Assignment 1: Evil Hangman

Due May 6, 11:59 PM

Introduction

The idea of making computers that play games is a noble one. The ability to codify a game into a concrete set of rules represents a triumph of human intellect, and the power to imbue computers with abstract strategy and reasoning bespeaks a thorough technological prowess. But like their human masters, computers can be made to cheat. In this assignment, you will build a mischievous program that bends the rules of *Hangman* to trounce its human user time and time again. In doing so, you'll cement your skills with the streams library and STL, and will end up with an enormously entertaining piece of software (at least, from your perspective.) ☺

Hangman is played as follows:

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 either when 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 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 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 based on the other players' guesses? Provided that the player choosing the word can do this quickly enough, the other players would almost certainly lose but would have no idea that anything suspicious was afoot.

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 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
```

Now, suppose the other players guess the letter E. We can perform one of two possible actions. First, we can simply eliminate all words that contain E and tell the other players that they guessed incorrectly. For example, if we decide to remove all words containing E from the list, then our word list would become

```
ALLY  COOL  GOOD
```

However, removing all words containing the guessed letter from the word list might not be a good idea. For example, given these three words as a word list, if the user were to guess O, removing all words containing the letter O would give us a length-one word list. If we instead remove all words *not* containing O from the list, then we get the following list:

```
COOL  GOOD
```

Which has two words instead of one.

We have to be careful about about what words we eliminate, though, because we have to keep up the illusion of a fair game. For example, consider the following word list:

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

If the other players guess O, we cannot simply eliminate all words containing O because we would end up with an empty word list. This means that we need to tell the other players that the word we've chosen contains at one O. However, the rules of *Hangman* dictate that we can't just say whether the word contains an O; we also need to specify the *positions* of the letter O. How should we choose which spaces we reveal as O's? One simple approach would be to reveal the letters in the positions that maximize the size of the resulting word list. To see how this works, let's take a look at the remaining words, highlighting the positions of the letter O:

```
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 families:" `-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 these 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, yielding the following word list:

```
FOLD  GOLD  SORE
```

We 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.

The above example illustrates the general algorithm for paring down the word list. Whenever the user guesses a letter, we'll construct the word families for each remaining word in the word list, find the most common family, and then eliminate all words that aren't in that family from the word list. At each step in the program, this keeps the maximum number of words remaining in the word list. While this is not an optimal strategy, in practice it works very well and isn't too difficult to implement.

To give a concrete example of how this algorithm works, let's return to our initial word list:

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

Now, if the user guesses the letter E, then we compute the word families for the above words with respect to E. Highlighting the positions of the letter E gives us

```
ALLY  B**E**TA  COOL  D**E**AL  **E**LS**E**  FL**E**W  GOOD  HOP**E**  IB**E**X
```

There are five word families here:

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

Notice that the collection `----` of words not containing the letter E is a valid word family – this is very important!

Of the five families listed above, the word family `----` contains the most words, so we update our word list by eliminating all words not in this family. This yields the word list

```
ALLY  COOL  GOOD
```

And we would report to the user that her guess was incorrect.

There are two possible outcomes of this game. 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 just print out a congratulatory message and say that she 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 pick a random word out of the word list and tell the user that it was the word she was 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 implement the Evil Hangman game using the aforementioned algorithm. You will be building the entire program from scratch, so feel free to use code from the course reader or from lecture, 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.

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.

* That is, without using the CS106B/X libraries.

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. 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.

4. 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."

5. 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.

Efficiency Concerns

The program you will be writing maintains a grand illusion: it pretends to play a game of *Hangman*, but instead does something more nefarious behind-the-scenes. Consequently, you should try to make your program as responsive as possible. If players have to wait three or more seconds after entering a letter, they're almost certain to notice that something is awry. The illusion will be broken, and the beauty of your program will be lost.

One of the major selling points of the STL is its incredible efficiency. While the STL is certainly much faster than the CS106 ADTs, that does not mean that code written using the STL will automatically be efficient. In particular, there are some operations you will want to shy away from then using the STL, in particular:

- *Passing or returning containers by value*. Deep-copying an STL container is fairly expensive. If you need to pass a container into a function, you should definitely pass it by reference. Similarly, if you need to return a container from a function, consider instead taking it in by reference and updating it inside the function.

- *Inserting or deleting elements at arbitrary positions in a `vector` or `deque`.* The `vector` container is designed to only grow and shrink at its end, though the `deque` can grow and shrink at both ends. However, neither container is designed to support insertion and deletion at arbitrary points. Inserting or deleting elements from the middle of a `vector` or `deque` will greatly slow down your program. If you want to remove a collection of elements from a `vector` or `deque`, you are better off creating a new `vector`/`deque` of the elements that should remain, then setting your original `vector`/`deque` to this new `vector`/`deque`.

Although efficiency is important in *Evil Hangman*, never forget the wisdom of programmers past:*

Premature optimization is the root of all evil.

* This has been attributed to both Don Knuth and Tony Hoare, both of whom are legends in the field.

There is an inherent tradeoff between *readable code* and *optimized code*. Code that is clean and easy to follow often contains subtle sources of inefficiency, while code that is optimized is often impossibly difficult to read. As an extreme example, consider the following piece of code, which is known as *Duff's Device*:

```
int n = (count + 7) / 8;

switch (count % 8)
{
case 0: do {  *to = *from++;
case 7:       *to = *from++;
case 6:       *to = *from++;
case 5:       *to = *from++;
case 4:       *to = *from++;
case 3:       *to = *from++;
case 2:       *to = *from++;
case 1:       *to = *from++;
           } while (--n > 0);
}
```

This code* is entirely equivalent to the following:

```
copy(from, from + n, to);
```

On some machines, Duff's Device performs slightly more efficiently than a hand-written `for` loop or a call to the `copy` algorithm due to the way that the processor handles loops. Of course, Duff's Device is impossibly hard to read, and even the best of programmers would have no way of deciphering it without extensive commenting.

As you write your Evil Hangman implementation, try to get the program working **before** you worry about runtime efficiency. If you try to optimize the code as you write it, you are almost certainly going to end up with code that is harder to read and maintain, which will make debugging a nightmare. Moreover, if you optimize as you go, there's no guarantee that your optimization will have any noticeable effect. If you spend an enormous amount of time optimizing parts of the code that aren't actually contributing to the overall runtime, you won't change your program's efficiency appreciably, and you'll have increased your code complexity unnecessarily.

The moral of the story is this: first get your program working, *then* worry about making it run quickly. Chances are, your program will be fast enough the first time you write it and you won't need to optimize it. If for some reason this isn't the case, feel free to send me an email and I can try to offer some advice.

* Retrieved from http://foldoc.org/Duff%27s+device

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 probably 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. *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.

2. *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.

3. *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.

4. *Don't explicitly enumerate word families*. If you are working with a word of length n, then there are 2^n possible word families for each letter. However, most of these families don't actually appear in the English language. Rather than explicitly generating every word family whenever the user enters a guess, see if you can generate word families only for words that actually appear in the word list. Provided that you pick a good representation for your word list, this should be considerably easier and much faster than explicitly enumerating all of the word families.

5. *Be careful when iterating over containers while 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're working with a sequence container (`list`, `vector`, or `deque`), you can avoid issues with invalidated iterators by updating the value of the iterator you pass to `erase` to the iterator returned by `erase`. For example:

```
while(itr != myContainer.end()
{
    if (/* some condition */)
        itr = myContainer.erase(itr);
    else
        ++itr;
}
```

When working with associative containers (`map`, `set`) , 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;
}
```

6. *Don't subtract guesses for correct answers*. If the user guesses a letter that is revealed in the word, don't deduct a remaining guess from her. Otherwise, it would be virtually impossible to guess a word of length fourteen with only eight guesses.

7. *Remember to only use standard C++*. Because this programming assignment is designed to help you acclimate to C++, you must **not** use any of the libraries we provide you in CS106B/X. This means that you cannot use `genlib.h`, `simpio.h`, the `Lexicon` or `Scanner` classes, the CS106 ADTs, etc. However, any code we've written in lecture is fair game, so feel free to use the `GetLine` and `GetInteger` functions covered in class.

Extensions

The algorithm outlined in this handout is only one possible way to implement Evil Hangman. There are many other ways that you can go about paring down the word list. If you have any suggestions for an improved algorithm, feel free to implement it and include it with your solution. If you end up writing a truly devious program, I'd be glad to demo it in class.

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!