CS211 – Programming Practicum Fall 2020

Programming Project 2

Due: Wednesday, 9/30/20 at 11:59 pm

Decoding a Secret Message

Alice and Bob are two spies that are under disguise in a hostile territory. They need to be in contact with each other regularly and share information about their mission. They cannot risk compromising their communication, so they decided to use encryption.
Alice and Bob both have two alphabet lists, the first one is the alphabet they use for sending their messages and the second one is a dummy alphabet to make it harder to decode their messages. The main alphabet is L and the dummy alphabet is L2:

L ={a, b, c, d}
L2 ={m, n, o, p}

The letters from L are mapped to letters in L2. So {a} is mapped to {m}, {b} to {n}, {c} to {o}, and {d} to {p}. They send information using letters from L, but they also add letters from L2 to increase the complexity of encryption. Based on their communication rules, when either of them sends a message, it must contain words that have letters from L and L2. There are some twists:

1. The meaningful words in any message they broadcast are words that are "completely mapped". All the other words are meaningless.
2. Since L2 is mapped to L, they can use both set of letters interchangeably: a string like "abnm" is indeed completely mapped since "n" is mapped to "b" and "m" is mapped to "a" in proper order.
3. The nesting and order of mapping is important. For example, "abmn" is not a valid word since the nesting and order is violated.
4. Once you find the mapped words, you have to remove all letters (including letters from L2) to find the actual message (which is all in L).

Example:

Encrypted Message: " abhcgmsopa bqcedpwon abmnpc abcdponm dfajbbmmn cabnmo"

Valid words: " abhcgmsopa bqcedpwon abmnpc abcdponm dfajbbmmn cabnmo"

Decoded Message: "bcd abcd cab"

Your task is to write a **C program** that reads in an encrypted message and outputs the decoded message. You **MUST** use a stack implemented in a dynamic array. This dynamic array is to grow to a larger size when a push operation would be done to a full array causing an array overflow. For this program, your dynamic array **MUST** start with 4 positions in the array. When the array needs to grow, it size MUST grow by 4 additional positions each time (note the array to grow in size from 4 to 8 to 12 to 16 to 20 to …).

The **push** operation is now defined as follows:

```
if (the stack array if full)
        grow the array
```

add the value to the stack array
increment the top-of-stack value

The **grow** operation is defined as follows:

Create a temporary pointer to the current stack array
Allocate a new dynamic array of the larger size and Have the stack array variable refer/point to the new dynamic array
Copy the existing values from the current stack array to the new dynamic array
Deallocate the current stack array
Update the maximum stack size variable

Input

The input for this program will come from standard input. Each line of input will be a series of string separated by space. You may assume that each line of input is less than 300 characters long. The program must loop to read in multiple lines of input. If the input on the line contains only the letter q or Q, quit the program.

Since we have limited the length of the input and are trying to process one line of input at a time, the best way to read the input is the **fgets()** function in the <stdio.h> library. Since we are reading from standard input, you are to use the value of **stdin** for the third parameter of **fgets().** This causes fgets() to read input from the standard input. **You MUST use fgets() for this programming project to read in the input**. See the base code to see how this is done. If you are not familiar with the fgets() functions, do a Google search and read. The cplusplus.com website has a good reference page on fgets().

Stack Data Structure in the Algorithm

For decoding the messages, we need to make sure every letter from L is perfectly masked by its corresponding letter form L2. This means the sequence of occurrences are very important in decoding the message. Letter "a" is only valid when it's properly mapped by its substitution from L2 which is "m". The letters from L2 must occur after the occurrence of corresponding characters from L. This means that an occurrence of letter 'm' is valid if there is a corresponding 'a' before it. We can also have nested maps which tricky to identify without a proper program.
For checking perfect mapping, we will use a stack data structure implemented in a dynamic array. The stack must be empty at the start of the expression.

The input can include all upper or all lowercase characters, or a mix of both. Your first task after reading the input message is to make all characters lowercase. After that, start the algorithm to detect whether a message is meaningful or not.
Start by reading the string from left to right. If you see any letters from L, push them into the stack. The stack at every step will only hold letters from L.
Once you encounter a letter from L2, compare it with the letter at the top of the stack.

- If the letter on the top of the stack is the perfect map for that letter, pop the stack and continue to the next letter.
- If the letter on the top of the stack does not map the letter from L2, the word is not part of the encrypted message and you can move to the next word.

- If the stack is empty, the word is not part of the encrypted message and you can move to the next word.

When the end of the word is encountered (i.e. the char null '\0'), check to see if the stack is empty.

- If the stack is empty, then the word was part of our encrypted message.
- If the stack is NOT empty, the word was not perfectly mapped and therefore meaningless.

It is important to note that we only care about letters from L and L2 (8 letters total), any other input can be ignored.
Also, note that we need every single letter from L to be perfectly mapped. In a meaningful word:

| # of letters from L2| = | # of letters from L|

Cleaning the message

Once you find all the meaningful words that form our encrypted message, you have to clean them to display the final message. Remove any character except {a, b, c, d} from every letter. You can use a function like RemoveExtraLetter() which takes in an array of character and properly trims it.

```
void RemoveExtraLetter ( char *str)
```

Output

For each line of input, your program should display the decoded message

- The decoded message will only consist of letters {a, b, c, d}.
- If all the words in a given line are invalid, output the following message:
  "No valid word decoded."

Use of C struct and C functions

When writing your code, you MUST place all of the data items needed for the stack in a C struct called "stack". These data items must include the following (and may include others if needed).

- the pointer to the dynamic array that actually holds the stack
- the integer variable specifying the current size of the dynamic array
- the integer variable specifying the top of the stack

The instance of this struct MUST be declared locally in main(). It may NOT be global. (Note: If you want it to be declared locally in some other function other than main(), that is also OK.)

In your program, you MUST write functions for:

- initializing the stack (typically named: init( )),
- checking if the stack is empty (typically named: is_empty( )),
- pushing/adding an element onto the stack (typically named: push( )),
- popping/removing an element off of the stack (typically named: pop( )),
- accessing/returning the top element on the stack (typically named: top( )), and
- clear the stack so that it is empty and ready to be used again (typically named: clear( )).

All of these functions MUST take **as their first parameter** a pointer to the struct that contains the instance of the stack that is being used. The only exception to this is that the initializing function may return a newly created instance and return a pointer to this instance.

Command Line Argument: Debug Mode

Your program is to be able to take one optional command line argument, the -d flag. When this flag is given, your program is to run in "debug" mode. When in this mode, your program is to display a message whenever an item is pushed or popped from the stack. This message must include the character being pushed or popped. Also, when the stack grows, you are to explicitly state the old and new size of the dynamic array for the stack as well as indicate the number of values copied from the current to the new dynamic array.

When the flag is not given, this debugging information should not be displayed. One simple way to set up a "debugging" mode is to use a "Boolean" variable which is set to true when debugging mode is turned on but false otherwise. This debugging mode variable can be set up as a global variable if desired. Then using a simple if statement controls whether information should be output or not.

```
if ( debugMode == TRUE )
     printf (" Debugging Information \n");
```

In order to pass the Gradescope tests, use the following printf statements:

Debugging Information for push:

```
printf("Character %c was pushed into the stack.\n", ch);
```

Debugging Information for pop:

```
printf("Character %c was popped from the stack.\n", ch);
```

Debugging Information for grow array:

```
printf("The stack has grown from %d to %d, and a total
        of %d values were copied into the larger
        array.\n", oldSize, newSize, numElemenets);
```

Coding Style

Don't forget to use good coding style when writing your program. Good coding style makes your program easier to be read by other people as the compiler ignores these parts/differences in your code. Elements of good code style include (but may not be limited to):

- Meaningful variable names
- Use of functions/methods
- Proper indentation
- Use of blank lines between code sections
- In-line comments
- Function/method header comments
- File header comments

The Code Review Checklist also hints at other elements of good coding style.

Program Submission

You are to submit the **main.c** file for this project via the proper submission link on **Gradescope**.