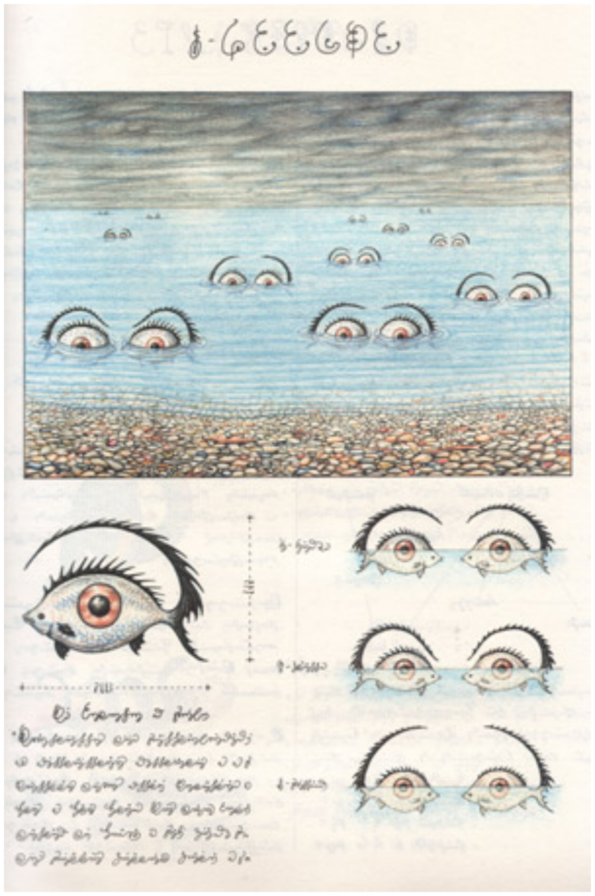
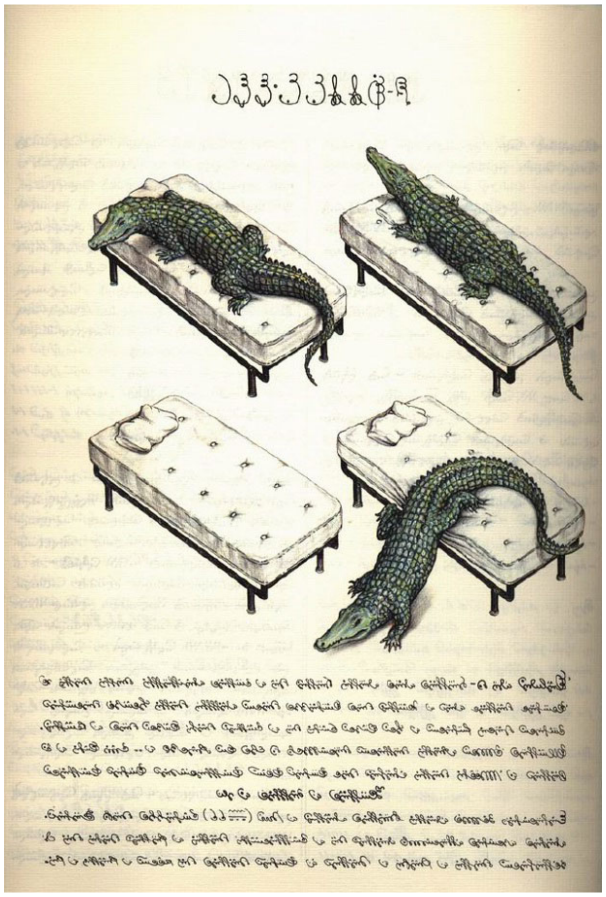
Assignment 2: Serafini

Thanks to Julie Zelenski, Jerry Cain, Marty Stepp, and Chris Piech; Random Writer comes from Joe Zachary.

JULY 5, 2017

Images from Codex Seraphinianus by Serafini

The title of this assignment pays homage to Luigi Serafini, author of a book called the *Codex Seraphinianus*. The codex is filled with pseudo-scientific writing and figures like the above images, all of which are believed to be purely the product of Serafini's imagination. Two hallmarks of Serafini's work are morphings—many objects in the codex change from one to another—and random writing.

As such, this assignment consists of two parts: (A) Word Ladders, where you find ways to morph one word into another and (B) Random Writer, where your program stochastically (i.e., randomly) generates text. Each part can be programmed separately, but they should be submitted together. The starter code for this project is available as a ZIP archive:

Starter Code: web.stanford.edu/class/cs106b//assnFiles/serafini/serafini_starter.zip

Due Date: Serafini is due Wednesday, July 12th at 12:00pm (noon).

Y.E.A.H hours (will be recorded):
Wednesday, July 5th
7-8pm
Building 380-380C

Turn in only the following files:

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

This is *not* a pair assignment (soon!). You must work on your code alone, though as always you are allowed to talk about the problems with other students, on Piazza, in LaIR, office hours, etc.

Part A: Word Ladders

A **word ladder** is a bridge between one word and 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` → **`cade`** → **`cate`** → **`date`** → **`data`**

There are many other word ladders that connect these two words, but this one uses the shortest path between them (which means that there might be other ladders of the same length, but none with fewer steps than this one).

In the first part of this assignment (which is simpler than part B), write a program that repeatedly prompts the user for two words and finds a minimum-length ladder linking the words. You must use the Stack and Queue collections from Chapter 5, and you must follow the algorithm provided below to find the shortest word ladder.

Example Run:

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

```
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.

Dictionary file name? dictionary.txt

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.
```

Notice that your word ladder should print out in reverse order, from the second word back to the first. If there are multiple valid word ladders of the minimum length between a given starting and ending word, your program can return any of them.

Your code should ignore case; in other words, the user should be able to type uppercase, lowercase, or mixed-case 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 reprompt the user. See the logs of execution in the following sample runs for examples of proper program output for such cases:

Ladder Run #1 Ladder Run #2 Ladder Run #3 Ladder Run #4

To implement Word Ladders, you will need to keep a dictionary of all English words. Your program should prompt the user to enter a dictionary file name and use that file as the source of English words. (If the user types a filename that does not exist, reprompt them; see the second execution log on the next page for an example.) We provide a file dictionary.txt containing these English words, one per line. Read the dictionary file just once during your program, and choose an efficient collection to store and look up words. Note that you should not ever need to loop over the entire dictionary as part of solving this problem. We suggest using the Stanford library's **`Lexicon`** class to store the dictionary, since using the **`Lexicon`** class is quite straightforward— please see the Lexicon class reference for details. After reading in the dictionary of words, you should use the **`contains()`** method as needed to determine if a word is in the dictionary.

Algorithm:

Finding a word ladder is a specific instance of what is known as a shortest-path problem: a problem in which we try to find the shortest possible route from a given start to a given end point. Shortest-path problems come up in routing Internet packets, comparing gene mutations, Google Maps, and many other domains. The strategy we will use for finding the shortest path between our start and end words is called breadth-first search ("BFS"). This is a search process that gradually expands the set of paths among which we search "outwards:" BFS first considers all possible paths that are one step away from the starting point, then all possible paths two steps away, and so on, until a path is found connecting the start and end point. A step can be understood as one unit of measurement—depending on the problem, this could be a millisecond, a minute, a mile, a subway stop, and so on. By exploring *all* possible paths of a given length before incrementing to the next length, BFS guarantees that the first solution you find will be as short as possible.

For word ladders, start by examining ladders that contain only words that are one "step" away from the original word—i.e., words in which only one letter has been changed. If you find your target word among these one-step-away ladders, congratulations—you're done! If not, look for your target word in all ladders containing words that are two steps away, i.e., ladders in which two letters have been changed. Then check three letters, four, etc., until your target word is located. We implement the breadth-first algorithm using a queue that stores partial ladders, each of which represents a possibility to explore (i.e., each item in the queue is a partial ladder that we will examine in turn, checking to see if it contains a path to our target word). Each partial ladder is represented as a stack, which means that your overall collection will be a queue of stacks.

Here is a partial pseudo-code description of the algorithm to solve the word-ladder problem (***Note: some students complain at this point that we have given you too much information and that they want to figure the problem out on their own — great! Don't look at the pseudocode below if you want to try it on your own!***)

```
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 (and it is the stack
                                     you are working on in the queue).
                                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 pseudo-code corresponds almost one-to-one with actual C++ code. Other parts are more abstract, such as the instruction 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; **"dog"** and **"bog"** are neighbors.

Your solution is not allowed to look for neighbors by looping over the dictionary every time; this is much too inefficient. 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 letter combinations 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. Only valid English words should be included in your group of neighbors.

Another way of visualizing the search for neighboring words is to think of each letter index in the word as being a "spinner" that you can spin up and down to try all values A-Z for that letter. The diagram below tries to depict this:

```
index      0   1   2   3
          ... ... ... ...
           a   m   b   c
           b   n   c   d
         +---+---+---+---+
         | c | o | d | e |
         +---+---+---+---+
           d   p   e   f
           e   q   f   g
          ... ... ... ...
```

Another subtle issue is that you should not reuse words that have been included in a previous, shorter ladder. For example, suppose that you have the partial ladder cat → cot → cog in your queue. Later on, if your code is processing the 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 that 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 which 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.

A final tip: it may be helpful to test your program on smaller dictionary files first to find bugs or issues related to your dictionary or to word searching.

Part B: Random Writer

In the second part of this assignment, you will write a program that takes in a text file and then (randomly) generates new text in the same style. In effect, your program will automate an exercise that writers have used for centuries to improve their style: imitating another writer. Benjamin Franklin, for example, wrote in his autobiography that he taught himself to write elegantly by trying to re-write passages from *The Spectator* (a popular 18th-century magazine) in the same style. In the 20th century, the writer Raymond Queneau featured the practice of trying to imitate the voice of other writers prominently among his famous "Exercises in Style." A more recent computational style imitator is the postmodern essay generator, created in 1996; hit refresh a few times to see what it can do.

Your program will accomplish the task of imitating a text's style by building and relying on a large data structure of word groups called "N-grams." A collection of N-grams will serve as the basis for your program to randomly generate new text that sounds like it came from the same author as your input file; your program will also use the **`Map`** and **`Vector`** collections from Chapter 5.

Below is an example log of interaction between your program and the user (console input in red).

```
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.
```

But what, you may ask, is an N-gram? The "Infinite Monkey Theorem" states that an infinite number of monkeys typing random keys forever would eventually produce the works of William Shakespeare. (See here.) That's silly, but could a monkey randomly produce a new work that sounded like Shakespeare's works, with similar vocabulary, sentence structure, 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" 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. Here's an example sentence produced by linking together a sequence of 2-grams:

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

A sentence of 2-grams tends not make any sense (or even be grammatically correct), so let's consider 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, together with their 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. Here's a sample sentence produced by linking 3-grams together:

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

We can generalize this idea from 2- or 3-grams to N-grams for any integer N. If you make a collection of all groups of N-1 words along with their possible following words, you can use this to select an Nth word given the preceding N-1 words. The higher your choice of N, 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:

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

As you can see, the text is now starting to sound a lot like the original. 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.

Here are a few more example runs:

Writer Run #1 Writer Run #2 Writer Run #3 Writer Run #4 Writer Run #5

Step 1: Building Map of N-Grams

Your program will read the input file one word at a time and build a specific kind of compound collection, namely, a map from each (N-1)-gram in your input text to all of the words that occur directly after it in the input. For example, if you are building sentences out of 3-grams, that is, N-grams for N=3, then your code should examine each sequence of 2 words and keep track of which third word directly follows it each time it occurs. So if you are using 3-grams 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}. We might think of these (N-1)-grams as “prefixes” and the words that follow them as various possible “suffixes.” Using this terminology, your map should be built so that it connects a collection of each prefix sequence (of N-1 words) to another collection of all possible suffixes (all Nth words that follow the prefix in the original text).

The following figure illustrates the map you should build from the input file. As this figure shows, when reading the input file, you should keep track of a window of N-1 words at all times. As you read each word from the file, you should insert the previous window as a key in your collection of prefixes, with the newly-read word, the current suffix, as its corresponding value. (Remember that your collection of suffixes **must** be able to contain multiple occurrences of the same word—such as "or" occurring twice after the prefix {to, be}—otherwise, your collection will not have the word-frequency information needed to weight your probabilities.) Then, discard the first word from the window, append the new word to the window, and repeat. The map is thus built over time as you read each new word:

File Location	**Data**
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, 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} }
```

Note that the order of sequences matters: for example, the prefix {you, are} is different from the prefix {are, you}. Also notice that the map wraps around. For example, if you are computing 3-grams like in the above example, perform 2 more iterations to connect the last 2 prefixes at 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 will be very useful for your algorithm later on in the program.

You **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.

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, randomly chose a key from your map. Remember that each key is prefix, a collection of N-1 words. Those N-1 words will form the start of your random text, and will be the first sequence in your sliding "window" as you create your text.

For all subsequent words, use your map to look up all possible prefixes that can follow the current N-1 words, and randomly choose one according to their appropriately-weighted probabilities. If you have built your map the way we described above, 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, and repeat the process

Since your random text is unlikely to start and end at the proper beginning or end of a sentence, just preface and conclude 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 is 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.

Creative Aspect myinput.txt

Along with your program, submit a file myinput.txt that contains a text file that can be used as input for Part B. This can be anything you want, as long as it is non-empty and is something you gathered yourself (not just a copy of an existing input file). For example, if you like a particular band, you could paste several of their songs into a text file, which leads to funny new songs when you run your N-grams program on this data. Or gather the text of a book you really like, or poems, or anything you want. This is meant to give you a chance to play around with the creative power of your program, and is worth a small part of your grade on the assignment.

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.

1. Unlike in the previous assignment, here we are interested in each word. If you want to read a file one word at a time, an effective way to do so is using the input >> variable; syntax rather than getLine(), etc.
2. 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.
3. Test each function with a very small input first. For example, use input file tiny.txt with a small number of words; this will let you print your entire map and examine its contents for debugging purposes.
4. Similarly, recall that you can print the contents of any collection to cout and examine its contents for debugging.
5. 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.
6. 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).

7. You know that you can loop over the elements of a vector or set using a for-each loop. A for-each loop also works on a map, iterating over its keys. You can look up each associated value based on the key in the loop.
8. 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.

Possible Extra Features:

Congratulations! You’re done. If you have the time, consider adding extra features.

Though your solution to this assignment must match all of the specifications mentioned previously, you are allowed and encouraged 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.

1. **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 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.
2. **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.
3. **Make the N-grams random text form complete sentences:** 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, extend your program so that when you create your map of (N-1)-gram prefixes, you also keep track of which prefixes are at the start of a sentence (i.e., prefixes whose first word begins with an uppercase letter) and which words are at the end of a sentence (i.e., 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 start, 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.
4. **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, 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 easily identify these parts of your code).

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 assignment spec, even if you do extra features. If your 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 file: a first one named life.cpp without any extra features added (or with all necessary features disabled or commented out), and a second one named life-extra.cpp with the extra features enabled. Please distinguish them by explaining which is which in the comment header. Our submission system saves every submission you make, so if you make more than one we will be able to view all of them; your previously submitted files will not be lost or overwritten.