Computing II Lab Assignment
Evil Hangman

It's hard to write computer programs to play games. When we as humans sit down to play a game, we can draw on past experience, adapt to our opponents' strategies, and learn from our mistakes. Computers, on the other hand, blindly follow a preset algorithm that (hopefully) causes it to act somewhat intelligently. Though computers have bested their human masters in some games, most notably checkers and chess, the programs that do so often draw on hundreds of years of human game experience and use extraordinarily complex algorithms and optimizations to out-calculate their opponents.

While there are many viable strategies for building competitive computer game players, there is one approach that has been fairly neglected in modern research – cheating. Why spend all the effort trying to teach a computer the nuances of strategy when you can simply write a program to play dirty and win handily all the time? In this assignment, you will build a mischievous program that bends the rules of Hangman to trounce its human opponent time and time again. In doing so, you'll cement your skills with abstract data types, and will hone your general programming savvy. Plus, you'll end up with a piece of software which will be highly entertaining. At least, from your perspective. ☺

In case you aren't familiar with the game of Hangman, the rules are as follows:

1. One player chooses a secret word, then writes out a number of dashes equal to the word length.

2. The other player begins guessing letters. Whenever she guesses a letter contained in the hidden word, the first player reveals every instance of that letter in the word. Otherwise, the guess is wrong.

3. The game ends either when all the letters in the word have been revealed or when the guesser has run out of guesses.

Fundamental to the game is the fact the first player accurately represents the word she has chosen. That way, when the other players guess letters, she can reveal whether that letter is in the word. But what happens if the player doesn't do this? This gives the player who chooses the hidden word an enormous advantage. For example, suppose that you're the player trying to guess the word, and at some point you end up revealing letters until you arrive at this point with only one guess remaining:

D O – B L E

There are only two words in the English language that match this pattern: “doable” and “double.” If the player who chose the hidden word is playing fairly, then you have a fifty-fifty chance of winning this game on your next guess if you guess 'A' or 'U' as the missing letter. However, if your opponent is cheating and hasn't actually committed to either word, then there is no possible way you can win this game on that first guess. No matter what letter you guess, your opponent can claim that she had picked the other word.

That is, if you guess that the word is “doable,” she can pretend that she committed to “double” the whole time, and vice-versa.

Let's illustrate this technique with an example. Suppose that you are playing Hangman and it's your turn to choose a word, which we'll assume is of length four. Rather than committing to a secret word, you instead compile a list of every four-letter word in the English language. For simplicity, let's assume that English only has a few four-letter words, all of which are reprinted here:

ALLY BETA COOL DEAL ELSE FLEW GOOD HOPE IBEX

Now, suppose that your opponent guesses the letter 'E.' You now need to tell your opponent which letters in the word you've "picked" are E's. Of course, you haven't picked a word, and so you have multiple options about where you reveal the E's. Here's the above word list, with E's highlighted in each word:

ALLY BETA COOL DEAL ELSE FLEW GOOD HOPE IBEX

If you'll notice, every word in your word list falls into one of five "word families:"

- `----`, which contains the words ALLY, COOL, and GOOD.
- `-E--`, containing BETA and DEAL.
- `--E-`, containing FLEW and IBEX.
- `E--E`, containing ELSE.
- `---E`, containing HOPE.

Since the letters you reveal have to correspond to some word in your word list, you can choose to reveal any one of the above five families. There are many ways to pick which family to reveal – perhaps you want to steer your opponent toward a smaller family with more obscure words, or toward a larger family in the hopes of keeping your options open. In this assignment, in the interests of simplicity, we'll adopt the latter approach and always choose the largest of the remaining word families. In this case, it means that you should pick the family `----`. This reduces your word list down to

ALLY COOL GOOD

and since you didn't reveal any letters, you would tell your opponent that his guess was wrong.

Let's see a few more examples of this strategy. Given this three-word word list, if your opponent guesses the letter O, then you would break your word list down into two families:

- `-OO-`, containing COOL and GOOD.
- `----`, containing ALLY.

The first of these families is larger than the second, and so you choose it, revealing two O's in the word and reducing your list down to

COOL GOOD

But what happens if your opponent guesses a letter that doesn't appear anywhere in your word list? For example, what happens if your opponent now guesses 'T'? This isn't a problem. If you try splitting these words apart into word families, you'll find that there's only one family – the family `-OO-` in which T

appears nowhere and which contains both COOL and GOOD. Since there is only one word family here, it's trivially the largest family, and by picking it you'd maintain the word list you already had.

There are two possible outcomes of this game. First, your opponent might be smart enough to pare the word list down to one word and then guess what that word is. In this case, you should congratulate him – that's an impressive feat considering the scheming you were up to! Second, and by far the most common case, your opponent will be completely stumped and will run out of guesses. When this happens, you can pick any word you'd like from your list and say it's the word that you had chosen all along. The beauty of this setup is that your opponent will have no way of knowing that you were dodging guesses the whole time – it looks like you simply picked an unusual word and stuck with it the whole way.

The Assignment

Your semester-long assignment, much of which you have already completed or are in the process of completing, is to write a computer program which plays a game of Hangman using this "Evil Hangman" algorithm.

Prerequisites:

1. A string-handling module. You have already completed your *MyString* module for this purpose.

2. A generic vector module. You have already completed this in class.

3. An array of *Vectors* of *MyStrings* containing the entire dictionary read from the "dictionary.txt" file. The dictionary file contains an unabridged dictionary with over 120,000 words. This should be the primary focus on today's lab.

   The array *of Vectors of MyStrings* should hold 30 *Vectors* (to hold vectors of words of lengths 1 to 29), each of which holds *MyString* objects, which each hold a word.

   TA CHECKPOINT 1 Although the entire assignment is being given today for the rest of the semester, this is as far as I would expect you to get today. Demonstrate to your TA that you can get your array of generic vectors of MyStrings to store all of the words in the dictionary in vectors where all the words in that vector are the same length. Have your program print out the size of each one of these vectors (essentially providing a count of how many words of each length that you have).

4. An *AssociativeArray* module, as described during a later lab, which allows using *MyString* data as the "key" or "index" to locate a particular *Vector of Mystrings*. The *AssociativeArray* must be based on (implemented using) the *AVL Tree* that you have researched and written on your own or in your lab pairing (or the easy way out described in lab). There will be no duplicate keys. When a new *MyString* data item is added to the *AssociativeArray*, a "key" value (also a *MyString* object) will be given. That "key" is located in the AVL tree (or added to the AVL tree), and the data *MyString* is added to the *Vector* associated with that "key". Look-up by "key" yields a *Vector* of all of the *MyStrings* that have been added to the *Vector* associated with that key.

Your Hangman program, then, should do the following:

1. Prompt the user for a word length, re-prompting as necessary until she enters a number such that there's at least one word that's exactly that long. That is, if the user wants to play with words of length -42 or 137, since no English words are that long, you should reprompt her.

2. Prompt the user for a number of guesses, which must be an integer greater than zero. Don't worry about unusually large numbers of guesses – after all, having more than 26 guesses is clearly not going to help your opponent!

3. Prompt the user for whether she wants to have a running total of the number of words remaining in the word list. This completely ruins the illusion of a fair game that you'll be cultivating, but it's quite useful for testing (and grading!)

4. Play a game of Hangman using the Evil Hangman algorithm, as described below:

    a. Select the *Vector* of all words in the English language whose length matches the input length.

    b. Print out how many guesses the user has remaining, along with any letters the player has guessed and the current blanked-out version of the word. If the user chose earlier to see the number of words remaining, print that out too.

    c. Prompt the user for a single letter guess, re-prompting until the user enters a letter that she hasn't guessed yet. Make sure that the input is exactly one character long and that it's a letter of the alphabet.

    d. Partition the words in the dictionary into groups by word family. The word family is the “key” in the *AssociativeArray*, and the words (already contained in *MyStrings* from having been read into the *Vector of Vectors of MyStrings* initially), are added to the *Vector* in the *AssociativeArray* which is associated with the given “key” word family.

    e. Find the most common “word family” in the remaining words, remove all words from the word list that aren't in that family, and report the position of the letters (if any) to the user. If the word family doesn't contain any copies of the letter, subtract a remaining guess from the user.

    f. If the player has run out of guesses, pick a word from the word list and display it as the word that the computer initially “chose.”

    g. If the player correctly guesses the word, congratulate her.

5. Ask if the user wants to play again and loop accordingly.

It's up to you to think about how you want to partition words into word families. Think about what data structures would be best for tracking word families and the master word list. Would an associative array work? How about a stack or queue? Thinking through the design before you start coding will save you a lot of time and headache.

Advice, Tips, and Tricks

Watch out for gaps in the dictionary. When the user specifies a word length, you will need to check that there are indeed words of that length in the dictionary. You might initially assume that if the requested word length is less than the length of the longest word in the dictionary, there must be some word of that length. Unfortunately, the dictionary contains a few "gaps." The longest word in the dictionary has length 29, but there are no words of length 27 or 26. Be sure to take this into account when checking if a word length is valid.

Deliverables

The official due date for a complete implementation of Evil Hangman is the last day of classes.
More hints and manageable subtasks will be given in later labs.

- **Grading.**
    - 12% attendance
    - 48% mystring.c
        - All functions working, all unit tests passed, no memory leaks.
        -
    - 40% hangman program works as required
        - program crashes will cause large deductions (up to 16%)
            - use gdb to find locations of crashes
            - you can get partial credit even with crashes if you document in the README file how you debugged, and what you have discovered through your debugging process (e.g., if you can determine and document where in your code the crash happens, but not why it is caused, you may get some partial credit.)
        - memory leaks will cause large deductions (up to 14%)
            - use valgrind to determine if you have any memory leaks, and to help you locate their causes.
            - you can get partial credit even with memory leaks if you document in the README file how you debugged, and what you have discovered through your debugging process (e.g., if you can determine and document where in your code the memory leaks are located, but have not figured out how to eliminate them, you may get some partial credit.)

This assignment is based on *Evil Hangman* by Keith Schwartz's "nifty" assignment at http://nifty.stanford.edu/2011/schwarz-evil-hangman