Students:
Section 14.3 is a part of 1 assignment:
Programming Project 03 - Bit Big Bug Tug - Word Ladder Builder

This assignment's due date has passed. Activity will still be recorded, but will not count towards this assignment (unless the due date is changed). See this article for more info.

Includes: 🟩 zyLab

Due: 10/10/2024, 11:59 PM CDT

Instructor created

14.3 P03 - Bit Big Bug Tug - Word Ladder Builder

Introduction to Word Ladders

A **word ladder** is a bridge between one word and another, formed by changing one letter at a time, with the constraint that at each step the sequence of letters still forms a valid word. For example, here is a word ladder starting from the word **"data"** and *climbing up* the ladder to the word **"code"**. For illustration, each changed letter is bolded and has a '^' mark underneath it:

```
|  code  |
|     ^  |
|  cove  |
|    ^   |
|  cave  |
|  ^     |
|  gave  |
|     ^  |
|  gate  |
|  ^     |
|  date  |
|      ^ |
|  data  |
```

The above word ladder has height 7, since there are 7 words in the ladder. There are many other word ladders that connect these two words. For example, here is another ladder of height 5 that uses some more obscure words:

```
|  code  |
|    ^    |
|  cade   |
|     ^   |
|  cate   |
|  ^      |
|  date   |
|      ^  |
|  data   |
```

Note that there are many other ladders that connect these two words. Word ladders can be very short, e.g. **cove** -> **code** trivially has the minimum possible height of 2. There are also pairs of words for which a word ladder cannot be formed, e.g. there is no word ladder to connect **stack** -> **queue**.

In this project, you are tasked with writing large components of a program that allows the user to generate valid word ladders using a real English dictionary. The project involves managing dynamic memory, which will require diagramming, careful attention to detail, and efficient debugging strategies. As you work on this project, it may feel like a **bit big bug tug** at times, but the delight from achieving the programming tasks below will have you singing "**ding dong done!**"

The Program

Dictionary Files & Demo Executable

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 the purposes of making word ladders for small to moderate sized words. Smaller dictionaries are also provided for testing purposes: *simple3.txt* contains a limited number of 3-letter words, *simple4.txt* contains a limited number of 4-letter words, and *simple5.txt* contains a limited number of 5-letter words. Also, *sampleDict.txt* contains a small number of words with varying word lengths, but you can still make an interesting word ladder connecting **toe** -> **ear**.

At this point, you should begin to test your program for its full output, which comprises the last 25 test cases in the autograder suite. Once again, before jumping straight to submitting to the autograder, you are encouraged to compare your program output to a *fully-functioning* output by running the provided **demo.exe** in the starter file tree.

Before diving into the programming details below, you can run a fully-functioning version of the word ladder game using **demo.exe.** 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
```

Additionally, **demo.exe** can be run with or without command-line arguments (which are not introduced until the very last task). Here one valid way to run **demo.exe** with ALL possible command-line arguments:

```
➜ chmod a+x demo.exe
➜ ./demo.exe -d dictionary.txt -n 4 -m 15 -s byte -f chew
```

Primary Application and Starter Code

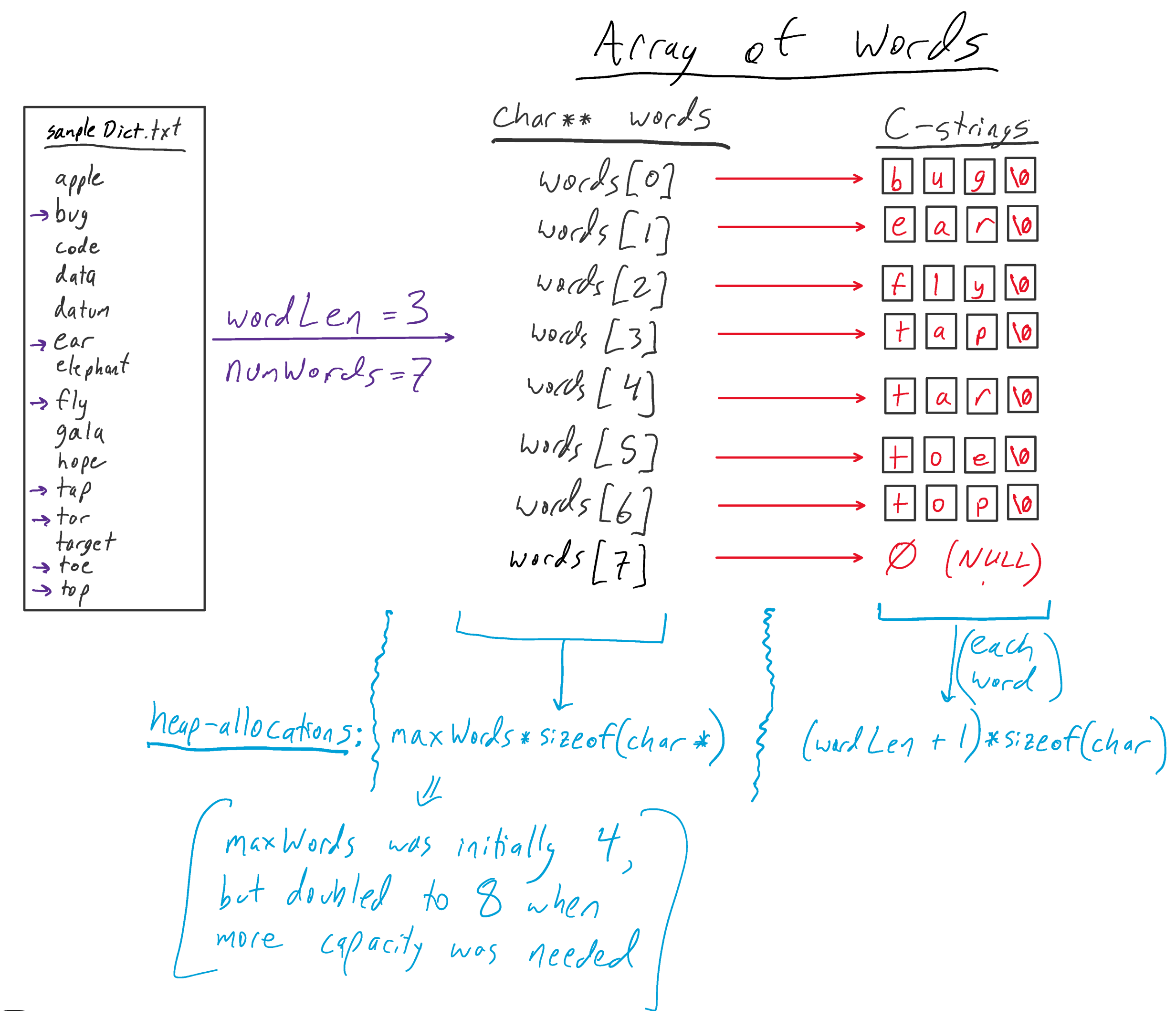
Most of the primary application is provided in **main()** of the starter code; however, a few parts of **main()** are incomplete and the completed portions contain many calls to functions that still need to be written. Your programming tasks are all labeled in the starter code with a *TODO* - *Task ##* statement.

Here are the main steps of the program:

- Program settings - the following are set by interactive user-input in the starter code, but may be preset by command-line arguments in the full functioning version of the program:
  - The user interactively sets the word length for the starting and final words (and, thus, all the words) for the word ladder. This, in turn, also sets the **wordLen** for the full set of words to be read in from the dictionary file.
  - The user interactively sets the maximum allowed word ladder height, **maxLadder**. This, in turn, sets the maximum capacity for the dynamic array of C-strings that will represent the word ladder.
  - The user interactively sets the dictionary file name, **dict**, to be used for reading words into the full array of possible words that could later make up the ladder.
- The dictionary file is opened and each word in the dictionary is scanned. Words that have the correct length (i.e. **wordLen**) are added to a dynamic **words** array, which is an array of C-strings. That is, it is a heap-allocated array of pointers, where each element points to a unique heap-allocated character array. The **words** array should start with a maximum capacity of 4 C-string pointers and double in capacity whenever more space is needed. A memory diagram for a sample dynamic **words** array, using the *sampleDict.txt* dictionary file and a word length of 3, is provided below. Notice that the

capacity variable **maxWords** was doubled to 8 as the dictionary file was scanned, since it contains more than four 3-letter words. However, it did NOT double again to a capacity of 16, since the dictionary only has seven 3-letter words. Thus, the **words** array has a capacity to store 8 C-strings, but it only stores 7 C-strings; i.e. **words[0]** through **words[6]** are pointers that point to distinct heap-allocated space where 3-letter words are stored (with one extra character for the null character **'\0'** to signify the end of each word), while **words[7]** points to **NULL**.

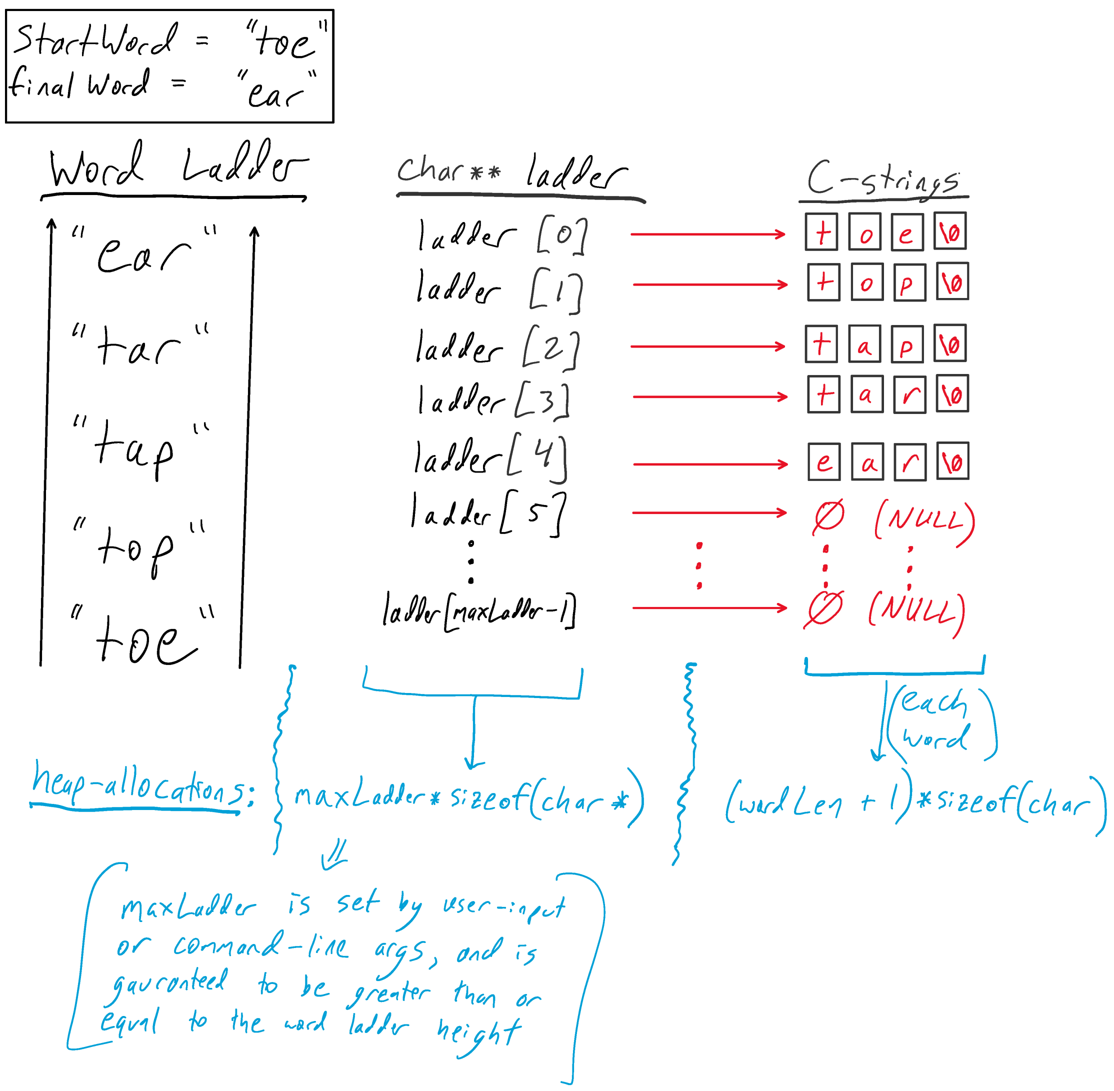
Figure 14.3.1: Memory diagram for a sample dynamic words array



- Once the **words** array has been built, the user interactively inputs both the starting word (**startWord**) and the final word (**finalWord**), unless these are validly set by command-line arguments. If either entered word has an incorrect size (i.e. not equal to **wordSize**) or the word is not found in the **words** array, then the user is requested to enter another word.

- Now, it is time to build a word ladder! The **ladder** is a dynamic array of C-strings, exactly like the **words** array. However, there is one key difference, namely that the maximum ladder height is already set, so we can reserve enough space for **maxLadder** pointers in the initial allocation, and there is no capacity doubling for the **ladder** array like there is for the **words** array. A memory diagram for a sample dynamic word **ladder** array, that connects "*toe*" to "*ear*" using the **words** array built above from the *sampleDict.txt* dictionary file, is provided below. Notice that there may be many **NULL** elements at the back of the **ladder** array, even once a word ladder has been completed. The program builds word ladders as follows:
    - The starting word is added to the empty **ladder** (i.e. the bottom rung of the ladder) array.
    - The user then enters a word, which is checked for validity; specifically, the program must check that the word is found in the dictionary, has the correct length (automatic if it is found in the **words** array), AND is a single-character change from the previous rung of the word ladder. If the word is invalid for any reason, the user is allowed to enter another word. Once a valid word is entered, it is added to the back of the **ladder** array (i.e. the highest rung of the ladder to that point).
    - The process of the user entering words to be added to the **ladder** array continues until either the ladder reaches the maximum allowable height, **maxLadder**, OR the **finalWord** has been added to the ladder, representing a completed word ladder.

Figure 14.3.2: Memory diagram for a sample dynamic word ladder array



Word Ladder Displays

Note: whitespacing is checked for word ladder displays, since proper whitespacing is crucial for a meaningful output here.

Incomplete word ladders are displayed to the user before each rung is filled and at the end of the program if the final word is NOT reached. Word ladders are always displayed with the starting word at the bottom rung, and each successive word shown above the previous word, one rung higher on the ladder. To signify the ladder as incomplete, three lines each with an ellipsis (i.e. three periods) are shown at the top of the ladder. A sample incomplete word ladder for start word "data" and final word "code" is shown below:

```
  ...
  ...
  ...
  gave
  gate
  date
  data
```

Note that there are two whitespaces at the beginning of each line in the word ladder display, no whitespaces after the three periods, and no whitespaces after each word.

Complete word ladders are displayed at the end of the program if the final word is reached. In between each rung of a completed word ladder, the symbol '^' signifies the character that changes between the two rungs of the ladder. A sample complete word ladder for start word "data" and final word "code" is shown below:

```
  code
    ^ 
  cove
   ^  
  cave
  ^   
  gave
    ^ 
  gate
  ^   
  date
     ^
  data
```

Note that the lines with the '^' symbol have the same number of total characters as the lines with the words. That is, the newline character occurs at the same location on each line, which is immediately following the characters of the words. In the example above, this means there is one whitespace after the topmost '^' between *cove* and *code*, there are two whitespaces after the '^' between *cave* and *cove*, ..., and there are no whitespaces after the '^' between *data* and *date*.

All word ladders are displayed with an indentation of two whitespaces to the left of every word. All word ladders are displayed with the starting word at the bottom (i.e. printed to console last) and each successive ladder rung climbing up to the last word entered into the word ladder (i.e. printed to console first).

Programming Tasks

Treat each of the following tasks as a milestone, where each task should be fully developed and tested for full functionality before moving on. Each task has a clearly labeled TODO statement in the starter code, with a full description following. Thus, only a brief overview of each task is provided in the descriptions to follow.

Tasks I & II - [20 points] - strCmpCnt( ) & strCmpInd( )

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, **strCmpCnt( )** 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, **strCmpCnt( )** 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 strCmpCnt(char* word1, char* word2)
   //------------------------------------------------------------------
   // TODO - Task I: write the strCmpCnt() function, which returns the
   //                count of character differences between two words
   //                (i.e. strings); include the extra characters in the
   //  longer word if the strings have different lengths
   // Exs: word1 = magic, word2 = magic, returns 0
   //      word1 = wands, word2 = wants, returns 1
   //      word1 = magic, word2 = wands, returns 4
   //      word1 = magic, word2 = mag, returns 2
   //      word1 = magic, word2 = magicwand, returns 4
   //      word1 = magic, word2 = darkmagic, returns 8
   //------------------------------------------------------------------
```

The second, **strCmpInd( )** 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, **strCmpInd( )** should return the index where the first difference occurs. If both strings are identical, then return -1. The starter code comments provide some examples, which are reproduced here for convenience.

```
int strCmpInd(char* word1, char* word2)
   //-------------------------------------------------------------------
   // TODO - Task II: write the strCmpInd() function, which returns the
   //                 index of the character where two strings first
   //                 differ, & returns -1 if there are no differences
   // Exs: word1 = magic, word2 = magic, returns -1
   //      word1 = wands, word2 = wants, returns 3
   //      word1 = magic, word2 = wands, returns 0
   //      word1 = magic, word2 = mag, returns 3
   //      word1 = magic, word2 = magicwand, returns 5
   //      word1 = magic, word2 = darkmagic, returns 0
   //-------------------------------------------------------------------
```

Task III - [10 points] - appendWord( )

Write the **appendWord( )** 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 appendWord(char*** pWords, int* pNumWords, int* pMaxWords, char* newWord)
```

The first parameter **pWords** is a passed-by-pointer(*) array(*) of C-strings(*); thus, it is a *triple pointer*! That is, the address location to the first word's C-string array pointer 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 ***pWords** array should occur when the number of words it stores (***pNumWords**) would exceed the maximum capacity of the array (***pMaxWords**), which is how much space has been allocated for it. If adding an additional word would make ***pNumWords** greater than ***pMaxWords**, 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 **pNumWords** and **pMaxWords** 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: *pWords is an array of pointers
            **pWords is an array of chars
            ***pWords is a char
            (*pWords)[0] is the 1st C-string in the array
```

```
(*pWords)[1] is the 2nd C-string in the array
(*pWords)[0][0] is 1st char of 1st C-string
(*pWords)[1][2] is 3rd char of 2nd C-string
etc.
```

Task IV - [8 points] - file-read the dictionary file to build the full list of possible words

Navigate to **main()** to find the *TODO statement* for this task. 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 the dictionary file **dict**, and store only words of the correct length, **wordLen**, in the word list array.
- Add words of the correct size to the word list array using a calls to **appendWord( )**, 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, **wordLen**, and all characters are lower-case letters.

As you file-read, 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), should automatically update inside the **appendWord()** function. The final statistics are then printed in **main()**.

Task V - [10 points] - linSearchForWord()

Write the **linSearchForWord()** function, which should use a *linear search* of the **words** C-string array to determine if and where a queried word lives in the array. Note that a linear search simply refers to the process of checking an array element-by-element, starting from the front and moving to the back. The function should return the index in the array where the queried word is found OR return -99 if the queried word is not found in the array.

Task VI - [10 points] - checkForValidWord()

Write the **checkForValidWord()** function, which checks if a queried word is a valid option for the next word in an incomplete word ladder. This is a Boolean function that should return FALSE if the word is invalid or return TRUE if the word is valid. There are three ways in which a word can be invalid:

1. the queried word is invalid if it does not have the correct character count,
2. the queried word is invalid if it is not in the dictionary, and
3. the queried word is invalid if it is not a single-character difference from the previous word in the ladder.

Note that many words may break more than one of these *invalidity* measures; so, the order of the above list is the order in which the invalid checks must be done. For example, if the queried word is both too long (1) and not a single-character difference from the top of the current ladder (3), then it would break on invalidity measure (1), since the length would be checked first.

Also note that one result of the validity check for *invalidity* measure (3) is that the queried word is invalid if it is an exact match to the previous word in the ladder, which is consistent with valid word ladders.

If a word is not deemed invalid based on the above list, then the word is valid.

There is one more special case that this function must handle: the user can enter *DONE* at any point to stop the process of building the word ladder. The primary application is written such that this function should return TRUE if the queried word is *DONE*. This must take precedence over all other *validity*/*invalidity* checks.

Task VII - [4 points] - isLadderComplete()

Write the **isLadderComplete()** function, which simply checks if a ladder is *complete*. A ladder is complete if the word on the top rung is the desired final word. The function should return TRUE if the ladder is complete and return FALSE if the ladder is incomplete.

Tasks VIII & IX- [16 points] - building incomplete ladders and displayIncompleteLadder()

Much of the scaffolding for building the word ladders is provided in **main()**. However, you are tasked with writing one important while loop condition (**Task VIII**). Specifically, you must write the condition such that the word ladder building process continues

ONLY if ALL of the following conditions are met:

1. the ladder has available rung(s) for more words to be added, before reaching the maximum allowed height,
2. the user is NOT attempting to stop the word ladder building process by entering *DONE*, and
3. the ladder is still incomplete.

Once the while loop condition is implemented correctly, the program should be able to generate valid word ladders. However, you must also write the function **displayIncompleteLadder()** in order to view the ladder correctly (**Task IX**). The proper format for displaying an incomplete ladder is described in the **Word Ladder Displays** section above and in the starter code *TODO statement* comments. Note that the autograder checks for exact output matches, including whitespaces. Thus, it is important that you follow all instructions laid out in the ladder display formatting descriptions.

Task X - [10 points] - displayCompleteLadder()

Once you have incomplete word ladders being generated and displayed accurately, write the **displayCompleteLadder()** function in order to view completed word ladders built by your program. Once again, the proper format for displaying a complete ladder is described in the **Word Ladder Displays** section above and in the starter code *TODO statement* comments. Note that the autograder checks for exact output matches, including whitespaces. Thus, it is important that you follow all instructions laid out in the ladder display formatting descriptions.

Task XI - [6 points] - free all heap-allocated memory & avoid array out-of-bounds issues

Avoid potential memory leaks by freeing all heap-allocated memory and avoiding array out-of-bounds issues. Since the word length for each word ladder is set by a user-input (or a command-line argument), any array whose size depends on the word length 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. You should check this using valgrind as you develop code and before submitting to the autograder. It is also checked by running valgrind in the autograder.

Task XII - [6 points] - handle command-line arguments

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

```
◦ [-n wordLen] = sets word length for word ladder;
                 if [wordLen] is not a valid input
                 (cannot be less than 2 or greater than 20),
                 or missing from command-line arguments,
                 then let user set it using interactive user input
◦ [-m maxLadder] = sets maximum word ladder height;
                   [maxLadder] must be at least 2;
                   if [maxLadder] is invalid or missing from
                   command-line arguments, then let user set it
                   using interactive user input
◦ [-d dictFile] = sets dictionary file;
                  if [dictFile] is invalid (file not found) or
                  missing from command-line arguments, then let
                  user set it using interactive user input
◦ [-s startWord] = sets the starting word;
                   if [startWord] is invalid
                   (not in dictionary or incorrect length) or
                   missing from command-line arguments, then let
                   user set it using interactive user input
◦ [-f finalWord] = sets the final word;
                   if [finalWord] is invalid
                   (not in dictionary or incorrect length) or
                   missing from command-line arguments, then let
                   user set it using interactive user input
```

Note that all valid arguments are of the form [***-c value***], where ***c*** is a single letter to flag a specific argument type to set the associated program variable to ***value***. Some example follow:

- ***./a.out -n 4*** : the presence of ***-n 4*** on the command-line during program execution means that **wordLen** should be set to the value 4 (and, thus, user-input is not needed to set **wordLen**)
- ***./a.out -n 4 -d dictionary.txt*** : in addition to **wordLen** being set to 4, the presences of ***-d dictionary.txt*** on the command-line during program execution means that **dict** should be set to "**dictionary.txt**" (and, thus, user-input is not needed to set **dict**). Note that **maxLadder**, **startWord**, and **finalWord** will all still be set by user-input in this case.
- ***./a.out -m 20 -n 4 -s word -f rung -d dictionary.txt*** : here, all program variables are set by command-line arguments as follows (and user-input is not needed to set them):
  - **maxLadder** = 20

  - **wordLen** = 4
  - **startWord** = "word"
  - **finalWord** = "rung"
  - **dict** = "dictionary.txt"

Every command-line argument is optional (as they can all be set using interactive user-input), and the arguments that are present on the command-line during execution can be placed in any order. Ignore any invalid command-line arguments. For example, ***./a.out -n 4 -x -t -v 25 -d dictionary.txt*** should result in **wordLen** being set to 4, **dict** being set to dictionary.txt, and the remaining faulty command-line arguments (***-x -t -v 25***) being ignored.

Requirements

- The word list must be generated by reading the dictionary file. 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 words for the ladder, i.e. have the correct character length. No copies of the word list can be made. Violations of this requirement will receive a manually graded deduction.
- Similarly, the word ladder must a heap-allocated array of C-strings, which is an heap-array of pointers to heap-allocated char arrays. The ladder does not need to change size, since the maximum ladder height is a program parameter set by user-input or command-line arguments, and should not change as the program runs. 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 initial allocations for the words and ladder arrays; 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 or ladder. One example of a violation is storing a full list of ALL words from the dictionary file to refer back to throughout the word ladder building process. That is a waste a storage space and computing resources. Violations of this requirement will receive a manually graded deduction. Required functions from earlier tasks must be called whenever relevant for later tasks. For example,
  - **Task I - strCmpCnt()** and **Task V - linSearchForWord()** should be called to achieve the proper functionality for **Task VI - checkForValidWord()**,
  - **Task II - strCmpInd()** should be called to achieve the proper functionality for **Task X - displayCompleteLadder()**,
  - **Task III - appendWord()** should be called to achieve the proper functionality for **Task VI - building the word list in main()**,
  - **Task VII - isLadderComplete()** should be called to achieve the proper functionality for **Task VIII - building ladders in main()**.
- Use the various methods of inputting variables and data to the program appropriately: command-line arguments for word ladder settings, file-reading builds the initial word list, and interactive input (or redirection from file) used for (re)setting word ladder settings and the word ladder building process. Violations of this requirement will receive a manually graded deduction.
- 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.

Tips for Success on this Programming Project

- **Main programming concepts: C-strings, file reading, and dynamic array memory management -** the program centers on an the creation of a word ladder that connects two English words using a dictionary of the full English language. The trickiest aspects of this project involve file reading from the full dictionary to build a list of words that have a desired character length using a dynamically growing heap-allocated array of C-strings. Much of the primary application is provided in the starter code, within **main()**. Thus, it is important that you read through the starter code and obtain a full understanding of all steps of the primary application as you read through the project description below.
- **NOT all tasks/functions are created equal -** the *Programming Tasks* for this project are primarily to develop many functions to support the primary application, most of which is provided for you in the starter code. The level of scaffolding is designed to lead you through the program development that leaves room for creatively applying course concepts and tools. A natural consequence is that..
    - some Tasks are straightforward and some Tasks are challenging;
    - some Tasks ask you to do one thing specifically and some Tasks are purposefully open-end with multiple components;
    - some Tasks test your ability to follow instructions verbatim and some Tasks require you to practice problem-solving and critical-thinking skills;
    - some Tasks may only take you a few minutes and some Tasks may require an initial attempt followed by a break to do something else only to come back multiple times to tackle the challenge;
- **Continual Testing & Debugging -** you should continue building the best practice habit of continual testing functionality as you develop code and utilizing the debugging tools introduced in class, such as valgrind, gdb, and/or GUI debuggers. There is no formal required structure to the testing/debugging you do, but continual testing in small chunks of code is essential for efficient code development. Testing should begin by compiling and running the program in the terminal with a set of simple interactive user inputs OR using input redirection. Effective testing strategies also include comparing your output to the fully-functioning output provide with **demo.exe**. You are strongly recommended to only submit to the autograder once you are convinced that your code for a given task is functioning correctly based on your own running and testing of the program. Whenever testing (or submitting to the autograder), slow down, dedicate ample time to process program output and/or error messages, take a meaningful look at recent code changes, and develop/implement productive debugging strategies.
- **No superfluous print statements -** except for explicitly-defined displays (e.g. displaying the word ladders, print statements for valid/invalid words, etc.), the functions you write should have no print statements. Most of the console output is handled in the primary application. Of course, as you develop, test, and debug your code, you may introduce

some print statements to achieve those purposes. These print statements must be removed and/or commented out before you submit to the autograder (especially if it involves a large amount of printed output, such as the entire words array). In the event that you experience issues with this page crashing or not loading due to extraneous output, we cannot provide support and you will need to contact the zyBooks team for assistance.

- **Gradescope submission -** at the bottom of this description there are instructions for submitting your work (code file main.c) to Gradescope. Your project grade will be determined using the zyLab autograder score AND your Gradescope submission. You must do BOTH to earn credit for you work. Your project submission timestamp is formally determined from the Gradescope submission time.

Submission & Grading

Develop your program in the IDE below. Do NOT use the "Run" button; instead, compile your code (e.g. "**gcc main.c**") and run the resulting executable (e.g. "**./a.out**" or "**./a.out -d dictionary.txt -n 4 -s word -f rung -m 25**") in the terminal 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 functionality grading.

Formally, you must submit your final program (**main.c**) to Gradescope, where your code will be manually inspected for style (details in the syllabus) and project requirements (details listed above). The official code submission is the version you submit to Gradescope (look for *Project03*).

The standard deadline for the project is **Thursday, October, 10th**. As detailed in the syllabus, early submissions receive +1 extra credit point/day early, up to a maximum of +3 points. Also, students are allocated a certain number of late days throughout the term, where no late penalty is applied. Students can use their late days whenever they choose, simply by submitting their work to Gradescope after the deadline. However, no more than two late days are allowed for each project.

The autograder has a test case suite to check functionality with a point breakdown given in the above task descriptions as follows:

- **Task I** - [10 points] - **strCmpCnt( )**
- **Task II** - [10 points] - **strCmpInd( )**
- **Task III** - [10 points] - **appendWord( )**
- **Task IV** - [8 points] - file-read the dictionary file to build the word list in **main()**

- **Task V** - [10 points] - **linSearchForWord()**
- **Task VI** - [10 points] - **checkForValidWord()**
- **Task VII** - [4 points] - **isLadderComplete()**
- **Tasks VIII & IX**- [16 points] - building incomplete ladders in **main()** and **displayIncompleteLadder()**
- **Task X** - [10 points] - **displayCompleteLadder()**
- **Task XI** - [6 points] - free all heap-allocated memory & avoid array out-of-bounds issues in **main()** and throughout program
- **Task XII** - [6 points] - handle command-line arguments in **main()**

Citation/Inspiration

The idea for this programming project is loosely inspired by Chris Gregg at Stanford University, with some of the introduction borrowed directly. However, the primary application and task breakdown was developed independently by Scott Reckinger.

Copyright Statement

LAB ACTIVITY | 14.3.1: Bit Big Bug Tug - Word Ladder Builder | 100 / 100 ✓

Run | History | Tutorial

main.c

```
/*------------------------------------------
Program 3: Bit Big Bug Tug - Word Ladder Builder

Description: This program prints out a word ladder where each word is c
```

```
and where each word higher up on the ladder has one different char thar
word. The program first takes in all of the words of the desired length
and puts it into a word array while dynamically allocating space as nee
creates a ladder with the words, where each next word only has one  cha
ladder array is also allocated space for the number of valid words the

Course: CS 211, Fall 2024, UIC

Author: Shreya Ganguly
------------------------------------------- */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>

```

DESKTOP CONSOLE

Submit for grading

Coding trail of your work [What is this?]

```
9/30 M0 ,0 ,0 ,10 ,20 ,0 ,20 ,20 M20 ,20 ,20 ,20 ,20 ,20 ,20 ,20 ,20
,20 ,20 ,20 ,0 ,0 ,20 ,0 ,0 ,30 ,30 ,30 ,30 ,30 ,30 ,32 ,30 ,30 ,33 ,30
,30 ,30 ,30 ,30 ,30 ,30 ,30 ,30 ,30 ,30 ,30 ,30 ,30 ,30 ,30 ,0 ,33 ,33
,33 ,0 ,30 ,30 ,33 ,34 ,30 ,33 ,33 ,33 ,33 ,33 ,0 ,33 ,33 ,33 ,33 ,23
,31 ,31 ,33 ,30 ,0 ,33 ,37 ,37 ,37 ,30 ,37 T30 ,30 ,30 ,38 ,38 ,0 ,30
,0 ,0 ,0 ,0 ,48 ,58 ,62 ,62 ,62 ,54 ,62 ,72 ,78 ,78 ,78 ,78 ,0 ,88 ,54
,54 ,54 ,88 ,0 ,94 ,94 ,94 ,94 ,94 ,94 ,0 ,0 ,0 ,94 ,0 ,32 ,74 ,32 ,74
,94 ,94 ,94 ,94 ,100 ,100 ,100 ,100 ,44 ,100 min:254
```

Latest submission - 9:36 PM CDT on 10/08/24

Submission passed all tests ✓

Total score: 100 / 100

✓ Only show failing tests *(36 tests hidden)*

Open submission's code

5 previous submissions

9:35 PM on 10/8/24	44 / 100	**View**
9:17 PM on 10/8/24	100 / 100	**View**
9:02 PM on 10/8/24	100 / 100	**View**
8:59 PM on 10/8/24	100 / 100	**View**
8:58 PM on 10/8/24	100 / 100	**View**

Trouble with lab?

bit big bug tug

ding dong done!

Feedback?

Activity summary for assignment: Programming Project 03 - Bit Big Bug Tug - Word Ladder Builder

100 / 100 points

Due: 10/10/2024, 11:59 PM CDT

This assignment's due date has passed. Activity will still be recorded, but will not count towards this assignment (unless the due date is changed). See this article for more info.

Completion details