Background

In the APP C30-1 assignment, all of the strings in the input file are separated on different lines such that reading in one string would read only one name or language at a time. However, text data will not always be available in such a nice form. For example, the **comma-separated values (CSV)** file type stores individual text strings separated by commas. This means that reading in one full string would read in all of the individual text strings and commas together. This full string must be manipulated in order to extract specific information.

The process of analyzing a string of characters like this is called **parsing**. One use of parsing is to extract specific information from a string which has several separate pieces of information separated by special characters called **delimiters**. Some common delimiters are spaces (`' '`), commas (`','`), and brackets (`'}'`). The subsets of characters (pieces of information) in between these delimiters which can be extracted are called **tokens**. Thus, this use of parsing can be summed up as analyzing the string to extract the tokens which are separated by the delimiters. (Another related use of parsing is to determine if a command string has proper syntax and to break it into its components for the compiler. This is called **syntax analysis**.)

In this extension assignment, you are given the text file `APP_C30_1_EXT_chemicals.txt`. This file contains information on the chemicals used in a chemical cleanroom over the course of two weeks. Each line represents one day, and there are 14 days total. Each line has two components: a string CSV list of chemicals used that day and an integer number representing the number of characters in the chemicals list string, including commas. The maximum number of characters for any individual chemical name is 10 characters (including null character). The cleanroom staff members are interested in parsing this file to determine a list of unique chemical tokens used in these two weeks. Thus, the end product of this program will be a list of all of the unique chemical tokens in the file.

However, the cleanroom staff are also interested in scaling this up to larger periods of time with possibly higher numbers of chemicals used each day, and they want to use as little computer memory as possible when storing and manipulating the information. Thus, you will use dynamic memory allocation and reallocation to handle these requests.

The `realloc()` function (which stands for memory **realloc**ation) takes two inputs: the pointer to memory which you wish to reallocate and the number of total bytes of memory to allocate. It outputs a pointer to the beginning of the new allocated memory. This function can be used to decrease or increase the size of memory pointed to by a pointer. A pointer must have been initialized with memory already before it can be reallocated (there are multiple ways to do this). An example of using `realloc()` to reallocate a pointer to characters is shown below:

```
    char *pointer;
    pointer = (char *) malloc(5*sizeof(char));
    pointer = (char *) realloc(pointer, 10*sizeof(char));
```

The last main dynamic memory allocation function is `calloc()`. This function works similarly to `malloc()` in that it allocates memory to a pointer, but it also initializes the data values by clearing them to zero or empty.

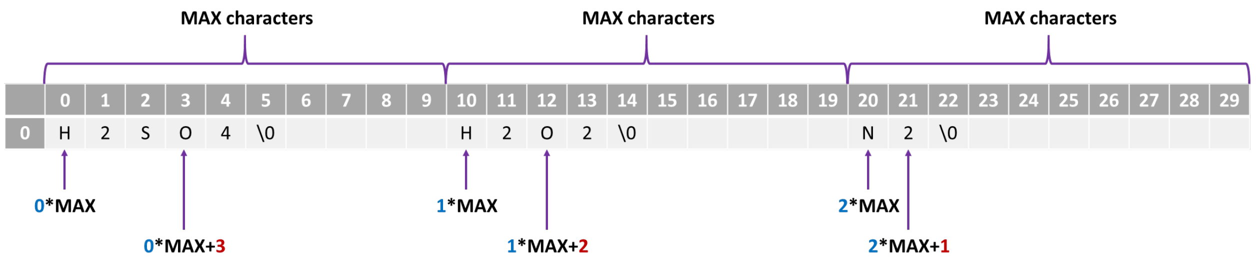
Some string manipulation functions which you might find useful are: `strlen()`, `strcmp()`, and `strcpy()`.

Instructions

- Complete the C30-1 assignment which introduces strings in C.
- Copy both `APP_C30_1_EXT_chemicals.txt` and `APP_C30_1_EXT_SKELETON.cpp` from:

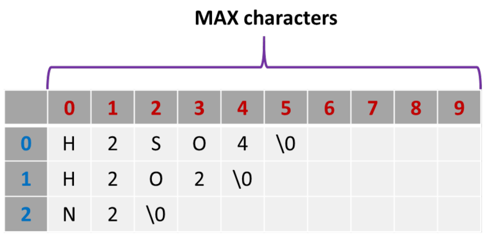
  `/share/EED/class/engr1281/students/c/Class_30/Application`

- Begin by opening the skeleton code and adding your header information.
- The whole code structure and most of the code itself is provided for you. However, you must fill in the blank spaces to complete the reading in of each line of the file, stepping through each character in each line to determine if it is part of a token, a delimiter, or the null character, assigning each individual character of a token to a token pointer in the proper spot, comparing tokens to determine unique chemical names, reallocating the tokens pointer to handle an increasing number of unique tokens, and printing the found unique token chemical names to the screen.
- **Hint #1:** The tokens pointer will point to memory storing all of the saved unique token chemical names. Each name will be a string of characters ending in a null character. Since you have been told that that the maximum number of characters for any one chemical name, including the null character, is 10 characters, 10 bytes of memory will be allocated for each unique token even if that token does not need all 10 characters. This is the defined constant `MAX` in the code.
- **Hint #2:** Related to Hint #1 above, since each token will have 10 bytes allocated for it, these tokens are stored sequentially in memory, one after the other. The beginning of the second token is 10 memory locations AFTER the beginning of the first token. This is actually how matrices/arrays are stored in memory, and you can think of this tokens pointer as kind of like a 2D array. To store individual characters, you need to know which token you are looking at (row number of an array) and know which character in that token you want to set (column number of an array). The variables `m` and `n` in the skeleton code act as these row and column indices. Figure 1 below shows the real storage of the tokens in memory, and Figure 2 shows the equivalent 2D array representation. Note that the variable `m` is the blue row/token numbers, and the variable `n` is the red column/character numbers. Also, note that the indices for an entire token (`m*MAX`) and for a specific character in a token (`m*MAX+n`) must be used with reference to the pointer for the allocated memory (`tokens`).



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
0	H	2	S	O	4	\0					H	2	O	2	\0						N	2	\0							

Figure 1: Actual memory sequential storage for tokens with reference to the allocated pointer.


MAX characters

	0	1	2	3	4	5	6	7	8	9
0	H	2	S	O	4	\0				
1	H	2	O	2	\0					
2	N	2	\0							

Figure 2: 2D array representation of the tokens.