Assignment 4
Hangman and Its Evil Twin
Due: Friday, April 25, 2014, 11:59pm

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, inheritance and iterators, 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 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 each 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 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. No matter what letter you guess, your opponent can claim that she had picked the other word, and you will lose the game. 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 underlined 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 word 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, that 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 ---- 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 assignment is to write a computer program which plays a game of Hangman using both the original Hangman and this "Evil Hangman" algorithm. In particular, your program should do the following:

1. Prompt the user for whether she wants to play the Original Hangman or the Evil Hangman game. If the user enters 'O', you will play the Original Hangman game. Otherwise, you will play Evil Hangman.

2. Read the file dictionary.txt, which contains the full contents of the Official Scrabble Player's Dictionary, Second Edition. This word list has over 120,000 words, which should be more than enough for our purposes.

2. Prompt the user for a word length.

3. Prompt the user for a number of guesses, which must be an integer greater than zero.

4. If the user had entered 'O' initially, play the original Hangman game as described below:

i) Choose a word from the dictionary whose length matches the input length.

ii) 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.

iii) Prompt the user for a single letter guess, re-prompting until the user enters a letter that she hasn't guessed yet.

iv) If the user guesses a letter from the word, report the position of the letters to the user. If the letter does not exist in the word, subtract a remaining guess from the user.

v) If the player has run out of guesses, display the originally chosen word.

vi) If the player correctly guesses the word, congratulate her.

5. If the user had entered 'E' initially, play a game of Hangman using the Evil Hangman algorithm, as described below:

i) 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!).

ii) Construct a list of all words in the English language whose length matches the input length.

iii) 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.

iv) Prompt the user for a single letter guess, re-prompting until the user enters a letter that she hasn't guessed yet.

v) Partition the words in the dictionary into groups by word family.

vi) 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.

vii) 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”.

viii) If the player correctly guesses the word, congratulate her.

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

<u>Using Inheritance:</u>

In order to maximize code re-use between the two Hangman approaches, try to use inheritance as much as possible. You will notice that there is a lot of commonality between the two. For example, the code to prompt the user for number of guesses, number of words in a letter, and read in the dictionary file will be exactly identical. Create one class named **Hangman** and another named **EvilHangman** which inherits from **Hangman.** Package the common code into functions in **Hangman** and have **EvilHangman** simply call these functions. With this approach, your main function should look like the following:

```
int main(int argc, char** argv) {
  char ch;
  std::cout << “Which version do you want to play? (O) or (E): “;
  std::cin >> ch;

  if (ch == ‘O’) {
    Hangman hangman;
    hangman.run();  //Read in the file, prompt number of guesses,
                     //prompt number of letters, play!
  }
  else {
    EvilHangman hangman;
    hangman.run();  //Read in the file, prompt number of guesses,
                     //prompt number of letters,
                     //prompt whether to display possible words, play!
  }
  return 0;
}
```

<u>Advice, Tips, and Tricks</u>

You’ll need to do a bit of planning to figure out what the best data structures are for the program. There is no “right way” to go about writing this program, but some design decisions are much better than others (e.g. you can store your word list in a map, but this is probably not the best option). Here are some general tips and tricks that might be useful:

1. Letter position matters just as much as letter frequency. When computing word families, it's not enough to count the number of times a particular letter appears in a word; you also have to consider their positions. For example, “BEER” and “HERE” are in two different families even though they both have two E's in them.

Consequently, representing word families as numbers representing the frequency of the letter in the word will get you into trouble.

2. 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.

3. Generate word families from the word list. See if you can generate word families only for words that actually appear in the word list. One way to do this would be to scan over the word list, storing each word in a table mapping word families to words in that family.

Grading Criteria:

80% of the points in this assignment are for functionality. If your program runs correctly, produces the correct word family sizes in the evil version, and fills out the blanks properly, you will get full points.

20% of the points are style points. This includes decomposing your code into classes, re-using through inheritance, good indentation, comments, and so on. **Also, be careful to split your class declarations and definitions in separate header and cpp files for this assignment**. For style, you will receive either 0, 10 or 20 points:

0 points means your code is in horrible condition.
10 points means your code could use some significant improvements.
20 points means your code is in reasonably good shape.

Example Output (only shows initial output):

```
Which version do you want to play? (O) or (E): E
Please enter the length of the word to guess: 8
Please enter the number of guesses you wish to have: 10
Print the number of possible words left after each guess? (Y) or (N): Y
You have 10 guesses remaining.
--------
There are 26448 possible words left.
Please guess a letter: a
You have 9 guesses remaining.
--------
There are 13050 possible words left.
Please guess a letter: e
You have 8 guesses remaining.
--------
There are 3255 possible words left.
Please guess a letter: i
You have 8 guesses remaining.
-----i--
There are 799 possible words left.
Please guess a letter: o
You have 7 guesses remaining.
-----i—
```

Acknowledgements:

This is a modification of a “nifty” (http://nifty.stanford.edu) assignment designed by Keith Schwarz at Stanford.