System Programming in C – Homework Exercise 5

Publication date: *Tuesday, June 13, 2017*
Due date: *Tuesday, June 27, 2017* @ 21:00

General guidelines:

- Start by copying all the files from directory `/share/ex_data/ex5/` to your exercise directory (note that before submitting, you should delete everything except the two C source files you edited).
- We have provided a **Makefile** and testers you can use to build and test your solutions. In order to get a list of available makefile commands, run “make help” in the exercise directory. (Running “make” without parameters will attempt to build everything and run all tests). Note that even if the tests pass, this does not guarantee your code is correct (they will only catch some of the bugs). We recommend that you implement additional tests to validate your implementation against the specification in the assignment.
- Your code should compile on the server using **`make`** without warnings.
- Document your code, and use proper indentation. Each function should have a documentation block with a brief description above its definition. You should also document the critical parts of each function’s implementation. See **`/share/classes/class07/power.c`** for reference.

  5 bonus points will be awarded for good documentation and form.

Problem 1:

The purpose of this problem is to implement a simple data structure for a list of strings. Like strings in C, a **string list** is implemented using a simple array of characters. The list consists of a sequence of consecutive strings separated by ‘\0’ characters, where the end of the list is indicated by two consecutive ‘\0’ characters.

The following example illustrates an array of size 15 (e.g., char arr[15]) that contains a string list with two strings (“Hello” and “world”). The last two entries in the array (arr[13] , arr[14]) are not part of the string list.

H	e	l	l	o	\0	w	o	r	l	d	\0	\0		

Implement the functions in `stringList.c` as specified below.

1. Write a function named **`numStrsInList`** that returns the number of strings in a given string list.
   - The function signature should be `int numStrsInList(const char* strList).`
   - The function receives pointer to the beginning of a string list and returns the number of strings in the list (until the terminating '\0\0' characters).

2. Write a function named **`strListLen`** that returns the number of characters in a given string list.
   - The function signature should be `int strListLen(const char* strList)`.
   - The function receives pointer to the beginning of a string list and returns the total number of characters in the list, including separating '\0' chars, but excluding the two terminating '\0' chars.

3. Write a function named **`emptyStrList`** that receives a pointer to a character array and writes an empty string list into that array.
   - The function signature should be `int emptyStrList(char* target)`.
   - The function receives a pointer to an array **`target`** of size ≥ 2 (to hold the two terminating '\0' chars), and it writes into **`target`** an empty string list.
   - Return -1 if target is NULL, 0 otherwise

   Note: functions **`numStrsInList`** and **`strListLen`** should both return 0 when invoked on an empty string list.

4. Write a function named **`strListFromWords`** that receives a pointer to a string containing words separated by white spaces, and creates a string list holding these words as separate strings within a given pre-allocated character array.
   - The function signature should be
     `int strListFromWords(const char* source, char* target, int buffSize)`.
   - The function receives a pointer to the source string (**`source`**), a pointer to the target character array (**`target`**), and the size of the target array (**`buffSize`**).
   - The function creates a string list inside the **`target`** array, and returns the number of strings in the list.
   - Strings in the list correspond to words in the **`source`** string, where a word is a maximal contiguous substring of source that does not contain white spaces (chars ' ', '\t', '\n', '\v', '\f', '\r'). You may use function **`isspace`** from library ctype to identify space characters. For instance, the words in string "Hello \tWor1d!" are "Hello" and "Wor1d!".
   - The strings in the resulting list should be in the same order as words in the source string.
   - Parameter **`buffSize`** indicates how many characters can be written into the **`target`** array (including all required '\0' characters). Thus **`buffSize`** should be ≥ 2 (to hold the two terminating '\0' chars). If **`buffSize`** < 2, the function should return -1 without doing anything. If **`buffSize`** ≥ 2, but the list requires more characters than **`buffSize`**, the function should write an empty list in **`target`** and returns 0.
   - See execution example in `testProblem1.out` for reference.

5. Write a function named **`createStrList`** that creates a string list in a newly allocated array from words in a given string.

- The function signature should be `char* createStrList(const char* str)`.
- The function receives a pointer to a string (**`str`**), and creates a new string list by allocating space and copying words from **`str`** into the list.
- The string list should be created the same way specified in (1.4) above for **`strListFromWords`**.
- The function should return a pointer to the newly created list.
- The space allocated for the string list should be exactly the space required to hold all strings and '\0' characters.

6. Write a function named **`nextStrInList`** that returns a pointer to the next string in a given list.

- The function signature should be `char* nextStrInList(char* strList)`. (parameter should not be `const char*`)
- If the previous call to this function (the one right before the current call) was not done with the same list (or this is the first call to the function in the program), then the function should return a pointer to the first string in the list.
- Otherwise (previous call to this function was done with the same list), the function should return a pointer to the string in the list that follows the last string returned by the function. If the string returned by the last call was the last string in the list, then the function should return NULL.
- If a string list **`strList`** contains two strings: "Hello" and "World", then the 1st call **`nextStrInList(strList)`** should return a pointer to "Hello", the 2nd call **`nextStrInList(strList)`** should return a pointer to "World", and the 3rd call **`nextStrInList(strList)`** should return NULL.
- Providing the function with a NULL argument (**`strList`**==NULL) is used to reset the function's recollection of the previous list (return NULL in this case).

Hint: static variables…

Problem 2:

The purpose of this problem is to implement a "sort" program that uses string lists and the functions that you implemented in Problem 1. The functions below should be implemented in `stringList.c` . For Q2.6, you will edit the file `sortFile.c`. Execution examples for the sort program (which utilizes all functions) are given on page 7.

1. Write a function named **`strcmpInList`** that compares two string lists according to the lexicographic order of the k'th string in both lists.

- The function signature should be
  `int strcmpInList(char* strList1, char* strList2, int k).`
- The key **`k`** should be a positive integer (>0) that indicates which strings in the two lists to use as a key for comparison. First string is indicated by 1, etc.
- The function should return:

  0 if the k'th strings in the two lists are **identical**

  1 if the the k'th string in strList1 is **lex. larger** than the k'th string in strList2

  -1 if the the k'th string in strList1 is **lex. smaller** than the k'th string in strList2

  -10 if either of the strings is NULL

- Use the function **`strcmp`** in the `string` C library to compare individual strings. Note that the lex. order assumed by **`strcmp`** uses the ASCII value of characters. Thus, all upper-case letters have lower value than all lower-case letters (e.g., 'Z' < 'a'). This is fine for the purpose of the assignment. **Note that strcmp is not guaranteed to return -1, 1, or 0 – just a negative, positive or zero value.**
- An empty string is smaller than all other strings (i.e., if one of the lists does not have k strings), and two empty strings are identical (i.e., if both lists do not have k strings).
- Use **`nextStrInList`** from (1.6) above to get the two relevant strings.
- You may assume that k>0.

2. Write a function named **`strListCmp`** that compares two string lists according to a given list of prioritized keys.

- The function signature should be `int strListCmp(char* strList1, char* strList2, const int keys[], int numKeys).`
- Parameter **`keys`** should be a pointer to an array of `numKeys` positive integers (>0), that act as (prioritized) keys for a sorting procedure.
- Keys in the array indicate, according to their order, which list is "larger" according to function **`strcmp`**, as described in (2.1) above.
- The function returns 0 iff the lists are identical according to all keys.
- The function returns 1 iff the first key in the array for which the two lists are different indicates that `strList1` is **larger** than `strList2`.
- The function returns -1 iff the first key in the array for which the two lists are different indicates that `strList1` is **smaller** than `strList2`
- The function returns -10 if any of the pointer inputs are NULL.
- Use **`strcmpInList`** from (2.1), and you may assume that all keys are positive.

3. Write a function named **`strListSort`** that sorts a given array of string lists in ascending lexicographic order according to list of prioritized keys.

- The function signature should be `int strListSort(char** strListArr, int numLists, const int keys[], int numKeys);`.
- Parameters **`keys`** and **`numKeys`** represent a list of keys, as in **`strListCmp`**.
- Return -1 if any of the pointer inputs are NULL, and 0 otherwise.
- You may assume that **`keys`**>0.
- Parameter **`strListArr`** points to an array of pointers to string lists, and **`numLists`** indicates the number of lists in the array.
- The function sorts the lists in the array according to ascending lexicographic order (from "small" to "large") and according to the prioritized list of keys.
- Use a simple $O(n^2)$ sorting algorithm.

4. Write a function named **`printStrLists`** that prints a given array of string lists.

- The function signature should be `int printStrLists(char** strListArr, int numLists);`
- Return -1 if strListArr is NULL, 0 otherwise.
- String lists should be printed in order, each one in its own line.
- Strings in a list should be printed in one line (in order), separated by tab characters ('\t').

5. Write a function **`string2posInt`** that reads a string that consists of digits and returns the integer value represented by the string.

- The function signature should be `int string2posInt(const char* str);`
- If the string contains a non-digit character, then the function returns 0.
- The function <u>does not</u> have to check for overflow (beyond max value for int).

Examples:

- **`string2posInt("123") = 123`**
- **`string2posInt("-123") = 0`**
- **`string2posInt("1.3")  = 0`**

6. Write the code of the **`main`** function of `sortFile.c` to implement a program similar to the `sort` command in Linux. The program should be executed as follows:

```
./sortFile file key1 [key2 ...]
```

The first argument is a path to a readable file, and it is followed by a list of positive integers (>0) representing sorting keys. The program should print the lines of the input file to the standard output, sorted in ascending lexicographic order of the words indexed according to the keys. For instance, executing `./sortFile myFile 4 2` will print the lines of `myFile` sorted primarily according to the lexicographic order of the 4th word in each line, and secondarily according to the lexicographic order of the 2nd word in each line. Lines that are identical in their 4th and 2nd words should be printed according to their order of appearance in the file. [similar to what `sort myFile -k4,4 -k2,2` does]

- Use dynamic allocation to create the integer array that holds the sort keys.
- Your main function should use a single array variable to read lines using the **`fgets`** function (from library **`stdio`**). You may assume that lines in the input file have no more than 200 characters (not including the newline).
- You may also assume that the file contains no more than 1,000 lines.
- Each line read from the file should be converted to a string list using function **`createStrList`** from your solution to 1.5.
- Use the functions you implemented in 2.3 – 2.5.
- Make sure to free all dynamically allocated memory.
- Instead of a file name, the user may specify "-" to indicate that lines should be read from the standard input. For instance, `sortFile - 4 2`.
- Error messages should be printed to standard error.
- See execution examples in next page.

Execution examples for sortFiles:

Not enough arguments:

```
==> ./sortFile myFile
Usage: ./sortFile file key1 [key2 ... ]
```

Providing non-existing file, or file with no read permissions:

```
==> ./sortFile myFile 1
Could not open input file myFile for reading
```

Bad indices:

```
==> ./sortFile great-expectations.txt 1 100 -1
key -1 is not a positive integer
==> ./sortFile great-expectations.txt 1 100 0 -1
key 0 is not a positive integer
==> ./sortFile great-expectations.txt 1 100 12.4
key 12.4 is not a positive integer
```

Good execution:

```
==> ./sortFile great-expectations.txt 4 5 1 8 100 10 > great-expectations.sorted.tst
```

(files `great-expectations.*` can be found in directory `/share/ex_data/ex5/`)

Submission Instructions:

In the directory `~/exercises/ex5/` you should have **exactly 2** files:

- `stringList.c` – file implementing functions for Problems 1 and 2.
- `sortFile.c` – file implementing main for Problem 2.

Important note: use the exact file and directory names specified in the assignment. Linux is case sensitive, and file extensions are part of a file's name.

Before submission, please delete all other items, including binaries. We will **deduct 5pts** for inclusion of additional files. Apply `test_ex 5` to make sure.

All files should be placed on the server (in `~/exercises/ex5/`) at the time of deadline (21:00 @ Jan 19, 2017). Any changes made after that point will count as a late submission. For more information, see the **Homework submission instructions** file on the course website.