CS 133, Spring 2023
Programming Project #2: Letter Guessing Game AI (20 points)
Due Friday, April 21, 2023, 11:59 PM

Thanks to Keith Schwarz, Stuart Reges and Marty Stepp for pieces of this project.

This program focuses on using `set/unordered_set` and `map/unordered_map` collections. Turn in files named `WordGuesser.h` and `WordGuesser.cpp` on the Project section of the course website. You will need the starter code zip from the course website that contains `dictionary.txt, dictionary2.txt` and `letterGuessingMain.cpp`. In this project you are going to write a class that keeps track of the state of a game in which the user guesses letter in a word or phrase. However, this won't be any ordinary letter guessing game! The program you will write is going to cheat.

Background:

There are many varieties of word and letter guessing games. You have likely played or heard of Wordle, Spelling Bee, anagrams, crosswords, Wheel of Fortune and many more.

In the game we will implement, one player (the computer in our case) is expected to pick a word and the other player (computer user in our case) is supposed to guess the word by guessing one letter at a time until they can figure out what the whole word will be and guessing that. The amount of guesses is usually capped, similarly to how guessers only get 20 guesses in a game of 20 Questions.

In our game, the AI we write will make the computer delay picking a word until it is forced to. As a result, at any given point in time there will be a set of words that are currently being used by the computer. Each of those words will have the same pattern of known letters which will be displayed to the user.

We have provided you with a client program called `letterGuessingMain.cpp`. It handles all user interaction. The class you will write, `WordGuesser` will keep track of the state of the game. You won't be writing any code that prints information or reads information from the user because all of that code is in `letterGuessingMain.cpp`.

Implementation Details:

You will write one class; `WordGuesser`. You must use the C++ standard library's `map` and `set` classes. It must be possible to call the member functions **multiple times in any order** and get the correct results each time. Your class must have the constructors/member functions below. However**, it is up to you to decide which parameters, returns and functions should be references and/or `const`.**

`WordGuesser` Class *(for you to implement)*:

`WordGuesser`

Parameters: `vector<string>` - contains the full dictionary of words
`int` - the length of the word the user will guess
`int` - the maximum number of allowed guesses

Use the passed in values to initialize the state of the game. The set of words should initially be all words from the dictionary that are of the given length. The constructor should throw string exception if the passed in length is less than 1 or if the passed in max is less than 0.

`getWords`

Returns: `set<string>*` - contains the current words that could be made with the current pattern

The client calls this member function to get access to the current set of words being considered by the letter guesser.

`getGuessesLeft`

Returns: `int` - the total number of remaining guesses

The client calls this member function to find out how many guesses the player has left.

`getGuesses`

Returns: `set<char>` - the set of letters that the user has already guessed

The client calls this member function to find out the current set of letters that have been guessed by the user. These should be in alphabetical order.

`getPattern`

Returns: `string` - a string containing all of the previously guessed letters in their correct spots and the other spaces displayed as "_".

Return the current pattern to be displayed for the guessing game taking into account guesses that have been made. Letters that have not yet been guessed should be displayed as a dash and there should be spaces separating the letters. There should be no leading or trailing spaces. This member function should throw a string exception if the set of words is empty.

`record`

Parameters: `char` - the character the user guessed

Returns: `int` - the number of characters replaced

This is the member function that does most of the work by recording the next guess made by the user. It should throw a string exception if the number of guesses left is not at least 1. Using this guess, it should decide what set of words to use going forward. It should return the number of occurrences of the guessed letter in the new pattern and it should appropriately update the number of guesses left. This member function should also throw a string exception if the list of words is empty. It should throw a string exception if the list of words is nonempty and the character being guessed was guessed previously.

`record` algorithm

As noted earlier, this version of hangman cheats. It doesn't actually pick a word until it needs to. Suppose that the user has asked for a 5-letter word. Instead of picking a specific 5-letter word, it picks all 5-letter words from the dictionary. But then the user makes various guesses, and the program can't completely lie. It has to somehow fool the user into thinking that it isn't cheating. In other words, it has to cover its tracks. Your `WordGuesser` object should do this in a very particular way every time the `record` member function is called.

Example:

Suppose that the dictionary contains just the following 9 words:

`[ally, beta, cool, deal, else, flew, good, hope, ibex]`

Now, suppose that the user guesses the letter `E`. You now need to indicate 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. Every word in the set falls into one of five "word families:"

- "`- - - -`": which is the pattern for [`ally, cool, good`]
- "`- e - -`": which is the pattern for [`beta, deal`]
- "`- - e -`": which is the pattern for [`flew, ibex`]
- "`e - - e`": which is the pattern for [`else`]
- "`- - - e`": which is the pattern for [`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 set of words to:

[`ally, cool, good`]

Since you didn't reveal any letters, you would count this as a wrong guess.

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

- "`- o o -`": containing [`cool, 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 set of words to

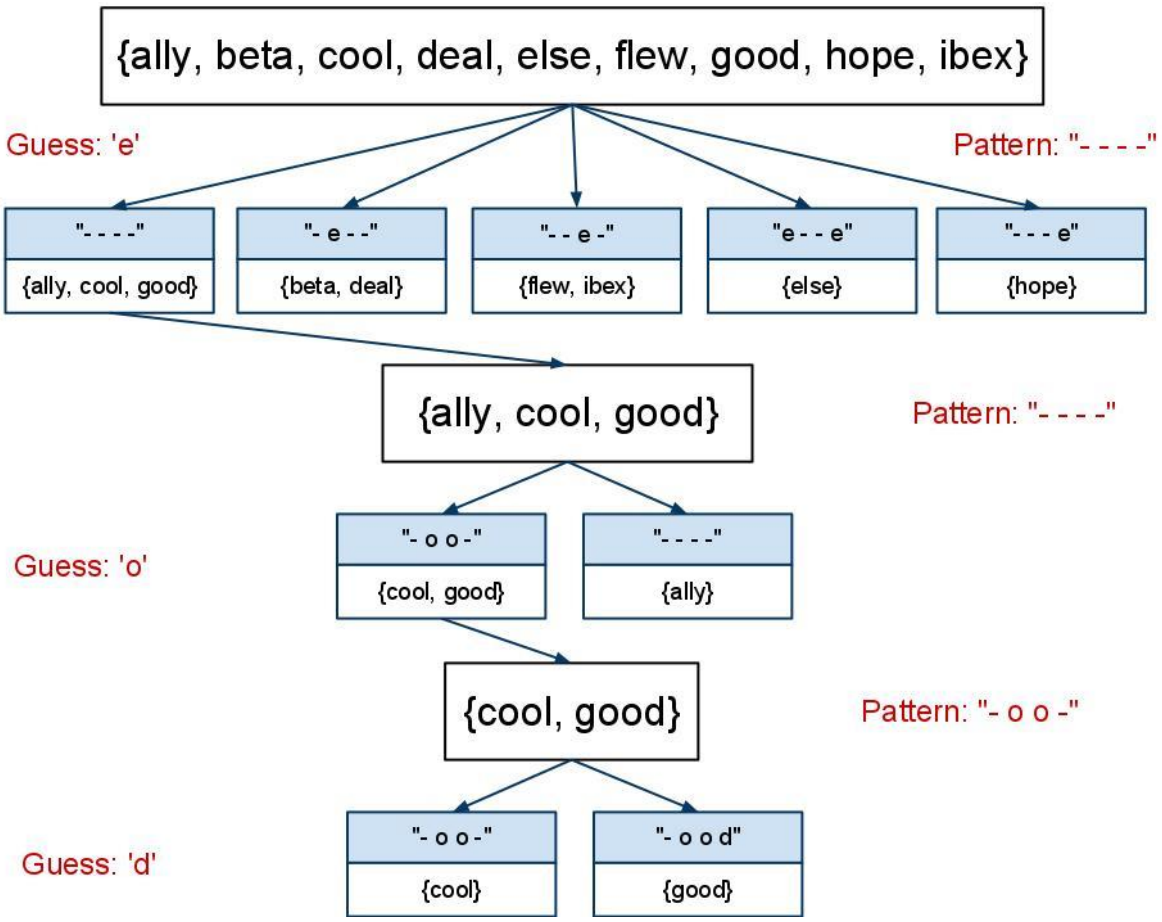
[`cool, good`]

In this case, you would count this as a correct guess because there are two occurrences of `O` in the new pattern.

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 "`- o o -`" 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 and you would count this as an incorrect answer.

To implement this strategy, you should use a map (it is up to you whether to use an `unsorted_map` or `map`; choose the one that will be most efficient while still providing the functionality you need). The keys will be the different patterns for each word family. Those keys should map to a set of words that have that pattern. For each call on record, you will find all of the word families and pick the one that has the most elements. This will become the new set of words for the next round of the game. If there is a tie (two of the word families are of equal size), you should pick the one that occurs earlier in the map (i.e., the one whose key comes up first when you iterate over the key set).

You can see this whole example in diagram form below:



Implementation Requirements:

You are expected to do some error checking, as outlined in the descriptions of the public member functions. However, you do not need to check for all possible errors. You may assume that the vector of words passed to your constructor is legal in that it will be a nonempty list of nonempty strings composed entirely of lowercase letters. You may assume that all guesses passed to your `record` member function are lowercase letters.

To match our expected output, you should make sure that when **you iterate over your maps and sets you will always do so in sorted order**.

Hints:

You will find two constants at the top of `letterGuessingMain.cpp` that you may want to change:

`DICTIONARY_FILE`: the name of the file that the dictionary of words will be read from. This starts off as `dictionary2.txt`, a very small file designed to help you test your code. However, once your code

works with `dictionary2.txt`, switch this to `dictionary.txt` to try the game out with a much larger dictionary. `dictionary.txt` contains all the words in the official English scrabble dictionary.

`SHOW_COUNT`: By default it is set to `false`. By setting it to `true`, you will be shown how many words there are in the current set of words as you play the game. You may find this very helpful for debugging.

Development Strategy:

We suggest the following development strategy for solving this program:

1. Start by writing your header file and empty (or containing a default simple return) implementations for all of the required functions. Make sure you can compile the provided code.
2. Implement your constructor. Use dictionary2.txt as it is much smaller and easier to test with than dictionary.txt. Add prints to your constructor so you can make sure your object is getting constructed correctly.
3. Implement `getWords`, `getGuessesLeft` and `getGuesses`.
4. Implement `getPattern`. This will be a little more challenging as you will need to build up the pattern string.
5. Last, implement `record`. This is the hardest function and most of the meat of the project. Follow the algorithm above adding in one piece at a time.

Style Guidelines and Grading:

To receive full credit on this project, you must follow the best practices that we discussed in 131 and 132 as well as the new style guidelines we have added this quarter. These include but are not limited to:

- Use private helper functions to help reduce redundancy and capture structure. In particular, **you should not have any member functions that have more than 20 lines of code in their body (**not counting blank lines and lines that have just comments or curly braces**).** If you have a member function that requires more than 20 lines of code, then you should break it up into smaller member functions.
- Eliminating other redundancy with loops, variables and good `if/else` factoring
- Always using good boolean zen.
- Using C++ naming conventions appropriately and giving representative names to variables.
- Indenting your code correctly and keeping all lines 100 characters long or shorter.
- Avoiding making a value a member variable when it can instead be a local variable.
- The only publicly accessible elements of your class should be the member functions outlined in this document. All member variables and additional functions should be private.
- Implementing the required functionality in a reasonably efficient manner.
- Making good choices when selecting between ordered and unordered collections.
- Using `const`, references and pointers well.
- Commenting descriptively at the top of your class, each function, and on complex sections of your code in your header file. These comments should explain each function's behavior, parameters, return, pre/post-conditions, and any exceptions thrown. Write descriptive comments that explain error cases, and details of the behavior that would be important to the client. Your comments should be written in your own words and not taken verbatim from this document. You must also include a comment with your name and a very brief description on your cpp files. These should also include comments within your complex functions explaining anything complicated in your implementation.