Lab 3 Activities [30 Marks]

An Autograded Evaluation

For the lab portion submitted for autograding, do not use `printf()` or `scanf()`.

IMPORTANT: **Your submission must compile without syntactic errors to receive grades. Non-compilable solutions will not be graded.**

IMPORTANT: **Penalties will be applied for poor memory management practices. Refer to marking schemes for details to watch out for.**

About Lab 3

- Use the following link to accept the lab invitation: https://classroom.github.com/a/E-VbGPYK
- Lab 3 is due on **Friday, November 8, 2024, at 11:59 pm** on Github.

- You are allowed to use functions from <stdio.h> and <stdlib.h>, both of which are included in the skeleton code.
- **String Library and other C Libraries are NOT PERMITTED.**
  Global Variables are NOT PERMITTED.
  Including them will result in zero for the question.

- **Memory profiler tools will be used to check for any memory leakage originating from within your implementation.**
  Marks will be deducted for poor memory management practices.

- **Note: You must provide your student information (Section, Mac ID, and Student ID) in the README file of the Lab 3 code package. Failure to do so will result in a 5% penalty on the entire Lab 3 grade.**

- **Lab 3 comes with launch.json to enable the VSCode debugger.**
  You need to set the workspace to your Lab 3 folder to start the debugging session.

Important Notes about Memory Leakage Evaluation

Given that memory leakages may take place within the installed toolchains and third-party libraries, you are not expected to eliminate all memory leakages captured by the memory profiler tool on your system. You are, however, required to **analyze the report and ensure that none of the leakages originate from malloc() calls within your code implementation**.

REVIEW: How to Perform Memory Usage Profiling

Windows

In VSCode terminal, under your Lab 3 folder containing the compiled Lab3.exe, type in:

```
drmemory ./Lab3.exe
```

The memory leakage report will be generated.

MacOS

In terminal, navigate under Lab 3 folder with the compiled Lab3 executable, do the following:

1. Enter: `export MallocStackLogging=1`
2. Enter: `leaks --atExit --list -- ./Lab3`

The memory leakage report will be generated.

Linux / Chrome

In terminal, navigate under Lab 3 folder with the compiled Lab3 executable, type in:

```
valgrind –leak-check=yes ./Lab3
```

The memory leakage report will be generated.

Lab Question 1 – Custom String Library Prototyping [15 marks]

This will be the first custom C library we are going to construct in 2SH4. Please rest assured that this library will be very handy throughout your project development process. This lab question requires you to construct the following four string library functions. You must ensure that each library function, with your custom implementation, passes all the unit tests before being packaged into a reusable string library in PPA3.

- `int my_strlen(const char * const str)`
  - This function returns the number of characters up to but NOT including the null termination character.
  - For example, given the input `str = "Hello"`, the return value should be 5, not 6.

- `int my_strcmp(const char * const str1, const char * const str2)`
  - This function compares the two strings and returns 0 if their contents are different, and 1 if identical.

- `int my_strcmpOrder(const char * const str1, const char * const str2)`
  - This function offers an extended functionality from `my_strcmp()` by identifying the ASCII character order between `str1` and `str2`.
    - If the contents of `str1` are larger than `str2` in ASCII order, return 1
    - If the contents of `str1` are smaller than `str2` in ASCII order, return 0
    - If the contents of the two strings are identical, return -1
  - For example:
    - Given `str1 = "Hello"`, `str2 = "World"`, `my_strcmpOrder(str1, str2)` will return 0, because the first different character between `str1` and `str2` is `'H'` (72) vs `'W'` (87).
    - Given `str1 = "Johnny"`, `str2 = "Jeffery"`, `my_strcmpOrder(str1, str2)` will return 1, because the first different character between `str1` and `str2` is `'o'` (111) vs. `'e'` (101).
    - Given `str1 = "Bless"`, `str2 = "Bless You"`, `my_strcmpOrder(str1, str2)` will return 0, because the first different character between `str1` and `str2` is `NULL`(0) vs `' '` (32)

- `char *my_strcat(const char * const str1, const char * const str2)`
  - This function returns the concatenated string of the two input strings in the following way:
    - Allocate a character array capable of holding the combined string contents of `str1` and `str2` on the heap.
    - Copy the non-`NULL` contents in `str1`, then `str2`, into the character array, then terminate the concatenated string with a `NULL` character.
    - Return the pointer to the concatenated string on the heap.
  - For example:
    - Given `str1 = "Hello "`, `str2 = "World!"`, `my_strcat(str1, str2)` will return a pointer to the string `"Hello World!"`.
  - **WARNING!!!** This is the first **Memory Unsafe** function requiring your attention to heap memory allocation and deallocation. For your learning purpose, you only need to worry about `malloc()` in the function implementation for now. The `free()` counterpart is already taken care of in the unit test setup in `testCases.c`. However, you need to carefully analyze how your code interacts with the unit test cases – focus on understanding where the heap memory for the combined string is allocated **and deallocated**, and its full lifetime on the heap. Getting familiar with this will help you carry out the full course of memory management in Question 2.

Marking Scheme

- Code Implementation
  - **[1 mark]** Correct implementation of the `my_strlen()` function
  - **[2 marks]** Correct implementation of the `my_strcmp()` function
  - **[3 marks]** Correct implementation of the `my_strcmpOrder()` function
  - **[4 marks, with breakdown below]** Correct implementation of the `my_strcat()` function
    - [1 mark] Allocated correct number of character spaces to hold the concatenated string
    - [2 marks] Copy the characters from both strings in the correct order
    - [1 mark] Correct insertion of the NULL character

- Code Behaviour, Analysis, and Test Plans
  - **[5 mark]** Correct Program Behaviour – Passing all default test cases.

- Memory Management Penalties
  - **Segmentation Failure will lead to additional 2 marks deducted.**
  - **Any detected memory leakage will lead to additional 4 marks deducted.**

Lab Question 2 – Sorting Arrays of Words in File [15 marks]

In this question, you will apply the custom string library built in Question 1 to sort an array of strings read from a provided input file `wordlist.txt` using the two sorting methods we've covered in the last two labs – *Bubble Sort* and *Selection Sort*. You will have to complete the following three interface functions and two helper functions to complete the required tasks. A quick instruction guide on how to access files in C is provided at the end of the lab manual. You will need to read it to complete this question.

As another debugger practice, the *Selection Sort* implementation of the word sorting function is provided, but with 2 semantic bugs. Find them using the debugger, fix the bugs, and produce the debugging report.

- **`char **`**`read_words(`**`const char *`**` input_filename, int *`**` nPtr)`**
  - This function opens the file pointed by `input_filename` in ASCII read mode, reads each line in the file as a string, and saves the individual strings in the array of strings (2D array of characters) on the heap. The address of the array of strings on the heap is returned.
  - Use the tips in the comments in the skeleton code to help you develop and test the implementation.

- **`void`** `swap(`**`char **`**`str1, `**`char **`**`str2)`**
  - This function swaps the contents of two strings via pointer operations.
  - This function accepts the starting address of the two strings in pass-by-reference mode. NOTE: you should NOT interpret the two parameters as plain double pointers.
  - Swapping the contents of the two strings is as simple as swapping the starting addresses stored in the two *str* pointers. Review lecture notes to understand the mechanism.

- **`void`** `delete_wordlist(`**`char **`**`word_list, `**`int`**` size)`
  - This function deallocates the array of strings allocated on the heap.
  - This function accepts the pointer to the array of strings, and the number of strings in the array.
  - Review lecture notes to develop the correct memory deallocation routine.

- **`void`** `sort_words_Bubble(`**`char **`**`words, `**`int`**` size)`
  - This function sorts all the words in Ascending ASCII Value Order using *Bubble Sort* algorithm.
  - Use `my_strcmpOrder()` from Question 1 to help you compare the strings.

- **`void`** `sort_words_Selection(`**`char **`**`words, `**`int`**` size)`
  - This function contains a faulty implementation of the word-sorting algorithm using *Selection Sort*.
  - The function contains 2 semantic bugs. Find them and repair them using the VSCode IDE Debugger and produce a debugging report.

For your workload management, **DO NOT** try to come up with an array size larger than 10.

Marking Scheme

- Code Implementation
  - **[2 marks]** Correct implementation of `read_words()` function
  - **[1 mark]** Correct implementation of `swap()` function
  - **[2 mark]** Correct implementation of `delete_wordlist()` function
  - **[2 marks]** Correct implementation of the `sort_words_Bubble()` function
  - **[2 marks]** Correct repair of the `sort_words_Selection()` function
- Code Behaviour, Analysis, and Test Plans
  - **[2 mark]** Correct Program Behaviour – Passing all default test cases
  - **[4 marks, breakdown below]** Debugger Report
    - For each of the two bugs in the `sort_words_Selection()` function
      - **[1 mark]** Setting up effective breakpoints
      - **[1 mark]** Setting up effective variable watches
      - **[1 mark]** Showing evidence of the buggy behaviour
      - **[1 mark]** Showing evidence of the fixed behaviour

- Memory Management Penalties
  - **Segmentation Failure will lead to additional 2 marks deducted.**
  - **Any detected memory leakage will lead to additional 4 marks deducted.**

Quick Start Guide on C/C++ File Access

The file access library in C and C++ is extensive and versatile, but mostly operating at either Binary mode or ASCII character mode. This quick start guide is going to cover the required file access features in Lab 3 – Reading from a file in ASCII read mode. The remaining functionalities are left for your self-directed learning. To use the C file system, you must include `stdlib.h` in the C source code.

Step 1: Create a File Handle (Pointer to the Start of the File)

To start the file accessing process, we must create a file pointer to hold the address of the start of the file. This is done by declaring at the beginning of the file access routine:

```
FILE* myFile;   // commonly known as the "File Handle"
```

Step 2: Open a File in ASCII Read Mode

Then, we will open our desired file using the `fopen()` function, which accepts the file name in the first parameter, and the access mode in the second parameter – `"r"` for ASCII read mode. There are many other access modes you can use.

Upon successfully opening the file, `fopen()` returns the address of the start of the file, which we should capture using the file handle from Step 1. If access failed, `fopen()` will return a NULL pointer, which can be used for error checking.

```
myFile = fopen("FileName.doc", "r");
// Open the file FileName.doc in ASCII read mode
```

From this point on, accessing the file contents can be done via `myFile` file handle. The file you've opened then remains locked and monopolized by your program until you close the file access in Step 4. Note: The extension of your file may be something other than .doc, such as .txt, and so on.

Step 3: Read Lines of Contents in Desired Formats

To read lines of contents from a file, we will use the fscanf() function as follows:

```
char buffer[20];
fscanf(myFile, "%s", buffer);
```

This code snippet carries out the following operations:

- Read one line of contents from the file, up to the newline character. Depending on your OS, the newline character may or may not be included. You need to check it by printing it out for confirmation.
- Interpret the read contents as a string. Change the conversion specifier for different value interpretations.
- Save the string into the string buffer. Watch out for buffer overflow if the string is larger than the buffer size.
- **Unobvious but CRITICAL**: Advance the address in the file handle to the next line in the file, so that when fscanf() is called the second time, it will read from the subsequent line of contents.

Call `fscanf()` as many times as needed to read contents out of the file. You may research into other file reading functions and **End-Of-File (EOF)** flag for additional code reliability when reading from files.

Step 4: Close the File Handle (i.e., Release the Access to the File)

After the file access is completed, you will have to close the file access on the file. This step is a ***MUST-DO***!! Otherwise, the file will remain inaccessible to other programs.

```
fclose(myFile);   // commonly known as "release the handle"
```