re-use words that have been included in a previous ladder. This is an optimization that avoids exploring redundant paths. For example, if you have previously tried the ladder **cat→cot→cog** and are now processing **cat→cot→con**, you would find the word **cog** one letter away from **con**, so looks like a potential candidate to extend this ladder. However, **cog** has already been reached in an earlier (and thus shorter) ladder, and there is no point in re-considering it in a longer ladder. The simplest way to enforce this is to keep track of the words that have been used in any ladder (using yet another lexicon!) and ignore those words when they come up again. This technique is also necessary to avoid getting trapped in an infinite loop building a circular ladder such as **cat→cot→cog→bog→bat→cat**.

Since you need linear access to all of the items in a word ladder when time comes to print it, it makes sense to model a word ladder using a **Vector<string>**. And remember that you can make a copy of a **Vector<string>** by just assigning it to be equal to another via traditional assignment (i.e. **Vector<string> wordLadderClone = wordLadder**).

A few implementation hints

Again, it's all about leveraging the class libraries—you'll find your job is just to coordinate the activities of various objects to do the search.

- The linear, random-access collection managed by a **`Vector`** is ideal for storing a word ladder.
- A **`Queue`** object is a FIFO collection that is just what's needed to track those partial ladders under consideration. The ladders are enqueued (and thus dequeued) in order of length so as to find the shortest option first.
- As a minor detail, it doesn't matter if the start and destination word are contained in the lexicon or not. You can eliminate non-words at the get-go if you like, or just allow them to fall through and be searched anyway.

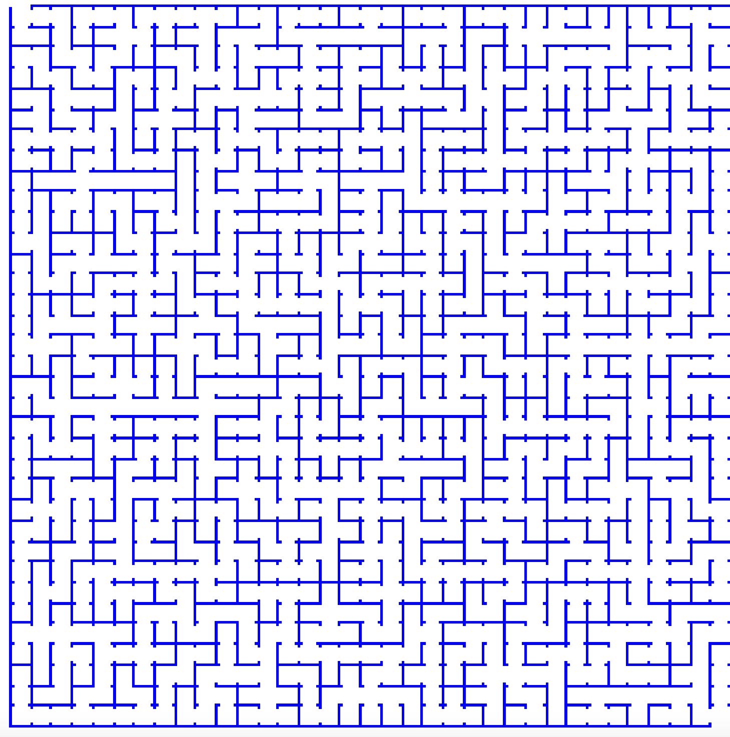
Word ladder task breakdown

This program requires just over a page of code, but it still benefits from a step-by-step development plan to keep things moving along.

- *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 **`Vector`** and **`Queue`** in lecture, but **`Lexicon`** is new to you. The reader outlines everything you need to know about the **`Lexicon`**, so make sure you skim over Chapter 4 and the part about how to use the **`Lexicon`** class (it's really very easy.)
- *Task 3— Conceptualize algorithm and design your data structure.* Be sure you understand the breadth-first algorithm and what the various data types you will be using. Note that since the items in the **`Queue`** are **`Vector`**s, you have a nested template here— be careful!
- *Task 4— Dictionary handling.* Set up a **`Lexicon`** object with the large dictionary read from our data file. Write a function that will iteratively construct strings that are one letter different from a given word and run them by the dictionary to determine which strings are words. Why not add some testing code that lets the user enter a word and prints a list of all words that are one letter different so you can verify this is working?
- *Task 5— Implement breadth-first search.* 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 writing some test code to set up a small dictionary (with just ten or so words) to make it easier for you to test and trace your algorithm while you are in development. Do your stress tests using the large dictionary only after you know it works in the small test environment.

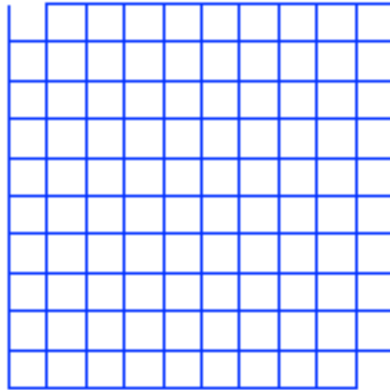
Part III: Generating Mazes

For the final task, you're to write a program that generates a bona fide maze. Here's an example of one your program might generate:

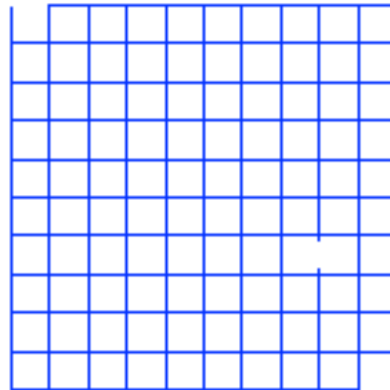


The goal is to create an **`n`** by **`n`** maze such that there's one unique path from the upper left corner to the lower right one. Our algorithm for generating such mazes is remarkably simple to explain, and given the services of the **`Set`** and the **`Vector`**, is almost as easy to implement. (By the way, it's not really our algorithm—it's a simplified version of Kruskal's minimal spanning tree algorithm, which we'll more formally discuss later on when we talk about graphs.)

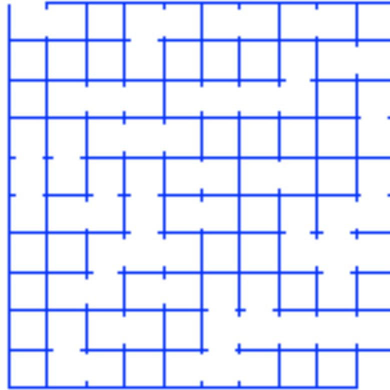
The basic idea has you start with a maze where all possible walls are present, as with:



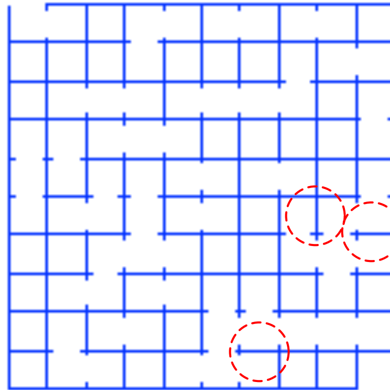
(Our example here happens to involve a 10 x 10 grid, but it generalizes to any dimension.) All walls are present, and the grid is divided into 10^2 = 100 **chambers**. Each iteration of the algorithm considers a randomly selected wall that's not been considered before, and removes that wall if and only if it separates two chambers. The first randomly selected wall can always be removed, because all locations are initially their own chambers. That first wall might be the one that's now missing in the following:



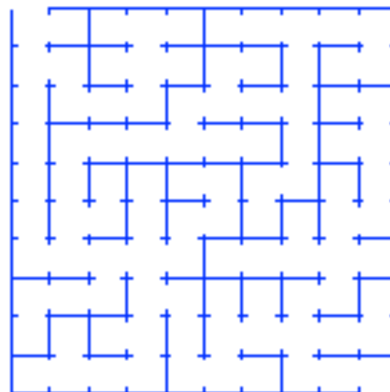
After a several more iterations, you'll see a good number of randomly selected walls have been removed:



The working maze pictured above includes at a few walls that should not be removed, because the cells on either side of each of them are part of the same chamber. I've gone ahead and circled some of them here (save for the circles, it's the same picture you see above):



All walls are considered exactly once in some random order, and if as you consider each wall you notice that it separates two different chambers, you merge the two into one by removing the wall. Initially, there are 100 chambers, so eventually 99 walls are erased to leave one big, scary maze, like this:



Notice the three walls I circled in the previous snapshot are still present. We can't tell from the photos when they were considered, but we shouldn't be surprised they're still there in the final maze.

Maze Generation Pseduo-code

Here is a high-level description of a reasonable solution:

```
choose a dimension
generate all walls, and shuffle them
generate all chambers, where each chamber includes one unique cell
foreach wall in randomly-ordered-walls
   if wall separates two chambers
      remove wall
```

We're going to let you tackle this one all by yourself, relying on the pictures and the pseudo-code above to get through it. Here are a few other details worth mentioning:

- All of the graphics routines are documented in **`maze-graphics.h`**. Everything provided is pretty straightforward, but if you want to make changes to these files, then go for it.
- There's a small header file called **`maze-types.h`**, which defines a **`cell`** type (which uses some advanced C++ so that you can build **`Set`**s of them without having to define a comparator, and a **`wall`** type, which helps describe a wall. You should use these definitions.
- Our solution uses the **`Set`** (to help model a chamber) and the **`Vector`** (to maintain an ordered list of walls and chambers). In particular, we didn't use a **`Grid`** at all, so you shouldn't be worried if you don't use one either.

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 as a procedural algorithm, starting with the **`main`** function and decomposed into **`ordinary`** top-level functions. This should be familiar to you from The Game of Life, but this assignment has more object orientation going on.
- Work extra hard to ensure 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 farcially mysterious 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<Vector<int> >`**. If you're stuck on some template syntax, bring your code by the Lair and we can help you sort it out.