Instructor created

15.8 Project 03 - Lab 8 - Tue 8am, SEL 2263

Project 03 - A Dynamically Evil Word Guessing Game

Introduction to the Classic Word Guessing Game

Writing computer programs to play games as optimally as possible is a difficult task. When humans sit down to play a game, we can draw on past experience, adapt to our opponents' strategies, and learn from our mistakes. Computers, traditionally, 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. Modern advances in artificial intelligence are actively revolutionizing optimal strategies for many games.

While there are many viable strategies for building competitive computer game players, our approach here is a nefarious one centered on a mischievous program that bends the rules of of a classic word game to trounce its human opponent time and time again. In building this program, you'll cement your skills with dynamic memory management, C-string operations, command-line arguments, and general programming savvy. Plus, you'll end up with a piece of software that is highly entertaining.

The game of interest here is a classic word guessing game that goes by many names: Hangman, Stickman, Snowman, Letter Box, Wheel of Fortune, etc. In case you aren't familiar with the classic game, the rules are as follows:

1. Player One (a.k.a. the *chooser*) chooses a secret word, then writes out a number of dashes equal to the word length.
2. Player Two (a.k.a. the *guesser*) begins guessing letters. Whenever the *guesser* guesses a letter contained in the hidden word, the *chooser* 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 wrong guesses.

A nuance of Step 1 that is fundamental to the game is that the *chooser* accurately represents the word chosen throughout the gameplay. That way, when the *guesser* guesses a letter, the *chooser* can reveal whether or not the guessed letter is in the word. But what happens if the *chooser* is evil and doesn't do this...?

Introduction to the Evil Word Guessing Game

If the *chooser* is evil and doesn't play by the rules, the *chooser* can have an enormous advantage, especially if the *chooser* is actively keeping a full record of ALL possible words AND can quickly analyze the word list for an optimally evil strategy to make the game as difficult as possible for the *guesser*.

As an example, suppose that you are the *guesser*, and you have revealed a set of letters until you arrive at the following word pattern 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 *chooser* is playing fairly, then you have a 50-50 chance of winning this game if you guess 'A' or 'U' as the missing letter. However, if your opponent is evil 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 to have picked the other word, and you will lose the game. That is, if you guess that the word is “doable,” the *chooser* can pretend to have committed to “double” the whole time, and vice-versa.

Let's illustrate this technique with an example. Suppose that you are playing the game 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 are evil and 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 that you've “chosen” are E's. Of course, you haven't chosen a word, and so you have multiple options about where you reveal the E's. Here's the above word list, with E's highlighted in each word:

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

Notice that every word in the word list falls into one of five “patterns:”

- ---- which is the pattern for ALLY, COOL, and GOOD.
- -E-- which is the pattern for BETA and DEAL.
- E--E which is the pattern for ELSE.
- --E- which is the pattern for FLEW and IBEX.
- ---E which is the pattern for HOPE.

Also notice how the word list is in alphabetical order AND the pattern list is printed in the order the patterns first appear in the alphabetical word list.

Since the letters you reveal have to correspond to some word in your word list, you can choose to move forward with any one of the above five patterns. There are many ways for an *evil chooser* to pick which pattern 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. For this project, in the interests of simplicity, we adopt the latter approach and always choose the pattern ----, which reduces the word list down to:

```
ALLY  COOL  GOOD
```

Since no letters were revealed, the guess is wrong and your opponent has used up one of their guesses.

Let's continue with our example gameplay using this evil strategy. Given this three-word word list, if your opponent guesses the letter O, then you would break your word list down into two patterns:

- ---- which is the pattern for ALLY.
- -OO- which is the pattern for COOL and GOOD.

Once again, notice how the updated word list is in alphabetical order AND the pattern list is printed in the order the patterns first appear in the alphabetical word list.

The second pattern has a family that is larger than the first, and so you choose it, revealing two O's in the word and reducing your list down to:

```
COOL  GOOD
```

Now, 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 the word list apart into patterns, you'll find that there's only one pattern -OO-, in which T appears nowhere and which contains both COOL and GOOD. Since there is only one possible pattern here, it's trivially the largest family, and by picking it you'd maintain the word list you already had. Of course, the guesser loses another guess by picking 'T' here.

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, congratulations are in order as that's an impressive feat considering the scheming you were up to! Second, your opponent may be completely stumped and will run out of guesses. When this happens, pick the first word alphabetically in your word list and present it as the word 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; instead, it looks like you simply picked an unusual word and stuck with it the whole way.

The Program

Starter Code, Command-Line Arguments, File Reading, and Interactive User-Input

The starter code provided in the IDE below contains a **Pattern** struct definition, three function prototypes, and MANY helpful comments on each of the required tasks, which are explained in more detail in the following sections. As you read through the required tasks, it is a good idea to reference the starter code for context. Specifically, you are required to build an array of **Pattern** structs for each letter guess using the **Pattern** struct defined in the starter code, and you must write the three functions as described in the starter code and in the section below.

The program you write involves three types of inputs:

- Command-line arguments are used to specify game details, such as the **word size** (default value is 5), the **number of guesses** (default value is 26), and flags to turn on various reporting modes (default is OFF):
  - **stats mode** prints out statistics about the dictionary and the number of possible words remaining during gameplay,
  - **word list mode** prints the full list of possible words before every guess,
  - **letter list mode** prints out a list of previously guessed letters before each new guess,
  - **pattern list mode** prints out a list of all unique patterns in the word list, together with the frequency of each pattern and the number of changes in the pattern due to the latest guess (i.e. the number of occurrences of the guessed letter in that pattern), and
  - **verbose mode** turns ON all other modes (i.e. stats mode, word list mode, letter list mode, and pattern list mode all turned ON).

- File-reading is used to build the original word list. The provided file *dictionary.txt* 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.
- Interactive user-input is used to read in the user's guesses during gameplay. The user should only enter lower case letters as their guesses. At any point, the user can enter '#' to end the game. Two provided files, namely *alphabetIn.txt* and *randomLets.txt*, contain guesses that span the entire alphabet in different orders. You can use these files with a redirection operator to quickly run the program to guarantee a complete run of the game without needing interactive input. The autograder uses this approach to test your program upon submission.

As an example, suppose you compile the program as follows: **gcc main.c -o game.out**

You then have option on how to run the game:

- default settings (word size of 5, 26 guesses, all special print modes OFF) and interactive guesses: **./game.out**
- word size of 9, 15 guesses, all special print modes ON and interactive guesses: **./game.out -n 9 -g 15 -v**
- default settings, but use a predefined set of guesses: **./game.out < randomLets.txt**
- word size of 21, 12 guesses, stats and letter list mode ON, predefined guesses: **./game.out -s -g 12 -l -n 21 < alphabetIn.txt**

Note that the order of the command-line arguments is arbitrary. If a command line argument is invalid, provide the following message and terminate the program:

```
Invalid command-line argument.
Terminating program...
```

If an invalid value is provided for the word size, provide the following message and terminate the program:

```
Invalid word size.
Terminating program...
```

If an invalid value is provided for the number of guesses, provide the following message and terminate the program:

```
Invalid number of guesses.
Terminating program...
```

Demo Program Executable

A working version of the program is provided as **demo.exe**. You can run it to play the game, view the expected behavior of the program, and/or check formatting of outputs. It accepts all of command-line arguments that your program should. To run **demo.exe**, you first must use the UNIX command **chmod** to change the permissions of the file to allow all users to execute it, as follows:

```
➜ chmod a+x demo.exe
➜ ./demo.exe –n 6 –g 15 –v
```

Sample Gameplay - Full Game with Word Size 9 and Letter List Mode

Note that this has mixed input/output, where the characters entered by the user are included after each "**Guess a letter (# to end game):**" prompt.

```
➜ gcc main.c –o program.exe
➜ ./program.exe –n 9 –l
Welcome to the (Evil) Word Guessing Game!

Game Settings:
  Word Size = 9
  Number of Guesses = 26
  View Stats Mode = ON
  View Word List Mode = OFF
  View Letter List Mode = ON
  View Pattern List Mode = OFF
The word pattern is: ---------

Number of guesses remaining: 26
Previously guessed letters:
Guess a letter (# to end game): r
Oops, there are no r's. You used a guess.
The word pattern is: ---------

Number of guesses remaining: 25
```

```
Previously guessed letters: r
Guess a letter (# to end game): s
Oops, there are no s's. You used a guess.
The word pattern is: ---------

Number of guesses remaining: 24
Previously guessed letters: r s
Guess a letter (# to end game): t
Oops, there are no t's. You used a guess.
The word pattern is: ---------

Number of guesses remaining: 23
Previously guessed letters: r s t
Guess a letter (# to end game): l
Oops, there are no l's. You used a guess.
The word pattern is: ---------

Number of guesses remaining: 22
Previously guessed letters: l r s t
Guess a letter (# to end game): n
Good guess! The word has at least one n.
The word pattern is: -------n-

Number of guesses remaining: 22
Previously guessed letters: l n r s t
Guess a letter (# to end game): a
Oops, there are no a's. You used a guess.
The word pattern is: -------n-

Number of guesses remaining: 21
Previously guessed letters: a l n r s t
Guess a letter (# to end game): i
Good guess! The word has at least one i.
The word pattern is: ------in-

Number of guesses remaining: 21
Previously guessed letters: a i l n r s t
Guess a letter (# to end game): o
Oops, there are no o's. You used a guess.
The word pattern is: ------in-

Number of guesses remaining: 20
Previously guessed letters: a i l n o r s t
Guess a letter (# to end game): w
Oops, there are no w's. You used a guess.
The word pattern is: ------in-

Number of guesses remaining: 19
Previously guessed letters: a i l n o r s t w
Guess a letter (# to end game): m
Oops, there are no m's. You used a guess.
The word pattern is: ------in-

Number of guesses remaining: 18
Previously guessed letters: a i l m n o r s t w
Guess a letter (# to end game): c
Oops, there are no c's. You used a guess.
The word pattern is: ------in-

Number of guesses remaining: 17
Previously guessed letters: a c i l m n o r s t w
Guess a letter (# to end game): b
Good guess! The word has at least one b.
The word pattern is: --b---in-
```

```
Number of guesses remaining: 17
Previously guessed letters: a b c i l m n o r s t w
Guess a letter (# to end game): u
Good guess! The word has at least one u.
The word pattern is: --bu--in-

Number of guesses remaining: 17
Previously guessed letters: a b c i l m n o r s t u w
Guess a letter (# to end game): e
Good guess! The word has at least one e.
The word pattern is: -ebu--in-

Number of guesses remaining: 17
Previously guessed letters: a b c e i l m n o r s t u w
Guess a letter (# to end game): d
Good guess! The word has at least one d.
The word pattern is: debu--in-

Number of guesses remaining: 17
Previously guessed letters: a b c d e i l m n o r s t u w
Guess a letter (# to end game): g
Good guess! The word has at least one g.
The word pattern is: debugging

You solved the word!
The word is: debugging
Game over.
➜
```

Sample Gameplay - Partial Game with Word Size 21 and Word List Mode

Note that this has mixed input/output, where the characters entered by the user are included after each "**Guess a letter (# to end game):**" prompt.

```
➜ gcc main.c -o program.exe
➜ ./program.exe -w -n 21
Welcome to the (Evil) Word Guessing Game!

Game Settings:
  Word Size = 21
  Number of Guesses = 26
  View Stats Mode = OFF
  View Word List Mode = ON
  View Letter List Mode = OFF
  View Pattern List Mode = OFF
Words of length 21:
  compartmentalizations
  counterrevolutionists
  establishmentarianism
  incomprehensibilities
  indistinguishableness
  ultraminiaturizations
The word pattern is: ---------------------

Number of guesses remaining: 26
Guess a letter (# to end game): m
Oops, there are no m's. You used a guess.
Possible words are now:
  counterrevolutionists
  indistinguishableness
The word pattern is: ---------------------


Number of guesses remaining: 25
Guess a letter (# to end game): #
```

```
Terminating game...
```

Sample Gameplay - Partial Game with Word Size 6 and Pattern List Mode

Note that this has mixed input/output, where the characters entered by the user are included after each "**Guess a letter (# to end game):**" prompt.

```
➜ gcc main.c -o program.exe
➜ ./program.exe -p -n 6
Welcome to the (Evil) Word Guessing Game!

Game Settings:
  Word Size = 6
  Number of Guesses = 26
  View Stats Mode = OFF
  View Word List Mode = OFF
  View Letter List Mode = OFF
  View Pattern List Mode = ON
The word pattern is: ------

Number of guesses remaining: 26
Guess a letter (# to end game): u
All patterns for letter u:
  ------        count = 11237   changes = 0
  ----u-        count = 400     changes = 1
  ---u--        count = 464     changes = 1
  --u---        count = 647     changes = 1
  -----u        count = 19      changes = 1
  -u----        count = 1181    changes = 1
  -u-u--        count = 38      changes = 2
  -u--u-        count = 60      changes = 2
  -u---u        count = 2       changes = 2
  --u-u-        count = 4       changes = 2
  --u--u        count = 1       changes = 2
  -uu-uu        count = 1       changes = 4
  --uu--        count = 1       changes = 2
  u-----        count = 284     changes = 1
  u---u-        count = 12      changes = 2
  u--u--        count = 23      changes = 2
  u-u---        count = 8       changes = 2
  ---uu-        count = 1       changes = 2
Oops, there are no u's. You used a guess.
The word pattern is: ------

Number of guesses remaining: 25
Guess a letter (# to end game): i
All patterns for letter i:
  ---i--        count = 1228    changes = 1
  ---ii-        count = 1       changes = 2
  ------        count = 6729    changes = 0
  ----i-        count = 657     changes = 1
  --i---        count = 858     changes = 1
  --i-i-        count = 42      changes = 2
  -----i        count = 58      changes = 1
  -i----        count = 1143    changes = 1
  -i-i--        count = 176     changes = 2
  --i--i        count = 8       changes = 2
  -i--i-        count = 96      changes = 2
  -i-i-i        count = 1       changes = 3
  ----ii        count = 1       changes = 2
  ---i-i        count = 7       changes = 2
  i---i-        count = 15      changes = 2
  i-----        count = 147     changes = 1
  i-i---        count = 15      changes = 2
  i--i--        count = 40      changes = 2
```

```
  i-i-i-        count = 4       changes = 3
  i----i        count = 1       changes = 2
  -i---i        count = 7       changes = 2
  --ii--        count = 3       changes = 2
Oops, there are no i's. You used a guess.
The word pattern is: ------

Number of guesses remaining: 24
Guess a letter (# to end game): o
All patterns for letter o:
  ------        count = 4020    changes = 0
  ----o-        count = 316     changes = 1
  ---o--        count = 393     changes = 1
  ---oo-        count = 32      changes = 2
  --o---        count = 461     changes = 1
  --o-o-        count = 21      changes = 2
  ----oo        count = 6       changes = 2
  -----o        count = 40      changes = 1
  -o----        count = 930     changes = 1
  --o--o        count = 7       changes = 2
  ---o-o        count = 3       changes = 2
  --oo--        count = 72      changes = 2
  -o--o-        count = 90      changes = 2
  -o---o        count = 18      changes = 2
  -oo-oo        count = 6       changes = 4
  -oo---        count = 133     changes = 2
  -o-o--        count = 55      changes = 2
  -o-oo-        count = 1       changes = 3
  -oo-o-        count = 2       changes = 3
  -o--oo        count = 1       changes = 3
  o-----        count = 93      changes = 1
  o--o--        count = 11      changes = 2
  o-o---        count = 3       changes = 2
  o---o-        count = 3       changes = 2
  o----o        count = 2       changes = 2
  oo----        count = 6       changes = 2
  oo-o--        count = 2       changes = 3
  o-o-o-        count = 1       changes = 3
  -o-o-o        count = 1       changes = 3
Oops, there are no o's. You used a guess.
The word pattern is: ------

Number of guesses remaining: 23
Guess a letter (# to end game): e
All patterns for letter e:
  ------        count = 756     changes = 0
  ----e-        count = 1032    changes = 1
  ---e--        count = 290     changes = 1
  --e-e-        count = 72      changes = 2
  -----e        count = 171     changes = 1
  ---e-e        count = 26      changes = 2
  --e---        count = 224     changes = 1
  --ee--        count = 76      changes = 2
  -e---e        count = 94      changes = 2
  -e--e-        count = 440     changes = 2
  ---ee-        count = 37      changes = 2
  --e--e        count = 38      changes = 2
  ----ee        count = 17      changes = 2
  -e----        count = 283     changes = 1
  -e-e--        count = 150     changes = 2
  -ee-ee        count = 5       changes = 4
  -ee---        count = 20      changes = 2
  -ee-e-        count = 68      changes = 3
  -ee--e        count = 5       changes = 3
  -e--ee        count = 10      changes = 3
```

```
  -e-ee-        count = 18      changes = 3
  --ee-e        count = 11      changes = 3
  -e-e-e        count = 13      changes = 3
  e--e--        count = 41      changes = 2
  e---e-        count = 26      changes = 2
  e-----        count = 45      changes = 1
  e----e        count = 19      changes = 2
  e-e---        count = 16      changes = 2
  e--e-e        count = 3       changes = 3
  e-e-e-        count = 5       changes = 3
  e--ee-        count = 5       changes = 3
  e-ee--        count = 1       changes = 3
  e-e--e        count = 2       changes = 3
  e---ee        count = 1       changes = 3
Good guess! The word has at least one e.
The word pattern is: ----e-

Number of guesses remaining: 23
Guess a letter (# to end game): a
All patterns for letter a:
  a-a-e-        count = 16      changes = 2
  a---e-        count = 27      changes = 1
  a---ea        count = 1       changes = 2
  a-a-ea        count = 1       changes = 3
  -a--e-        count = 682     changes = 1
  --a-e-        count = 261     changes = 1
  ----e-        count = 43      changes = 0
  -aa-e-        count = 1       changes = 2
Good guess! The word has at least one a.
The word pattern is: -a--e-

Number of guesses remaining: 23
Guess a letter (# to end game): n
All patterns for letter n:
  -a--e-        count = 521     changes = 0
  -a--en        count = 37      changes = 1
  -an-e-        count = 72      changes = 1
  -anne-        count = 18      changes = 2
  -a-ne-        count = 21      changes = 1
  na--e-        count = 10      changes = 1
  nan-e-        count = 1       changes = 2
  -an-en        count = 2       changes = 2
Oops, there are no n's. You used a guess.
The word pattern is: -a--e-

Number of guesses remaining: 22
Guess a letter (# to end game): #
Terminating game...
```

Sample Gameplay - Full Game (NOT solved) with Word Size 4, Invalid/Repeat Guesses, 7 Guess Limit, and Stats Mode

Note that this has mixed input/output, where the characters entered by the user are included after each "**Guess a letter (# to end game):**" prompt.

```
➜ gcc main.c -o program.exe
➜ ./program.exe -g 7 -n 4 -s
Welcome to the (Evil) Word Guessing Game!

Game Settings:
  Word Size = 4
  Number of Guesses = 7
  View Stats Mode = ON
  View Word List Mode = OFF
  View Letter List Mode = OFF
```

```
  View Pattern List Mode = OFF
The dictionary contains 127142 words total.
The longest word floccinaucinihilipilification has 29 chars.
The dictionary contains 3862 words of length 4.
Max size of the dynamic words array is 4096.
The word pattern is: ----

Number of guesses remaining: 7
Guess a letter (# to end game): $
Invalid letter...
Guess a letter (# to end game): ?
Invalid letter...
Guess a letter (# to end game): 6
Invalid letter...
Guess a letter (# to end game): A
Invalid letter...
Guess a letter (# to end game): a
Oops, there are no a's. You used a guess.
Number of possible words remaining: 2489
The word pattern is: ----

Number of guesses remaining: 6
Guess a letter (# to end game): a
Letter previously guessed...
Guess a letter (# to end game): e
Oops, there are no e's. You used a guess.
Number of possible words remaining: 1484
The word pattern is: ----

Number of guesses remaining: 5
Guess a letter (# to end game): o
Oops, there are no o's. You used a guess.
Number of possible words remaining: 786
The word pattern is: ----

Number of guesses remaining: 4
Guess a letter (# to end game): i
Oops, there are no i's. You used a guess.
Number of possible words remaining: 352
The word pattern is: ----

Number of guesses remaining: 3
Guess a letter (# to end game): u
Good guess! The word has at least one u.
Number of possible words remaining: 248
The word pattern is: -u--

Number of guesses remaining: 3
Guess a letter (# to end game): t
Oops, there are no t's. You used a guess.
Number of possible words remaining: 198
The word pattern is: -u--

Number of guesses remaining: 2
Guess a letter (# to end game): s
Oops, there are no s's. You used a guess.
Number of possible words remaining: 108
The word pattern is: -u--

Number of guesses remaining: 1
Guess a letter (# to end game): r
Oops, there are no r's. You used a guess.
Number of possible words remaining: 75
The word pattern is: -u--
```

```
You ran out of guesses and did not solve the word.
The word is: buck
Game over.
```

Programming Tasks

Treat each of the following tasks as a milestone, where each task should be fully developed and tested for fully functionality before moving on. Remember to reference the starter code for context as you read through the required tasks. .

Task I - [13 points] - Command-Line Arguments

Your program should handle the following command-line arguments in ANY order:

- `[-n wordSize] = sets number of characters in word to be guessed;`
  `if [wordSize] is not a valid input (cannot be`
  `less than 2 or greater than 29), then print:`
  `"Invalid word size."`
  `"Terminating program..."`
  `if dictionary has no words of length wordSize,`
  `then print:`
  `"Dictionary has no words of length [wordSize]."`
  `"Terminating program..."`
  `default wordSize = 5`
- `[-g numGuesses] = sets number of unsuccessful letter guesses;`
  `numGuesses must be a positive integer, otherwise print`
  `"Invalid number of guesses."`
  `"Terminating program..."`
  `default numGuesses = 26`
  `(note: 26 guesses guarantees player can win)`
- `[-s] = stats mode, which includes printing of dictionary statistics AND`
  `number of possible words remaining during gameplay`
  `default is OFF`
- `[-w] = word list mode, which includes a print out of the full list of possible words before every guess`
  `default is OFF`
- `[-l] = letter list mode, which prints out a list of previously guessed letters before each new guess`
  `default is OFF`
- `[-p] = pattern list mode, which prints out a list of all patterns in the word list, together with the`
  `frequency of each pattern and the number of changes in the pattern due to the current guess`
  `default is OFF`
- `[-v] = verbose mode, which turns ON stats mode, word list mode, letter list mode, AND pattern list mode`

Once command-line arguments are handled, print the game settings using the following format:

```
Game Settings:
  Word Size = 6
  Number of Guesses = 26
  View Stats Mode = OFF
  View Word List Mode = OFF
  View Letter List Mode = OFF
  View Pattern List Mode = ON
The word pattern is: ------
```

Refer to the sample gameplay outputs above OR the demo executable to see how each command-line argument game setting affects the game play and for other exact print statement formatting.

Task II - [10 points] - addWord( )

Write the addWord( ) function, which adds a new word (i.e. **newWord**) to a dynamic heap-allocated array of C-strings. The function prototype is as follows:

```
void addWord(char*** words, int* numWords, int* maxWords, char* newWord)
```

The first parameter **words** is a passed-by-pointer(*) array(*) of C-strings(*); thus, it is a *triple pointer*! That is, the address location to the first word is what is passed to this function; because the array may be reallocated inside the function, it is passed-by-pointer.

Make no assumption about the scope of **newWord**. That is, you must explicitly allocate heap-space for the characters of the word to be added, and copy the characters of **newWord** to this newly allocated space inside this function. This is tested by the autograder.

In this function, the reallocation of the entire ***words** array should occur when the number of words it stores (***numWords**) would exceed the maximum capacity of the array (***maxWords**), which is how much space has been allocated for it. If adding an additional word would make ***numWords** greater than ***maxWords**, then the capacity of the array should be doubled; that is, allocate space for a new word list array with double the capacity, copy all important data over to the new array, free up the old space, rewire the pointer, and make the required updates to the size parameters.

Note that **numWords** and **maxWords** are passed-by-pointer because they may change inside the function. Also note that **newWord** is a char pointer parameter, simply because it is a char array (i.e. a C-string). The starter code comments include some helpful tips, reproduced here for convenience:

```
note: *words is an array of pointers
      **words is an array of chars
      ***words is a char
      (*words)[0] is the 1st C-string in the array
      (*words)[1] is the 2nd C-string in the array
      (*words)[0][0] is 1st char of 1st C-string
      (*words)[1][2] is 3rd char of 2nd C-string
      etc.
```

Tasks III & IV - [15 points] - File-Read dictionary.txt to Build Initial Word List

The list of possible words must be a dynamic array of C-strings, which must be declared as an array of pointers (e.g. **char** wordList**). This dynamic array of pointers must dynamically grow in the following way:

- The initial allocation should allow a maximum capacity of 4 C-strings (i.e. 4 pointers).
- File-read words one-by-one from *dictionary.txt*, and store only words of the correct length in the word list array. Note that the word size is a command-line argument with default value of 5.
- Add words of the correct size to the word list array using a calls to **addWord( )**, which will dynamically grow the word list array by doubling its capacity when more space is necessary.

In the end, each element of the word list array should point to a heap-allocated C-string that stores a word, where all words have the same length set by the -n command-line argument (or 5 by default), and all characters are lower-case letters.

As you file-read *dictionary.txt*, keep track of the total number of words, the longest word, the number of words copied over to the word list array (i.e. number of words of the correct length), and the maximum capacity of the word list array (must be a power of 2, since it starts with max size 4 and doubles when needed). If the stats mode is ON, then print out these statistics using the format shown in the sample output above.

Tasks V & VI - [20 points] - strNumMods( ) & strDiffInd( )

These tasks require writing two functions that are of similar nature to those found in the string.h library, but customized for our needs in this program. Both functions analyze two C-strings, which are the only arguments, and return an integer. Follow the model of the string.h library functions when developing these functions; use pointer incrementation to work through the array(s), character-by-character, always keeping an eye out for the null character, which signifies the end of the string.

The first, **strNumMods( )** is similar to **strcmp( )**, in that both analyze the differences between the characters, index-by-index, of the two strings. Whereas **strcmp( )** finds the first character difference and returns a measure of the difference, **strNumMods( )** should count and return the total number of character differences. If one string is longer than the other, the extra characters should add to the total count of character differences. The starter code comments provide some examples, which are reproduced here for convenience.

```
int strNumMods(char* str1, char* str2)
  //-------------------------------------------------------------------
  // TODO - Task V: write the strNumMods() function, which
  //      returns count of character differences between two strings;
  //      includes extra characters in longer string if different lengths
  // Exs: str1 = magic, str2 = magic, returns 0
  //      str1 = wands, str2 = wants, returns 1
  //      str1 = magic, str2 = wands, returns 4
  //      str1 = magic, str2 = mag, returns 2
  //      str1 = magic, str2 = magicwand, returns 4
  //      str1 = magic, str2 = darkmagic, returns 8
  //-------------------------------------------------------------------
```

The second, **strDiffInd( )** is also similar to **strcmp( )**, in that both seek the first character difference between the two strings. Whereas **strcmp( )** returns a measure of the difference, **strDiffInd( )** should return the index where the first

difference occurs. If both strings are identical, then it should return strlen(str1) = strlen(str2). The starter code comments provide some examples, which are reproduced here for convenience.

```
int strDiffInd(char* str1, char* str2)
  //---------------------------------------------------------------
  // TODO - Task VI: write the strDiffInd() function, which
  //      returns index of character where two strings first differ, &
  //      returns strlen(str1) = strlen(str2) if no differences
  // Exs: str1 = magic, str2 = magic, returns 5
  //      str1 = wands, str2 = wants, returns 3
  //      str1 = magic, str2 = wands, returns 0
  //      str1 = magic, str2 = mag, returns 3
  //      str1 = magic, str2 = magicwand, returns 5
  //      str1 = magic, str2 = darkmagic, returns 0
  //---------------------------------------------------------------
```

Tasks VII - [30 points] - Play the Game

This is the most open-ended task of the program and encompasses the bulk of the game play. At this point, a thorough review of the sample outputs above and playing the game using demo.exe is a good idea in order to understand all of the nuances of the game. In summary, the game should proceed as follows:

- Input a character from the the user as their guess. If the input character is not a lower-case letter OR the input lower-case letter has already been guessed, then prompt the user for another input until a valid guess has been input. One exception is that if the user enters '#', then terminate the game immediately.
- Build a dynamic array of unique **Pattern** structs, where a **Pattern** struct is defined in the starter code and contains a heap-allocated C-string representing the current status of the final word, with a '**–**' for each unsettled character; for each word in the current word list, produce an updated pattern by replacing '**–**'s with the current guessed letter wherever that letter appears in the word. Then, append the pattern to the dynamic array of **Pattern**s as a new element if it is a unique pattern (i.e. does not match any of the patterns already in the array). Otherwise, locate the **Pattern** already in the array that it matches and increment the count subitem for that **Pattern**. The array of **Pattern**s should dynamically grow, beginning with a capacity of 4 **Pattern**s and doubling whenever more space is needed. As an example, suppose in the midst of a game play, the current word list and final word pattern are as follows:

  ```
  Possible words are now:
    bowed
    boxed
    foxed
    jowed
    joyed
    moved
    mowed
    vowed
    wowed
    yowed
  The word pattern is: –o–ed
  ```

  Then, the user guesses letter '**w**'. The resulting **Pattern** array contains 3 elements, because there are only 3 unique patterns in the word list made by adding the instances of '**w**' to the current final word pattern **–o–ed**:

  ```
  All patterns for letter w:
    –owed         count = 5       changes = 1
    –o–ed         count = 4       changes = 0
    wowed         count = 1       changes = 2
  ```

  Note that the patterns are printed in the order they first appear in the word list. Also, the patterns are printed with their frequencies in the word list (i.e. count) and how many changes there are to the final word pattern for that pattern (i.e. number of occurrences of the current letter guess, '**w**' here). In this case, the pattern that will be chosen is **–owed**, since it has the highest count.

- Find the most common pattern in the word list, but analyzing the dynamic **Pattern** struct array. Ties between pattern family sizes are broken by first checking which pattern has the least changes due to the guessed letter, i.e. the pattern with the least number of occurrences of the guessed letter is chosen. If multiple patterns are still tied, the pattern where the guessed letter appears earliest is chosen; if the first occurrence of the guessed letter is at the same index, then the second occurrence is checked; etc., until a single pattern is chosen. For example, consider the following set of patterns & counts:

```
All patterns for letter e:
  -----s         count = 2       changes = 0
  --e-es         count = 3       changes = 2
  --ee-s         count = 3       changes = 2
  -e---s         count = 1       changes = 1
  -e-e-s         count = 3       changes = 2
  -e--es         count = 3       changes = 2
  -e-ees         count = 3       changes = 3
  e----s         count = 2       changes = 1
```

To select the pattern, first find the highest count(s). Here, there is a 5-way tie:

```
  --e-es         count = 3       changes = 2
  --ee-s         count = 3       changes = 2
  -e-e-s         count = 3       changes = 2
  -e--es         count = 3       changes = 2
  -e-ees         count = 3       changes = 3
```

The first tiebreaker is to go to the smallest number of changes (i.e. instances of the guessed letter). But, we still have a 4-way tie to break:

```
  --e-es         count = 3       changes = 2
  --ee-s         count = 3       changes = 2
  -e-e-s         count = 3       changes = 2
  -e--es         count = 3       changes = 2
```

Next, we look for the earliest instance of the guessed letter, but we still have a 2-way tie:

```
  -e-e-s         count = 3       changes = 2
  -e--es         count = 3       changes = 2
```

So, we continue comparing instances of the guessed letter, until we find one pattern with an earlier instance, leaving us with the final chosen pattern as: `-e-e-s`.

- Once the most common pattern is found (with tiebreakers), update and print out the final word pattern after notifying the player whether their guessed letter is in the final word or not. If their letter is not in the final word pattern, then there guess was wrong and they just used up one of their valuable wrong guesses. At this point, end the game if the player has used up all of the allowed wrong guesses or has solved the final word by revealing all letters. Otherwise, repeat the process by inputting another letter guess from the user.
- Throughout the game play, produce print statements matching the format of the sample gameplay outputs and demo.exe; only print the statistics [-s], word lists [-w], letter list [-l], and patter list [-p] if the proper command-line argument flags are turned ON.

Tasks VIII - [12 points] - Free all Heap-Allocated Memory

Avoid potential memory leaks by freeing all heap-allocated memory leaks. Since the word size for each game play is set by a command-line argument, any array whose size depends on the word size should be dynamically heap-allocated, and, thus, must be tracked carefully and freed whenever the array goes out of scope and/or before the program is terminated. This is checked by running Valgrind in the autograder.

Requirements

- The word list must be generated by reading the file dictionary.txt. The word list must be a heap-allocated array of C-strings, which is an heap-array of pointers to heap-allocated char arrays. The word list must be a true dynamic array, where the maximum capacity begins with size 4 and doubles exactly when more space is needed. Whereas there is flexibility in how the reallocation is done, you are strongly encouraged to manually grow the array using malloc() only. The word list must only contain the words that are possible final words for the game. That is, the word list begins with all words of the specified length found in dictionary.txt, and then continually reduces in size as the gameplay progresses. The word list should be reallocated to a smaller size whenever the set of possible words has been reduced. No copies of the word list can be made, except during reallocation steps. Violations of this requirement will receive a manually graded deduction.
- A dynamic array of Pattern structs must be generated and used in deciding which pattern to move forward with in the gameplay. Whereas there is flexibility in how the reallocation is done, you are strongly encouraged to manually grow the array using malloc() only. The descriptions above contain full tie-breaking rules amongst the various

patterns to ensure each gameplay is deterministic upon the command-line game settings and the order of the input character guesses. Violations of this requirement will receive a manually graded deduction.
- Use the starter code as provided, adding code to complete functions and developing additional functions to complete the tasks, without making any structural changes: do NOT modify the Pattern struct definition (no name change, do not add subitems, do not remove subitems, do not modify subitem definitions, etc.), and do NOT change the function headers (no name changes, do not add parameters, do not remove parameters, do not modify parameter types, etc.). Violations of this requirement will receive a manually graded deduction.
- Solve each task and the program at large as intended. Additional arrays (static or dynamic) are allowed, as long as they don't represent the same data that is (or should be) stored in the word list. Ex: you need to keep track of previous guesses, which requires additional array(s); you will likely also need to build/use additional array(s) and/or Patterns to select a most-common pattern. However, one example of a violation is storing a full list of ALL words from the file dictionary.txt to refer back to throughout the gameplay. That is a waste a storage space and computing resources. Violations of this requirement will receive a manually graded deduction.
- Use the various methods of inputting variables and data to the program appropriately: command-line arguments set game settings and reporting modes, file-reading builds the initial word list, and interactive input (or redirection from file) used for game play letter guesses.
- All dynamic heap-allocated memory must be freed to prevent possible memory leaks. This issue is checked by the autograder but may also receive a manually graded deduction.
- Coding style issues are manually graded using deductions, worth up to 25% of the total project score. Style points are graded for following the course standards and best practices laid out in the syllabus, including a header comments, meaningful identifier names, effective comments, code layout, functional decomposition, and appropriate data and control structures.
- Programming projects must be completed and submitted individually. Sharing of code between students in any fashion is not allowed. Use of any support outside of course-approved resources is not allowed, and is considered academic misconduct. Examples of what is allowed: referencing the zyBook, getting support from a TA, general discussions about concepts on piazza, asking questions during lecture, etc. Examples of what is NOT allowed: asking a friend or family member for help, using a "tutor" or posting/checking "tutoring" websites (e.g. Chegg), copy/pasting portions of the project description to an AI chatbot, etc. Check the syllabus for Academic Integrity policies. Violations of this requirement will receive a manually graded deduction, and may be reported to the Dean of Students office.

Submission

Develop your program in the IDE below. Use the "Run" button to test your code interactively as you develop your program. Use the "Submit for grading" button to test your code against the suite of autograded test cases, which also submits your program for grading. Formally, the official code submission is your latest submission that scores the highest when run through the autograder. Late submissions are allowed for a point reduction. If you submit your highest scoring program after the deadline it will be treated as a late submission, which will incur the point reduction.

Optional Extension Mode (Extra Credit)

The algorithm outlined in this description is by no means optimal, and there are several cases in which it will make bad decisions. For example, suppose that the human has exactly one guess remaining and that the computer has the following word list:

```
DEAL    TEAR    MONK
```

If the human guesses the letter 'E' here, the computer will notice that the word family `-E--` has two elements and the word family `----` has just one. Consequently, it will pick the family containing DEAL and TEAR, revealing an E and giving the human another chance to guess. However, since the human has only one guess left, a much better decision would be to pick the family `----` containing MONK, causing the human to lose the game.

There are several other places in which the algorithm does not function ideally. For example, suppose that after the player guesses a letter, you find that there are two word families, the family `-E--` containing 10,000 words and the family `----` containing 9,000 words. Which family should the computer pick? If the computer picks the first family, it will end up with more words, but because it revealed a letter the user will have more chances to guess the words that are left. On the other hand, if the computer picks the family `----`, the computer will have fewer words left but the human will have fewer guesses as well. More generally, picking the largest word family is not necessarily the best way to cause the human to lose. Often, picking a smaller family will be better.

After you have completed all required tasks and have built a fully functioning and fully devious program, take some time to think over possible improvements to the algorithm. You might weight the word families using some metric other than size. You might consider having the computer “look ahead” a step or two by considering what actions it might take in the