

Assignment 2: ADT Client Applications

Inspiration credit to Joe Zachary, University of Utah (random writer) and Owen Astraction, Duke University (word ladder).

Now that you've been introduced to the most excellent 106 class library, 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 take on. Your next assignment is to write two 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 only about 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 familiar with using C++ class templates.
4. To gain practice with classic data structures such as the stack, queue, vector, and map.

Due: Friday, February 1st at 1:15 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 ststfo hhfosdsdewno oe wee h .mr ae irii ela iad o r te u t mnyto onmalysnce,
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 snuffindeed 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.

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'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. 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 Vectors, 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 successfully 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 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 II: Word ladders

Leveraging the stack and queue, and with the addition of our lexicon class, 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 radially 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 dequeuing 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 pseudocode:

```

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 additions to the ladder always occur at one end, a stack is a natural abstraction to represent a word ladder. The elements in the stack are simply strings. When you need to make a copy of a stack, remember that the assignment operator works as expected for

all our container classes, and thus just assigning one stack to another will create a new stack with the same contents.

Note that breadth-first search is not the most efficient algorithm for generating minimal word ladders. As the lengths of the partial word ladders increase, the size of the queue grows exponentially, leading to exorbitant memory usage when the ladder length is long and tying up your computer for quite a while examining them all. Later this quarter, we will touch on improved search algorithms, and in advanced courses such as CS161 you will learn even more efficient alternatives.

A few implementation hints

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

- The LIFO collection managed by a `stack` object is ideal for storing a word ladder. The first entry pushed on the stack is the starting word and each subsequent word is pushed on top.
- 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.
- For this program, we also supply `Lexicon`, a special-purpose class for storing a word list. It will be useful for the dictionary of English words as well as tracking those words that have already been examined to avoid redundantly processing them. Read the interface file `lexicon.h` in the starter project for details on how to create a new lexicon, add words to it, check if a word exists, and so forth. Behind the scenes, the lexicon employs a scarily complex representation that is extremely efficient and compact to give you blindingly fast access to a list of over 100,000 words. However you don't need to know any of that in order to use it!
- 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 amazingly just over a page of code, but it still 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 `stack` and `queue` in lecture, but `Lexicon` is new to you. Reading the comments in the header files before you start will help you get oriented in how to use these classes correctly.
- 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 `stacks`, 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.

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.
- 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> >`. If you're stuck on some template syntax, bring your code by the Lair and we can help you sort it out.
- Since you have two programs to write, it's probably easiest to just create two 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 (i.e. remove `randomwriter.cpp` before adding `ladder.cpp`) or temporarily rename the unused `main` function(s) in the program you're not working on.

Extensions

We plan our assignments to be fairly challenging, but we know a few of you may not be content to stop there. We want you to first concentrate your energy on doing an excellent job on the standard requirements! 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 bump up your grade. 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 coexist.

- 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.
- 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 recent unveiling of the author of the anonymous bestseller *Primary Colors*. It has also been 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>

- Word ladder is fun to play around with to learn about the patterns within English words. Can you change the algorithm to print all tied-for-shortest ladders instead of just the first ladder found? What word-pairs have a lot of different possible shortest ladders? How could you write an algorithm to find the longest ladder within the entire lexicon? Are there "lonely" words that participate in no ladders? What other interesting properties can you discover about word ladders?

Accessing files

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

lexicon.cpp/h

Interface/implementation for Lexicon (add .cpp files to project)

lexicon.dat	Large lexicon word file in binary format (keep in project folder)
RWDemo	Our random writer demo application
LadderDemo	Our word ladder demo application

To get started, create your own starter project and add the `.CPP` files for the non-template classes.

Deliverables

As always, you are to submit both a printed version of your code as well as an electronic version via ftp. Both the electronic and paper submissions should be handed in by the 1:15 p.m. deadline on Friday. You only need to submit/print the source for the files you modified. Please firmly staple all the pages and mark it clearly with your name and your section leader's name. (SCPD students need only e-submit.)