Eric Roberts
CS106B

Handout #13
April 10, 2009

Assignment #2—Using ADTs

The idea for Random Writer comes from Joe Zachary at the University of Utah.
Parts of this handout were written by Julie Zelenski and Jerry Cain.

Due: Friday, April 17

You've been introduced to the handy CS106 class library, now it's time to put these objects to work! In your role as a client of these ADTs with the low-level details abstracted away, you can put your energy toward solving more interesting problems. In this assignment, your job is to write two short client programs that use these 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 let you experience more fully the joy of using powerful library classes. Most of the heavy-lifting is handled by objects from the CS106 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 increase your familiarity with using C++ class templates.
4. To give you some practice with classic data structures such as the stack, queue, vector, map, and lexicon.

Problem 1. 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 → core → care → dare → date → data
```

That word ladder, however, is not the shortest possible one. Although the words may be a little less familiar, the following ladder is one step shorter:

```
code → cade → cate → date → data
```

Your job in this problem is to write a program that finds a minimal word ladder between two words. Your code will make use of several of the ADTs from Chapter 4, along with a powerful algorithm called breadth-first search to find the shortest such sequence. Here, for example, is a sample run of the word-ladder program in operation:

```
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**, in which 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 routing packets in the Internet, robot motion planning, determining proximity in social networks, comparing gene mutations, and more.

One approach for finding a shortest path is the classic algorithm known as **breadth-first search**, which is a search process that expands outward from the starting position, considering first all possible solutions that are one step away from the start, then all possible solutions that are two steps away, and so on, until an actual solution is found. Because you check all the paths of length 1 before you check any of length 2, the first successful path you encounter must be as short as any other.

For word ladders, the breadth-first strategy starts by examining those ladders that are one step away from the original word, which means that only one letter has been changed. If any of these single-step changes reach the destination word, you're done. If not, you can then move on to check all ladders that are two steps away from the original, which means that two letters have been changed. In computer science, each step in a such a process is often called a **hop**.

The breadth-first algorithm is typically implemented by using a queue 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. Because queues guarantee first-in/first-out processing, these partial word ladders will be dequeued in order of increasing length.

To get the process started, you simply add a ladder consisting of only the start word to the queue. From then on, the algorithm operates by dequeueing the ladder from the front of the queue and determining whether it ends at the goal. If it does, you have a complete ladder, which must be minimal. If not, you take that partial ladder and extend it to reach words that are one additional hop away, and enqueue those extended ladders, where they will be examined later. If you exhaust the queue of possibilities without having found a completed ladder, you can conclude that no ladder exists.

It is possible to make the algorithm considerably more concrete by implementing it in **pseudocode**, which is simply a combination of actual code and English:

```
Create an empty queue.
Add the start word to the end of the queue.
while (the queue is not empty) {
    Dequeue the first ladder from the queue.
    if (the final word in this ladder is the destination word) {
        Return this ladder as the solution.
    }
    for (each word in the lexicon of English words that differs by one letter) {
        if (that word has not already been used in a ladder) {
            Create a copy of the current ladder.
            Add the new word to the end of the copy.
            Add the new ladder to the end of the queue.
        }
    }
}
Report that no word ladder exists.
```

As is generally the case with pseudocode, several of the operations that are expressed in English need to be fleshed out a bit. For example, the loop that reads

```
for (each word in the lexicon of English words that differs by one letter)
```

is a *conceptual* description of the code that belongs there. It is, in fact, unlikely that this idea will correspond to a single `for` loop in the final version of the code. The basic idea, however, should still make sense. What you need to do is iterate over all the words that differ from the current word by one letter. One strategy for doing so is to use two nested loops; one that goes through each character position in the word and one that loops through the letters of the alphabet, replacing the character in that index position with each of the 26 letters in turn. Each time you generate a word using this process, you need to look it up in the lexicon of English words to make sure that it is actually a legal word.

Another issue that is a bit subtle is the restriction that you not reuse words that have been included in a previous ladder. One advantage of making this check is that doing so reduces the need to explore redundant paths. For example, suppose that you have previously added the partial ladder

```
cat → cot → cog
```

to the queue and that you are now processing the ladder

```
cat → cot → con
```

One of the words that is one hop away from `con`, of course, is `cog`, so you might be tempted to enqueue the ladder

```
cat → cot → con → cog
```

Doing so, however, is unnecessary. If there is a word ladder that begins with these four words, then there must be a shorter one that, in effect, cuts out the middleman by eliminating the unnecessary word `con`. In fact, as soon as you've enqueued a ladder ending with a specific word, you never have to enqueue that word again.

The simplest way to implement this strategy is to keep track of the words that have been used in any ladder (which you can easily do using another lexicon) and ignore those words when they come up again. Keeping track of what words you've used also eliminates the possibility of getting trapped in an infinite loop by building a circular ladder, such as

```
cat → cot → cog → bog → bag → bat → cat
```

One of the other questions you will need to resolve is what data structure you should use to represent word ladders. Conceptually, each ladder is just an ordered list of words, which should make your mind scream out "Vector!" (Given that all the growth is at one end, stacks are also a possibility, but vectors will be more convenient when you are trying to print out the results. The individual components of the `Vector` are of type `string`.

Implementing the application

At this point, you have everything you need to start writing the actual C++ code to get this project done. It's all about leveraging the class library—you'll find your job is just to coordinate the activities of various different queues, vectors, and lexicons necessary to get the job done. The finished assignment requires less than a page of code, so it's not a question of typing in statements until your fingers get tired. It will, however, certainly help to think carefully about the problem before you actually begin that typing.