Tokenizer Review Mini-Project

This exercise will allow you to leverage everything we covered so far to develop a string tokenizer. Similarly to the *strtok* function from the standard library, we want to build a function which will take a string containing several words separated by white spaces or tabulations, then build and return an array of strings in which each entry is a copy of one of the words in your original input.

We will, later in the semester, revisit this program and expand it so that we use these tokens to do something useful but right now our main objective is to implement the functions of this program so that we are capable to tokenize any string and display the result on the screen.

Step #1: Implementation of the helper functions

Start by studying the source in the files main.c, tokenizer.h, and tokenizer.c. The latter is the file in which the tokenizing function & its helpers are implemented.

Start by making sure the data structure to store these tokens is well understood by studying the *tokens_allocate* and *tokens_deallocate* functions. You should then study the implementation of the *tokens_display* function.

Step #2: the tokens_add function

Before to go any further, study the *tokens_add* function. Trace the execution of the program from within the main to understand how it is being used.

Step #3: the tokenize function

This function implements the main functionality of this program and allows us to identify words inside a string and call the *tokens_add* function to add each of these words to our data structure containing all the tokens we identified.

Start by understanding its algorithm on paper and by working through it before to trace the execution through the source.

To tokenize a string, you will need to analyze it character by character and keep track of where a word starts (by the index of the first character of this word) and where it ends (by the index of the last character of this word). Once you have those two indexes for a given word, you can call the *tokens_add* function and then skip all the white spaces that follow it until you find another non white-space character.

This can be achieved by using a variable (an *int* for instance) to keep track of what you are looking for. Let's say that we call this variable *state* and we assign to it the initial value of 0, meaning that we start exploring the string looking for the first letter of a word. Then we can enter a loop to iterate over all the characters of the string from index 0 to the end. For each character, if we are in *state* 0, then we test if the character is alphanumeric. If so, we assign its index to a variable *start* to remember where the word starts. We then change our *state*, we are no longer looking for the first character of a word but the first white space following one. Let's assign the value 1 to *state* to represent this and then we iterate again in our main loop to go look at what the next character is. The same way we had a if statement testing, if we were in *state 0*, whether we encountered an alphanumerical character, we need to also have a if statement testing, if we are in state 1, whether we encounter a white space. If this is the case, then we mark this index by storing it in a variable *stop* and call *tokens_add*.

This is not the full story but you should get enough hints from this to get you to understand the implemented algorithm. Use the forums to post questions and get help with this.