Assignment 2: ADT Client Applications

Inspiration credit goes out to Mike Cleron from Google (random sentence generator) and Owen Astrachan from Duke University (word ladder). Jerry developed the maze generator assignment. Assignment handout by Julie Zelenski and Jerry Cain.

Now that you've taken in the CS106 container classes, it's time to put these objects to use. In your role as client, the low-level implementation details have been dealt with and locked away as top secret so you can focus your attention on solving more interesting problems. Having a library of well-designed and debugged classes significantly extends the range of tasks you can easily take on. Your next assignment has you write three short client programs that leverage the container classes to do great 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 idea of using objects.
2. To stress the notion of abstraction as a mechanism for managing data and providing functionality without revealing internal representations.
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: Monday, January 25th at 5:00 p.m.

Part I: Word Ladders [Prose by Julie Zelenski]

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 she is done.

Here are some sample output of the word ladder program:

```
Welcome to the CS106 word ladder application!

Please enter the source word [return to quit]: work
Please enter the destination word [return to quit]: play
Found ladder: work fork form foam flam flay play

Please enter the source word [return to quit]: sleep
Please enter the destination word [return to quit]: awake
Found ladder: sleep sheep sheen shewn shawn sharn share sware aware awake

Please enter the source word [return to quit]: angel
Please enter the destination word [return to quit]: devil
Found ladder: angel anger agger egger eager lager leger lever level devel devil
```

Implementation Overview

Finding a word ladder is a specific instance of a *shortest path* problem, where the challenge is to find the shortest path from source to target. Shortest path problems come up in a variety of situations—packet routing, robot motion planning, social networks, gene mutation, and travel. One approach for finding a shortest path uses a classic algorithm known as *breadth-first search*. A breadth-first search stretches outward from the start in a radial fashion until it reaches some goal. For our word ladder problem, 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, then woo! 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 are only considered if none of the one-hop or two-hop ladders worked out, and so forth, so that the algorithm is guaranteed to find the shortest one.

Breadth-first is typically implemented using a queue. The queue is used to store partial ladders that represent the remaining possibilities worth exploring. 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 the FIFO nature of the queue, ladders will be dequeued in order of increasing length. The algorithm operates by dequeueing the front ladder and determining if it's a solution. 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 ever finding a word ladder, you can assume no such 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. 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 restriction: you should not reuse words that have been included in a previously generated partial 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** to be one letter away from **con**, so it looks like a potential candidate to extend this ladder. However, **cog** has already been reached in an earlier (and thus shorter) ladder, so there's no reason to consider it a second time. 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 by 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 (e.g. **Vector<string> clone = original**).

A few implementation hints

It's all about leveraging the class libraries. You'll find your job is 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. 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. The sample application requires that the endpoints be English

words, but you can let them be anything you want if you'd rather be more flexible. (The connectors need to be legitimate English words, of course.)

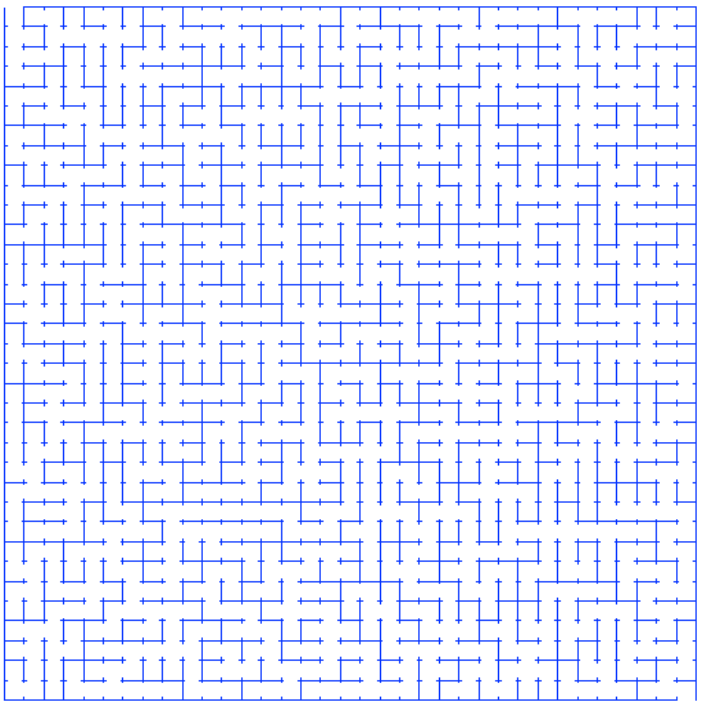
Word ladder task breakdown

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 5 and the part about how to use the **`Lexicon`** class (it's really very easy).
- *Task 3— Conceptualize the algorithm and design your data structure*. Be sure you understand the breadth-first algorithm and 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. Deep!
- *Task 4— Dictionary handling.* Set up a **`Lexicon`** 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.

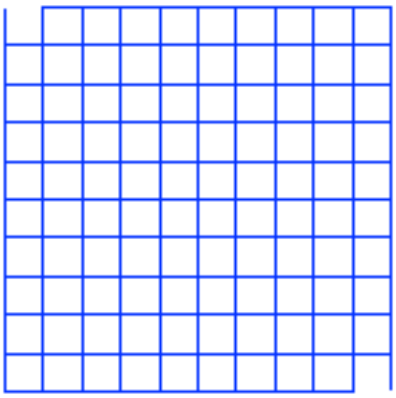
Part II: Generating Mazes [prose by Jerry]

Next, you're to write a program that animates the construction of a bona fide maze. Here's an example of one your program might generate:



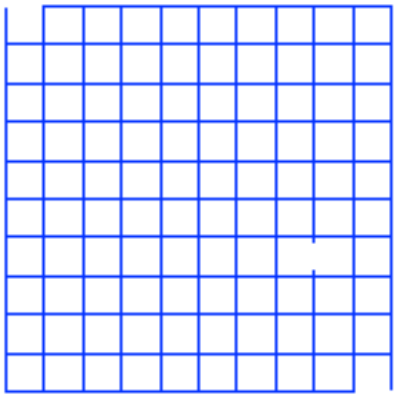
The goal is to create an **`n`** by **`n`** maze so there's one unique path from the upper left corner to the lower right one. Our algorithm for generating such mazes is remarkably simple to explain, and given the services of the **`Set`** and the **`Vector`**, is almost as easy to implement. (By the 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:

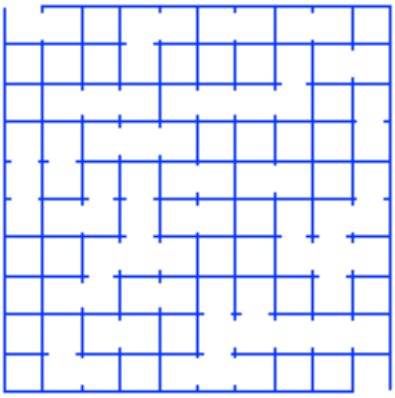


(Our example here happens to involve a 10 x 10 grid, but it generalizes to any dimension.)

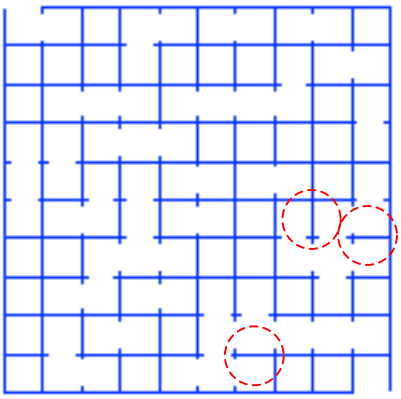
All internal walls are present, and the grid is divided into $10^2 = 100$ **chambers**. With each iteration, the algorithm considers a randomly selected wall that's not been considered before, and removes that wall **if and only if** it separates two chambers. The first randomly selected wall can always be removed, because all locations are initially their own chambers. That first wall might be the one that's now missing:



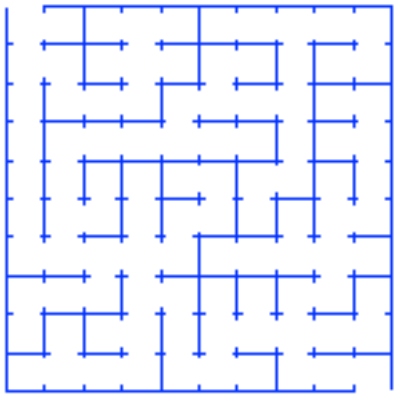
After a several more iterations, you'll see a good number of randomly selected walls have been removed:



The working maze pictured above includes 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, bad maze.



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

Maze Generation Pseudo-code

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

```
choose a dimension
generate all walls, draw them, 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. 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 **`cell`** and **`wall`** types. You should use these definitions. Note that I've already defined **`operator<`** for both **`cell`** and **`wall`**.
- 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`**, so you shouldn't be worried if you don't use one either.
- You shouldn't need to modify anything other that **`maze-generator.cpp`**. The other **`.cpp`** files and all interface files should be fine as. If you do change them, then be sure to submit them and detail how you changed them so we know what to look for.

Part III: Random Sentence Generator [prose by of Julie Zelenski]

Over the past three or four decades, computers have revolutionized student life. In addition to providing entertainment and distraction, computers have also facilitated all sorts of student work. One important area of student labor that has been painfully neglected is the task of filling up space in papers, Ph.D. dissertations, extension requests, etc. with important sounding and somewhat grammatically correct random sequences.

Neglected, that is, until now.

The Random Sentence Generator is a marvelous piece of technology that creates random sentences from a structure known as a **context-free grammar**. A grammar is a construct describing the various combinations of words that can be used to form valid sentences. There are profoundly useful grammars available to generate extension requests, generic Star Trek plots, your average James Bond movie, "Dear John" letters, and more. You can even create your own grammar! Fun for the whole family! Let's show you the value of this practical and wonderful tool:

- Tactic #1: Wear down the TA's patience.

  I need an extension because I had to go to an alligator wrestling meet, and then, just when my mojo was getting back on its feet, I just didn't feel like working, and, well I'm a little embarrassed about this, but I had to practice for the Winter Olympics, and on top of that my roommate ate my disk, and right about then well, it's all a haze, and then my dorm burned down, and just then I had tons of midterms and tons of papers, and right about then I lost a lot of money on the four-square semi-finals, oh, and then I had recurring dreams about my notes, and just then I forgot how to write, and right about then my dog ate my dreams, and just then I had to practice for an intramural monster truck meet, oh, and then the bookstore was out of erasers, and on top of that my roommate ate my sense of purpose, and then get this, the programming language was inadequately abstract.

- Tactic #2: Plead innocence.

I need an extension because I forgot it would require work and then I didn't know I was in this class.

- Tactic #3: Honesty.

  I need an extension because I just didn't feel like working.

What is a grammar?

A grammar is a set of rules for some language, be it English, Java, C++, or something you just invent for fun. ☺ If you continue to study computer science, you will learn much more about languages and grammars in a formal sense. For now, we will introduce to you a particular kind of grammar called a context-free grammar (**CFG**).

Here is an example of a simple CFG for generating poems:

```
<start>
1
The <object> <verb> tonight.

<object>
3
waves
big yellow flowers
slugs

<verb>
3
sigh <adverb>
portend like <object>
die <adverb>

<adverb>
2
warily
grumpily
```

According to this grammar, two syntactically valid poems are **"The big yellow flowers sigh warily tonight."** and **"The slugs portend like waves tonight."** Essentially, the strings in brackets (**<>**) are variables that expand according to the rules in the grammar.

More precisely, each string in brackets is known as a **nonterminal**. A nonterminal is a placeholder that will expand to another sequence of words when generating a poem. In contrast, a **terminal** is a normal word that is not changed to anything else when expanding the grammar. The name terminal is supposed to conjure up the image that it's something of a dead end, and that no further expansion is possible.

A **definition** consists of a nonterminal and a list of possible **productions** (or **expansions**). There will always be at least one and potentially several productions for each nonterminal. A production is just a text string of words, some of which themselves may be non-terminals. A production can be the empty string, which makes it possible for a nonterminal to evaporate into nothingness. An entire definition is summarized within a grammar text file as:

```
<verb>                    ⇐ the first line names the nonterminal and is delimited by < and >
3                         ⇐ the second line is always the number of possible expansions
sigh <adverb>             ⇐ the third line is the first possible expansion
portend like <object>     ⇐       followed by another expansion if there is a second one
die <adverb>              ⇐            followed by another expansion if there is a third one, etc
                          ⇐ for readability, there's a blank line after each definition, including the last one
```

You always begin random sentence generation with the single non-terminal **<start>** as the working string, and iteratively search for the first nonterminal[1] and replace it with any one of its possible expansions (which may and often will include its own nonterminals). Repeat the process over and over until all nonterminals are gone.

```
<start>
The <object> <verb> tonight.                        // expand <start>
The big yellow flowers <verb> tonight.              // expand <object>
The big yellow flowers sigh <adverb> tonight.       // expand <verb>
The big yellow flowers sigh warily tonight.         // expand <adverb>
```

Since we are choosing productions at random, a second generation would almost certainly produce a different sentence.

Your program should repeatedly prompt the user for a grammar file (understood to be in the **grammars** subdirectory), read in the grammar, and generate three random sentences. Only when the user hits return without actually typing in anything should you end the program.

You may assume all grammar files are well formed, and you needn't worry about word wrap as you print über-long sentences. Using the sample application as a guide, you are to make all design and implementation decisions.

If you have the interest and energy, invent a new grammar or two. If you think others would like them, send them to **jerry@cs.stanford.edu** or **truman@cs.stanford.edu** and we'll post them to the course web site.

[1] This problem could also be solved using recursion. We ask that you suppress your desire you employ recursion and implement it iteratively, though.