Evil Hangman

This lab assumes that you have completed all steps involved in lab1 through 9 and provides hints and targets for finishing more of the lab given in lab9.

We will be working on a portion of the whole game for this week.

Let's examine a sample play of the game.

In this example, the user has selected to play the game with a word that is 7 letters long and they want to have only 1 guess. What this means is that the user only has one incorrect guess but if they happen to guess a letter correctly then they do not lose their guess. The user selects the letter 'z' as their guess. Behind the scenes, the program reads through every 7 letters word in the dictionary and examines them individually (**lab8 you worked on creating an array of vector of strings. Each entry in the array represents the word length. For example, at index 3, we will find all the words that have a length of 3 stored inside a vector of strings**). Each word can be sorted into a family of words that all have a similar pattern. For example, there are 21193 words in the dictionary that are 7 letters long and do not contain any occurrence of the letter 'z'. There are 6 words that end in a single 'z', 57 words that have a single 'z' as the next to last character and so on. Our task for this week is to simply write the function that can determine what family each word belongs to when given a particulate guess. Let's look at the function here and then explain in more details how to use it.

```
//Precondition:current_word_family, new_key and word are all handles to valid
//   MY_STRING opaque objects.  guess is an alphabetical character that can be either
//   upper or lower case.
//Postcondition:  Returns SUCCESS after replacing the string in new_key with the key
//   value formed by considering the current word family, the word and the guess.
//   Returns failure in the case of a resizing problem with the new_key string.
Status get_word_key_value(String_Ptr current_word_family, String_Ptr new_key, String_Ptr word, char guess);
```

The function takes a number of inputs: **current_word_family, new_key, word, guess:**

- **current_word_family:** represents the current family the word belongs to. For example, when we first start the program, the default word_family would be for a 7 letters word: **--- ----**

- **new_key:** represents the new word family for the word. This will be modified inside the function and determined inside the function
- **word:** a word that you want to define its **new_key**
- **guess:** the letter that the user entered us the guessing.

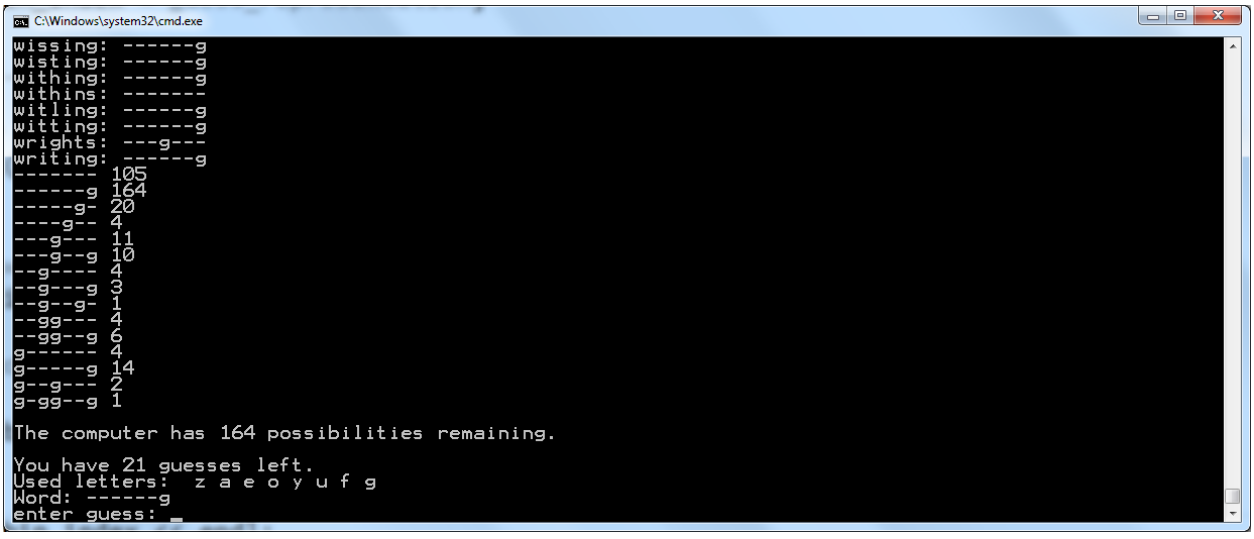
Let's have a simple example about this function. Let's say that you are working with three letter words and you have in your dictionary the words: **car, cat, hat.** Let's say the **guess = c,** the variables to the function then would be as follow:

- **current_word_family = ---**
- **word = car**
- **guess = c**
- **new_key (will be generated inside the function) = c--.** Since the word **car** has the letter **c** located in the first position.

However, when we run the function on another word in the dictionary, **hat**, in that case the **new_key = --- (similar to the input family word).** Since the word hat does not have any c letters.

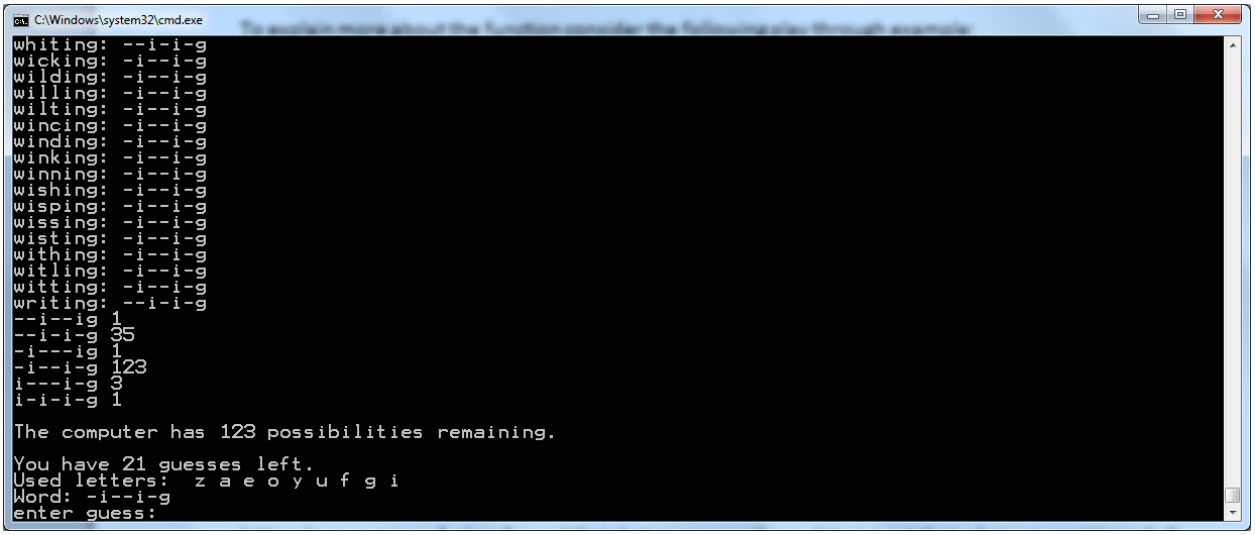
Another example, if we had our input word family as: **current_word_family = -a-.** The input **guess= t,** and the **word = hat.** In that case, the generated **new_key= -at**

To explain more about the function, consider the following play through example with added information being displayed (**the following example illustrates how the program works and how to use the function but you are not going to provide such output, you will be able to calculate the word numbers in later labs**):

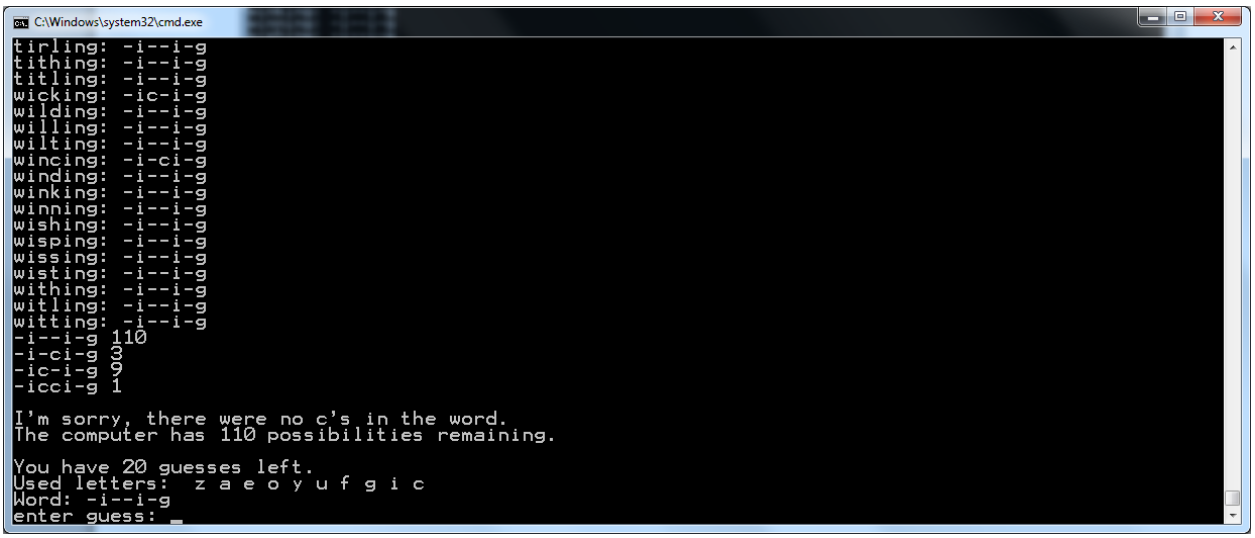


In this game, the user started out with unfair number of guesses (28?). They have guessed the letters (z, a, e, o, y, u, f, g) so far and their last guess was the g. It turns out that when we partitioned all of the words that we had left that the largest bin remaining was actually the one that had a g at the end of the word. Following our rule that we always choose the largest word bin we choose the bin that has 164 words in it and at this point must concede to the user that their guess of 'g' was actually in the word. Our internal collection of words now contains only words that are 7 letters long, do not contain any of the guessed letters given above (except for

‘g’) and has exactly one occurrence of ‘g’ at the end of the word. Let us suppose that the user now selects the letter ‘i’.



Here, not too surprisingly considering we used all of the other vowels and y, the letter ‘i’ occurs in every possible word in our list. All of the words fall into one of six categories or word families where the category -i--i-g has 123 words in it and that is the one we select. Suppose now that the user selects the letter ‘c’.



Each guess takes all of the possible words remaining and generates a new key or word family for each word. The word family or key value is always the same as the current_word_family string except that one or more of the dashes could be changed to the guessed letter. These changes occur if there is an occurrence of the guessed letter at the corresponding position in the word we are examining. You will notice in the above example that when we guessed the letter ‘c’ the key generated for the word “wicking” became -ic-i-g and the key for the word “wincing” became -i-ci-g respectively. Your function takes a given previous key, in our context it is the string that is being displayed to the user as the correct letters so far. It also takes the word that you are

considering, like “wishing”, and the guessed letter (‘c’ in our case). All of these input values are used to compute a new string value for the parameter new_key.

TA CHECKPOINT 1: Demonstrate to your TA that your function generates the following keys from the given input (Use init_c_string to set up each case)
Given:

Old Key (current word family)	Word	Guess		New Key
---	The	T		t--
-----	Truck	r		-r---
--ppy	happy	h		h-ppy
--e---e	awesome	z		--e---e