Stanford University, CS 106B
Homework Assignment 2: Word Ladder and N-Grams

Thanks to Julie Zelenski and Jerry Cain for the assignment idea and much of the text; Random Writer comes from Joe Zachary.

The purpose of this assignment is to practice using collections. You will use vector, stack, queue, set, and map. Since it is hard to craft a single problem that exercises all of these collections, this is a **two-part** assignment.

Files:

We will provide you with two separate **ZIP archives**, each of which contains a starter version of that part of your project. Download these archives from the class web site and finish the code. Turn in only the following file:

- **wordladder.cpp**, the C++ code for the Part A Word Ladder program (including a `main` function)
- **ngrams.cpp**, the C++ code for the Part B N-grams program (including a `main` function)
- **myinput.txt**, your own unique Part B input file representing text to read in as your program's input

The ZIP archives contain other files and libraries; you should not modify them. When grading your code, we will run your file with our own original versions of the support files, so your code must work with them.

Part A: 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 the word `"code"` to the word `"data"`. Each changed letter is underlined as an illustration:

code → c<u>a</u>de → ca<u>t</u>e → <u>d</u>ate → dat<u>a</u>

There are many other word ladders that connect these two words, but this one is the shortest. That is, there might be others of the same length, but none with fewer steps than this one.

In the first part of this assignment, you will write a program that repeatedly prompts the user for two words and then finds a minimum-length word ladder between the words. You must use the **`Stack` and `Queue` collections** from Chapter 5, along with following a particular provided algorithm to find the shortest word ladder sequence. This part of the assignment is intended to be shorter and simpler than Part B.

Here is an example **log of interaction** between your program and the user (with console input underlined):

```
Welcome to CS 106B Word Ladder.
Please give me two English words, and I will change the
first into the second by changing one letter at a time.

Word #1 (or Enter to quit): code
Word #2 (or Enter to quit): data
A ladder from data back to code:
data date cate cade code

Word #1 (or Enter to quit):
Have a nice day.
```

Part A Word Ladder sample log of interaction (see course web site for more logs)

Notice that the word ladder prints out in reverse order, from the second word back to the first. If there are multiple valid word ladders of the same length between a given starting and ending word, your program would not need to generate exactly the ladder shown in this log, but you must generate one of minimum length.

Your code should ignore case; in other words, the user should be able to type uppercase, lowercase, mixed case, etc. words and the ladders should still be found and displayed in lowercase. You should also check for several kinds of **user input errors**, and not assume that the user will type valid input. Specifically, you should check that both words typed by the user are valid words found in the dictionary, that they are the same length, and that they are not the same word. If invalid input occurs, your program should print an error message and re-prompt the user. See the logs of execution on the course web site for examples of proper program output for such cases.

You will need to keep a **dictionary** of all English words. We provide a file `dictionary.txt` that contains these words, one per line. Read this file a single time in your program, and choose an efficient collection to store and look up words. Note that you should not ever need to loop over the dictionary as part of solving this problem. You may assume that the English dictionary file exists and is readable by your program.

Finding a word ladder is a specific instance of a **shortest-path problem** of finding a path from a start position to a goal. Shortest-path problems come up in routing Internet packets, comparing gene mutations, and so on. The strategy we will use for finding a shortest path is called **breadth-first search ("BFS")**, a search process that expands out from a start position, considering all possibilities that are one step away, then two steps away, and so on, until a solution is found. BFS guarantees that the first solution you find will be as short as any other.

For word ladders, start by examining ladders that are one step away from the original word, where only one letter is changed. Then check all ladders that are two steps away, where two letters have been changed. Then three, four, etc. We implement the breadth-first algorithm using a **queue** to store partial ladders that represent possibilities to explore. Each partial ladder is a **stack**, which means that your overall collection is a **queue of stacks**.

Here is a partial **pseudocode** description of the algorithm to solve the word-ladder problem:

```
finding a word ladder between words w1 and w2:
    create an empty queue of stacks.
    create/add a stack containing {w1} to the queue.
    while the queue is not empty:
        dequeue the partial-ladder stack from the front of the queue.
        for each valid English word that is a "neighbor" (differs by 1 letter)
        of the word on top of the stack:
            if that neighbor word has not already been used in a ladder before:
                if the neighbor word is w2:
                    hooray!  we have found a solution.
                otherwise:
                    create a copy of the current partial-ladder stack.
                    put the neighbor word on top of the copy stack.
                    add the copy stack to the end of the queue.
```

Some of the pseudocode corresponds almost one-to-one with actual C++ code. One part that is more abstract is the part that instructs you to examine each "**neighbor**" of a given word. A neighbor of a given word *w* is a word of the same length as *w* that differs by exactly 1 letter from *w*. For example, "`date`" and "`data`" are neighbors.

It is *not* appropriate to look for neighbors by looping over the entire dictionary every time; this is way too slow. To find all neighbors of a given word, use two nested loops: one that goes through each character index in the word, and one that loops through the letters of the alphabet from `a`-`z`, replacing the character in that index position with each of the 26 letters in turn. For example, when examining neighbors of "`date`", you'd try:

- `aate`, `bate`, `cate`, ..., `zate` ← *all possible neighbors where only the 1st letter is changed*
- `date`, `dbte`, `dcte`, ..., `dzte` ← *all possible neighbors where only the 2nd letter is changed*
- ...
- `data`, `datb`, `datc`, ..., `datz` ← *all possible neighbors where only the 4th letter is changed*

Note that many of the possible words along the way (`aate`, `dbte`, `datz`, etc.) are not valid English words. Your algorithm has access to an **English dictionary**, and each time you generate a word using this looping process, you should look it up in the dictionary to make sure that it is actually a legal English word.

Another subtle issue is that you **do not reuse words** that have been included in a previous ladder. For example, suppose that you have add the partial ladder `cat` → `cot` → **`cog`** to the queue. Later on, if your code is processing ladder `cat` → `cot` → `con`, one neighbor of `con` is `cog`, so you might want to examine `cat` → `cot` → `con` → **`cog`**.

But doing so 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`. As soon as you've enqueued a ladder ending with a specific word, you've found a minimum-length path from the starting word to the end word in the ladder, so you *never* have to enqueue that end word again.

To implement this strategy, keep track of the set of words that have already been used in *any* ladder. Ignore those words if 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`**.

Part B: N-Grams

In the second part of this assignment, you will write a program that reads an input file and uses it to build a large data structure of word groups called "*N*-grams" as a basis for randomly generating new text that sounds like it came from the same author as that file. You will use the `Map` and `Vector` collections from Chapter 5.

At right is an example **log of interaction** between your program and the user (console input underlined).

But what, you may ask, is an *N*-gram?

```
Welcome to CS 106B Random Writer ('N-Grams').
This program makes random text based on a document.
Give me an input file and an 'N' value for groups
of words, and I'll create random text for you.

Input file name? hamlet.txt
Value of N? 3

# of random words to generate (0 to quit)? 40
... chapel. Ham. Do not believe his tenders, as you
go to this fellow. Whose grave's this, sirrah?
Clown. Mine, sir. [Sings] O, a pit of clay for to
the King that's dead. Mar. Thou art a scholar; speak
to it. ...

# of random words to generate (0 to quit)? 20
... a foul disease, To keep itself from noyance; but
much more handsome than fine. One speech in't I
chiefly lov'd. ...

# of random words to generate (0 to quit)? 0
Exiting.
```

The "Infinite Monkey Theorem" states that an infinite number of monkeys typing random keys forever would eventually produce the works of William Shakespeare. That's silly, but could a monkey randomly produce a *new work* that "sounded like" Shakespeare's works, with similar vocabulary, wording, punctuation, etc.? What if we chose *words* at random, instead of individual letters? Suppose that rather than each word having an equal probability of being chosen, we weighted the probability based on how often that word appeared in Shakespeare's works?

Picking random words would likely produce gibberish, but let's look at *chains of two words* in a row. For example, perhaps Shakespeare uses the word "to" occurs 10 times total, and 7 of those occurrences are followed by "be", 1 time by "go", and 2 times by "eat". We can use those ratios when choosing the next word. If the last word we chose is "to", randomly choose "be" next 7/10 of the time, "go" 1/10 of the time, and "eat" 2/10. We never choose any other word to follow "to". We call a chain of two words like this, such as "to be", a *2-gram*.

```
+--------------------+
| Chose "to".        |----> choose "be"    (7/10 chance)
| Next random word?  |----> choose "go"    (1/10 chance)
+--------------------+----> choose "eat"   (2/10 chance)
```

Go, get you have seen, and now he makes as itself? *(2-gram)*

A sentence of 2-grams isn't great, but look at chains of 3 words (*3-grams*). If we chose the words "to be", what word should follow? If we had a collection of all sequences of 3 words-in-a-row with probabilities, we could make a weighted random choice. If Shakespeare uses "to be" 22 times and follows them with "or" 5 times, "in" 3 times, "with" 10 times, and "alone" 4 times, we could use these weights to randomly choose the next word. So now the algorithm would pick the third word based on the first two, and the fourth based on the (second+third), and so on.

```
+--------------------+----> choose "or"    ( 5/22 chance)
| Chose {"to", "be"}. |----> choose "in"    ( 3/22 chance)
| Next random word?  |----> choose "with"  (10/22 chance)
+--------------------+----> choose "alone" ( 4/22 chance)
```

One woe doth tread upon another's heel, so fast they follow. *(3-gram)*

You can generalize the idea from 2-grams to *N*-grams for any integer *N*. If you make a collection of all groups of *N* words along with their possible following words, you can use this to select an *N*+1'th word given the preceding *N* words. The higher *N* level you use, the more similar the new random text will be to the original data source. Here is a random sentence generated from 5-grams of *Hamlet*, which is starting to sound a lot like the original:

I cannot live to hear the news from England, But I do prophesy th' election lights on Fortinbras. *(5-gram)*

Each particular piece of text randomly generated in this way is also called a *Markov chain*. Markov chains are very useful in computer science and elsewhere, such as artificial intelligence, machine learning, economics, and statistics.

Part B, Algorithm Step 1: Building Map of N-Grams

In this program, you will read the input file one word at a time and build a particular compound collection, a **map** from prefixes to suffixes. If you are building 3-grams, that is, *N*-grams for *N*=3, then your code should examine sequences of 2 words and look at what third word follows those two. For later lookup, your map should be built so that it connects a collection of *N*-1 words with another collection of all possible suffixes; that is, all possible *N*'th words that follow the previous *N*-1 words in the original text. For example if you are computing *N*-grams for *N*=3 and the pair of words `"to be"` is followed by `"or"` twice and `"just"` once, your collection should map the key `{to, be}` to the value `{or, just, or}`. The following figure illustrates the map you should build from the file:

When reading the input file, the idea is to keep a window of *N*-1 words at all times, and as you read each word from the file, discard the first word from your window and append the new word. The following figure shows the file being read and the map being built over time as each of the first few words is read to make 3-grams:

to be or not to be just ... ^	map = {} window = {to, be}
to be **or** not to be just ... ^	map = {**{to, be} : {or}**} window = {be, **or**}
to be or **not** to be just ... ^	map = {{to, be} : {or}, **{be, or} : {not}**} window = {or, **not**}
to be or not **to** be just ... ^	map = {{to, be} : {or}, {be, or} : {not}, **{or, not} : {to}**} window = {not, **to**}
to be or not to **be** just ... ^	map = {{to, be} : {or}, {be, or} : {not}, {or, not} : {to}, **{not, to} : {be}**} window = {to, **be**}
to be or not to be **just** ... ^	map = {{to, be} : {or, **just**}, {be, or} : {not}, {or, not} : {to}, {not, to} : {be}} window = {be, **just**}
...	...
to be or not to be just be who you want to be or not okay you want okay	map = {{to, be} : {or, just, or}, {be, or} : {not, not}, {or, not} : {to, okay}, {not, to} : {be}, {be, just} : {be}, {just, be} : {who}, {be, who} : {you}, {who, you} : {want}, {you, want} : {to, okay}, {want, to} : {be}, {not, okay} : {you}, {okay, you} : {want}, **{want, okay} : {to},** **{okay, to} : {be}**}
input file, tiny.txt	*resulting map of 3-gram suffixes*

Note that the **order matters**: For example, the prefix `{you, are}` is different from the prefix `{are, you}`. Note that the same word can occur multiple times as a suffix, such as `"or"` occurring twice after the prefix `{to, be}`.

Also notice that the map **wraps around**. For example, if you are computing 3-grams like the above example, perform 2 more iterations to connect the last 2 prefixes in the end of the file to the first 2 words at the start of the file. In our example above, this leads to `{want, okay}` connecting to `"to"` and `{okay, to}` connecting to `"be"`. If we were doing 5-grams, we would perform 4 more iterations and connect the last 4 prefixes to the first 4 words in the file, and so on. This turns out to be very useful to help your algorithm later on in the program.

Your **should not change case or strip punctuation** of words as you read them. The casing and punctuation turns out to help the sentences start and end in a more authentic way. Just store the words in the map as you read them.

Part B, Algorithm Step 2: Generating Random Text

To generate random text from your map of *N*-grams, first choose a random starting point for the document. To do this, pick a randomly chosen key from your map. Each key is a collection of *N*-1 words. Those *N*-1 words will form the start of your random text. This collection of *N*-1 words will be your **sliding "window"** as you create your text.

For all subsequent words, use your map to look up all possible next words that can follow the current *N*-1 words, and randomly choose one with appropriate weighted probability. If you have built your map the way we described, as a map from {prefix} → {suffixes}, this simply amounts to choosing one of the possible suffix words at random. Once you have chose your random suffix word, slide your current "window" of *N*-1 words by discarding the first word in the window and appending the new suffix. The following diagram illustrates the text generation algorithm.

Action(s)	Current (*N*-1) "window"	Output so far
choose a random start	`{"who", "you"}`	`who you`
choose new word; shift	`{"you", "want"}`	`who you want`
choose new word; shift	`{"want", "okay"}`	`who you want okay`
choose new word; shift	`{"okay", "to"}`	`who you want okay to`
...	`...`	`...`

Note that in our random example, at one point our window was `{want, okay}`. This was the end of the original input file. Nothing actually follows that prefix, which is why it was important that we made our map **wrap around** from the end of the file to the start, so that if our window ever ends up at the last *N*-1 words from the document, we won't get stuck unable to generate further random text.

Since your random text likely won't happen to start and end at the beginning/end of a sentence, just prefix and suffix your random text with "..." to indicate this. Here is another partial log of execution:

```
Input file? tiny.txt
Value of N? 3
# of random words to generate (0 to quit)? 16
... who you want okay to be who you want to be or not to be or ...
```

Your code should check for several kinds of **user input errors**, and not assume that the user will type valid input. Specifically, re-prompt the user if they type the name of a file that does not exist. Also re-prompt the user if they type a value for *N* that not an integer, or is an integer less than 2 (a 2-gram has prefixes of length 1; but a 1-gram is essentially just choosing random words from the file and is uninteresting for our purposes). When prompting the user for the number of words to randomly generate, re-prompt them if the number of random words to generate is not at least *N*. You may assume that the value the user types for *N* is not greater than the number of words found in the file. See the logs of execution on the course web site for examples of proper program output for various cases.

Development Strategy and Hints:

This program can be tricky if you don't develop and debug it step-by-step. Don't try to write everything all at once. Make sure to **test** each part of the algorithm before you move on. See the **Homework FAQ** for more tips.

- Think about exactly what **types of collections** to use for each part. Are duplicates allowed? Does order matter? Do you need random access? Where will you add/remove elements? Etc. Note that some parts of each program require you to make compound collections, that is, a collection of collections.
- Test each function with a very **small input** first. For example, use input file `tiny.txt` with a small number of words because you can **print your entire map** and examine its contents.
- Recall that you can **print** the contents of any collection to `cout` and examine its contents for debugging.
- Remember that when you assign one collection to another using the `=` operator, it makes a full copy of the entire contents of the collection. This could be useful if you want to copy a collection.
- To choose a random prefix from a map, consider using the map's `keys` member function, which returns a `Vector` containing all of the keys in the map. For **randomness** in general, include `"random.h"` and call the global function `randomInteger(`***`min, max`***`)`.
- You can loop over the elements of a vector or set using a for-each loop. A for-each also works on a map, iterating over the keys in the map. You can look up each associated value based on the key in the loop.
- Don't forget to test your input on **unusual inputs**, like large and small values of *N*, large/small # of words to generate, large and small input files, and so on. It's hard to verify random input, but you can look in smallish input files to verify that a given word really does follow a given prefix from your map.

Implementation and Grading:

All items mentioned in the "Implementation and Grading" from the previous assignment(s) specs also apply here. Please refer to those documents as needed. Note the instructions in the previous assignment about procedural decomposition, variables, types, parameters, value vs. reference, and commenting. As in the previous assignment, you should break down each problem into several coherent functions. Don't forget to **cite any sources** you used in your comments. Refer to the course **Style Guide** for a more thorough discussion of good coding style.

Algorithms: You should follow the general algorithms as described in this document and should not substitute a very different algorithm. In particular, you should *not* write a recursive algorithm for finding word ladders or N-grams.

Collections: Additionally, on this assignment part of your Style grade comes from making intelligent decisions about what kind of **collections** from the Stanford C++ library to use at each step of your algorithm, as well as using those collections elegantly. As much as possible, **pass collections by reference**, because passing them by value makes an expensive copy of the collection. Do not use pointers, arrays, or STL containers on this program.

You should also avoid expensive operations that would cause you to reconstruct bulky collections multiple times unnecessarily. For example, in Part B, generate the map of prefixes exactly once; do not regenerate it each time the user asks to generate random text.

Don't forget to use the course web site's **Output Comparison Tool** to help check your output for various test cases.

Honor Code: Please remember to follow the **Honor Code** when working on this assignment. Submit your own work and do not look at others' solutions. Also please do not give out your solution and do not place a solution to this assignment on a public web site or forum. If you need help, please seek out our available resources to help you.

For reference, our solution to Part A (the complete file) is around 115 lines long including spacing and comments. Our solution to Part B (the complete file) is around 130 lines long, and it has 4 functions besides `main`, though you do <u>not</u> need to match or come close to these numbers to get full credit; they are just here as a ballpark sanity check.

Possible Extra Features:

Though your solution to this assignment must match all of the specifications mentioned previously, it is allowed and encouraged for you to add extra features to your program if you'd like to go beyond the basic assignment. Here are some example ideas for extra features that you could add to your program.

- **Allow word ladders between words of different length:** The typical solution forbids word ladders between words that are not the same length. But as an extra feature, you could make it so that it is considered legal to add or remove a single letter from your string at each hop along the way. This would make it possible to, for example, generate a word ladder from "car" to "cheat": car, cat, chat, cheat.

- **Allow word ladder end-points to be outside the dictionary:** Generally we want our word ladders to consist of words that are valid English words found in the dictionary. But it can be fun to allow only the start and end words to be non-dictionary words. For example, "Marty" is not an English word, but if you did this extra feature, you could produce a word ladder from "Marty" to "curls" as: marty, party, parts, carts, cards, curds, curls.

- **Make the *N*-grams be complete sentences:** The typical version of the assignment indicates that you should start and end your input with "..." since it will likely not begin with the start of a sentence nor end with the end of a sentence from the original input. As an extra feature, make it so that when you are creating your map of *N*-1 word prefixes, you also keep track of which prefixes are the start of a sentence (prefixes whose first word begins with an uppercase letter) and which words are the end of a sentence (words that end with a period, question mark, or exclamation mark). Use this extra data to begin your randomly generated text with a random sentence starter, rather than any random prefix. And instead of generating exactly the number of words requested by the user, keep going until you reach the end of a sentence. That is, if the user requests 100 words, after generating those 100 words, if you aren't at the end of a sentence, keep going until you end it.

- **Other:** If you have your own creative idea for an extra feature, ask your SL and/or the instructor about it.

Indicating that you have done extra features: If you complete any extra features, then in the comment heading on the top of your program, please list all extra features that you worked on and where in the code they can be found (what functions, lines, etc. so that the grader can look at their code easily).

Submitting a program with extra features: Since we use automated testing for part of our grading process, it is important that you submit a program that conforms to the preceding spec, even if you want to do extra features. If your extra feature(s) cause your program to change the output that it produces in such a way that it no longer matches the expected sample output test cases provided, you should **submit two versions of your program files**: a first one where the files are named **wordladder.cpp** and **ngrams.cpp** without any extra features added (or with all necessary features disabled or commented out), and a second one named **wordladder-extra.cpp** (and/or **ngrams-extra.cpp**) with the extra features enabled. Please distinguish them in by explaining which is which in the comment header. Please distinguish them in by explaining which is which in the comment header. Our turnin system saves every submission you make, so if you make multiple submissions we will be able to view all of them; your previously files will not be lost or overwritten.