# Assignment 0: Evil Hangman

**Due October 30, 11:59 PM**

**Introduction**

It's time to pull together the IOStream library and STL to build an incredible piece of software – **Evil Hangman**. Your task is to construct a computer program that dupes mortals like you and I into thinking we're playing a simple game of hangman while in reality we're facing the full power of a silicon foe.

**The Basic Idea**

Normally, a game of hangman goes like this:

1. One player chooses a secret word, then writes out a number of dashes equal to the word length.
2. The other players begin guessing letters. If a player guesses a letter that's in the word, the first player reveals all instances of that letter in the word. Otherwise, the guess is incorrect.
3. The game ends when either all the letters in the word have been revealed or when the guessers have run out of guesses.

Fundamental to the game is the fact the first player actually picks a word and accurately represents it when the other players are guessing. That way, when the other players guess letters, the player who chose the hidden word can tell whether that letter is in the word. But what happens if the player doesn't actually choose a word? What if instead that player just comes up with a list of every possible word that could fit in the spaces, then starts eliminating words whenever the other players start guessing? Provided that the player choosing the word can do this quickly enough, the other players would have no idea what was going on.

Let's see exactly how this might work in practice. Suppose that you are choosing a hangman word of length four. Rather than choosing a word, instead you build a list of all of the four-letter words in the English language. For simplicity, let's assume this is your word list:

```
ALLY  BETA  COOL  DEAL  ELSE  FLEW  GOOD  HOPE  IBEX  JAZZ  KING
```

Now, whenever other players guess letters, you simply eliminate all words containing that letter from your word list and tell the player that the letter isn't in the word. So, for example, if someone guesses E, then you'd end up with a word list that looks like this:

```
ALLY  COOL  GOOD  JAZZ  KING
```

And if they then guessed I, the word list would become

```
ALLY  COOL  GOOD  JAZZ
```

Theoretically, you can keep this up for a long time, constantly telling the guessing players that the letters they're guessing aren't in the word. If your word list is big enough (say, the full contents of the CS106 `Lexicon`), then you might be able to completely stump a guesser.

Practically speaking, however, you will almost certainly reach a point where the letter the user guesses is in every remaining word. Suppose, for example, that you have the following word list:

```
MOOD   FOLD   TOOL   PROD   GOLD   SORE
```

If the user guesses O at this point, since all the words contain the letter O, we can't simply throw out all the words containing O and tell the user that their guess was wrong. Instead, we'll have to be a bit clever about what we do. Let's take a look at all of the remaining words, highlighting the positions of the letter O in each word:

M**OO**D  F**O**LD  T**OO**L  PR**O**D  G**O**LD  S**O**RE

If you'll notice, all of the above words fall into one of three "word patterns:" -OO-, -O--, and --O-. That is, if you take any of the above words and consider the positions of the o's in that word, you'll get one of the three patterns. We see that there are two words in the family -OO-, three words in the family -O--, and one word in the family --O-. Since -O-- is the most common family, we'll throw out all words that don't match this pattern, leaving the group of words

```
FOLD   GOLD   SORE
```

And will then tell the user that they correctly guessed that there was an O at the second letter of the word. Now, we're left with the largest possible set of words to continue play with, and we can still maintain the illusion that we're playing a fair game.

As play progresses, one of two things will happen. First, the user might be smart enough to pare the word list down to one word and then guess what that word is. In this case, we'll just print out a congratulatory message and say that they guessed correctly. Second, and by far the most common case, the player will be completely stumped and will run out of guesses. When this happens, we'll just pick any remaining word out of the word list and tell them that that was the word they were guessing at all along. The irony is that the user will have no way of knowing that we were dodging guesses the whole time – it looks like we simply picked an unusual word and stuck with it the whole way.

**The Assignment**

Using only standard C++, you are to write an implementation of the Evil Hangman game using the aforementioned algorithm. You will be building the entire program from scratch, so feel free to use any code off of the course website, especially for user input validation. Please be sure to cite your sources!

Your program should do the following:

1. Prompt the user for a word length, reprompting as necessary until the user 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 at an integer greater than zero. Don't worry about the case where the user enters very large numbers, since guesses above 26 can't make a difference.
3. Prompt the user for whether she wants to have a running total of the number of words remaining in the word list. This defeats the purpose of the game, but is useful for testing (and grading!)

4. Play a game of hangman using the Evil Hangman algorithm, as described below:
    1. 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.
    2. Prompt the user for a single letter guess, reprompting 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 (you can use the `isalpha` function from the `<cctype>` header to determine if a character is a letter).
    3. If there is at least one word left in the word list that doesn't contain that letter, remove all words from the word list that contain that letter and tell the user that the guess was incorrect.
    4. Otherwise, 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 to the user.
    5. 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."
    6. If the player correctly guesses the word, congratulate her.
5. Ask if the user wants to play again and loop accordingly.

Your program should use the file `dictionary.txt`, available on the CS106L website, as its master word list. Do not read the file multiple times – it's over one megabyte, which makes for some rather lengthy load times. Instead, just create a master word list that you won't change in the course of the program, then copy the relevant sections when needed. Don't worry about the case where your program can't find the `dictionary.txt` file.

**Advice, Tips, and Tricks**

Since you're building this project from scratch, 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 `stack<string>`, but this is clearly not the best option). As always, feel free to email me if you have any questions.

Here are some general design tips that might be useful:

1. When reading in the words from the file, make sure you're reading every word. You can test this by replacing `dictionary.txt` with a file that's a single word long. Don't worry about the case where the dictionary is empty or missing.
2. Make sure that you handle invalid user input appropriately. Use functions like `GetInteger` and `GetLine` to read input, and be sure to confirm that any user-supplied data is valid before processing it. These touches will make your program look more professional and are important if you plan on demonstrating it to your friends.
3. When considering word families, letter position matters just as much as letter frequency. Thus "BEER" and "HERE" are in two different families even though they both have two E's in them.
4. Be careful when iterating over container classes and removing elements. If you remove an element from a container, you *invalidate* all iterators that point to that element. Invalid iterators may point to garbage data, and their `*` and `++` operators are not guaranteed to work as expected. If you have an iterator `itr` to an element that you want to remove, and you're also using `itr` inside of a loop, you can prevent `itr` from being invalidated by using this technique:

```
while(itr != myContainer.end())
{
    if (/* some condition */)
    {
        containerType::iterator toRemove = itr;
        ++itr;                          // Advance itr.
        myContainer.erase(toRemove); // Remove element.
    }
    /* Otherwise, we didn't advance itr yet, so do it here. */
    else ++itr;
}
```

5. While efficiency is not critically important, your program should be responsive to user input. If your program takes awhile to do all of the required bookkeeping, the user might get suspicious that something fishy is afoot. With good choices about which container classes to use, you shouldn't have to worry about this. That said, if you're experiencing delays when running your program (more than three seconds to respond to input), you probably have made some suboptimal design decisions and might want to rewrite parts of your program.

**Deliverables**

To submit the assignment, email any source files to htiek@cs.stanford.edu. If you've added any extensions or special features I should be aware of, let me know in the body of your email.

Good luck!