50.051 - Lab 1 – Week 3

1. Please submit your assignment as a Zip file. This zip may contain any number of `.c` files and `.h` files. For every `.c` file, there should be a `.h` file. Keep every file used from the lab template as they will be used for testing your code and as input to your program.
2. Please download the lab template, which contains a .c and .h file with specific global variables and function declarations you will need to use for this assignment.
3. Your assignment should compile using `-ansi -pedantic -Wall -Werror.`
4. Please submit on eDimension by Wednesday, 21 February, 23:59.

This assignment is mostly about strings (arrays of characters) and pointers, and encompasses two different parts. Read the instructions first, they also explain how to use the different files available in the template.

Throughout the assignment, we use the wording "string" to refer to an array of characters, or a pointer to characters (`char *`).

To help you with memory management when compiling your code, you may also use the following flag: `-fsanitize=address` which will let you know if there is any potential memory that is allocated but not freed, or out of bounds access. assignment is mostly about strings (arrays of characters) and pointers, and encompasses two different parts. Read the instructions first, they also explain how to use the different files available in the template.

Part 1: (Trying to) Detect the Language of a text

For this part, you will try to determine whether a text is most likely to be in the English or French language based on the frequency of occurrence of the letters.

The lab1.h file provided in the template for this lab contains the definition of a constant called `NUM_LETTERS`, and two float arrays containing 26 floating point numbers which store the frequency of appearance of the 26 letters from the latin alphabet, respectively in English (`englishFreq`) and French (`frenchFreq`).

Using argc and argv

We want the user to provide the name of the text file to be processed by the user when running the program, e.g. by typing `./lab1 sample-file.txt` on the command line. Before doing anything in the main, please make sure that the file name is indeed provided when calling the program. To do so, you will need to check the value of `argc`, and if it is not correct, print an error message and return 1. If the value is correct, you can make use of `argv` to retrieve the file name (`argv` can be seen as an array of strings).

Reading from a text file (provided code)

For this solution to work, you will need to use a text that is large enough. The template contains two sample files `english-sample.txt` and `french-sample.txt`, which can be used to test your code. To convert a file to a string (or array of chars), you should use the function called file_to_str, which is defined in the .h file and implemented in the .c file of the template. The code of the function is voluntarily messy and obfuscated but should work reliably to read the entire content of a text file (provided as an argument) and return it in a single, large, array of characters.

Computing the frequency of every letter

Once you have retrieved the content of the file in a char *, you may process it using the calculateLetterFrequency() function defined in the .h file.

```
void calculate_letter_frequency(char *text, float *frequency);
```

This function takes the text to be processed, and a pointer to float (which can be an array of 26 elements) which will be populated by the function. Here are some key points to consider where computing the frequency of appearance of letters in a text:

1. Any character that is NOT a letter should be ignored (digits, punctuation, spaces…)
2. Upper case and lower case do not matter. As such one occurrence of 'a' and one occurrence of 'A' should be treated as two instances of the same character.
3. You can assume that the French samples never contain characters with accents.
4. You can and are encouraged to create helper functions to make your code clearer. For example, a function that would convert a character to lower case, or check if a character is a letter or not.
5. You are free to use extra arrays or any number of local variables inside the function.

Note that this function DOES NOT return anything. Simply, the array (or `float *`) given as an argument will be modified inside the function.

Computing the difference between observed and expected frequencies

Now that you have the observed frequency of the 26 letters, you should compare your expected frequency array to both the english and french expected frequencies. To do so, please write a function with the following declaration:

```
float calculate_difference(const float *observedFreq, const
float *expectedFreq);
```

This function computes a difference score (a float) as follows:

$$similarity = \sum_{i=0}^{25} (observed\_frequency_i - expected\_frequency_i)^2$$

which is basically the squared differences between each individual observed and expected frequency.

Note that this **difference score should be as close to 0 as possible**. Ultimately, your program should output either:

- "The text is likely to be English", if the English difference score is lower than the French difference score
- "The text is likely to be French." Otherwise.

After outputting either, you can simply return 0; and end the program.

Try it with the different text files. The english-test1.txt and english-test2.txt should be recognized as english, same for the french-test1.txt and french-test2.txt. What happens with the mystery files (`mystery-sample1.txt`, `mystery-sample2.txt`, `mystery-sample3.txt`). What can you tell about your code in terms of classification of texts?

Part 2: String Processing

For this second part, you will need to write a few functions for string processing. You are allowed to use the following functions from the `string.h` library (and only these functions from the library):

- `strncpy` and `strcpy`
- `strlen` and `strnlen`
- `strstr` and `strnstr`
- `strdup` and `strndup`
- `strcat` and `strncat`

Splitting a text into sentences

First, we want to be able to split a large text (array of chars) into an array of strings (array of array of chars), where each element is a single sentence. Please refer to the following function declaration:

```
char ** split_sentences(const char *text);
```

You will need to implement this function, which takes a `char *` as an argument and returns a `char **` which is an array containing every sentence from this text. This array should end with a `NULL` pointer, so we can easily determine when the array ends. You will need to dynamically allocate the `char **` you want to return.

You can consider that any sentence in a given text ends with either '.', '!' or '?'.

Please note that the input is a `const char *`, which means that the values stored inside this array of characters cannot be modified. In the general case however, `const char *` may point to different addresses over time.

Once this function is implemented, you will need to implement two more functions:

`void print_string_array(char **strings);` which prints every single string stored in a `char **`.

`void free_sentences(char **sentences);` which properly frees the array of strings that was previously allocated.

Here is the example of an input:

```
"This text contains two sentences! Or maybe more? At least two. Maybe six"
```

Here is what the output should look:

```
{ "This text contains two sentences!", "Or maybe more?", "At least two.", "Maybe six", NULL }
```

Replacing words/phrases in a text

For this final feature, you will need to write a function which takes multiple pairs of strings, e.g. "cat" and "dog" and replaces every occurrence of the first value of a pair with the second value.

For example with the two pairs "fox", "squirrel" and "lazy", "handsome", the following sentence:

```
The quick brown fox jumps over the lazy dog. What a fox!
```

becomes:

```
The quick brown squirrel jumps over the handsome dog. What a squirrel!
```

Here is an incomplete declaration for this function:

```
char * change_wording(const char *text, ...)
```

The first argument is the original text (which should not be modified), the second argument should be basically a collection (list, array) of pairs of strings (in the example above, "fox", "squirrel" and "lazy", "handsome").

The function should return a new text with the replaced occurrences.

Constraints:

- You may only use the functions from `string.h` listed at the beginning of part 2. No other functions (e.g. from `ctype.h`) other than the usual functions used to deal with pointers and printing are allowed.
- You can create your own data type for storing the pairs. Please note that you can only add one argument to the function above. You may use struct if you want.
- Your final string should be properly allocated with malloc and have enough space to store the final sentence.
- You should not assume that two strings in a pair will have the same length.
- The replacement process is case insensitive. "FoX", "foX" and "Fox" should all be replaced with "squirrel" in the example above.
- It is acceptable for the final output to be entirely in lower case.
- If any dynamic allocation of memory operation fails, you should print an error message and exit the program (use `exit()` function).