System Programming in C – Homework Exercise 5

Publication date: *Thursday, May 30, 2024*
Due date: *Sunday, June 16, 2024* @ 21:00

The purpose of this assignment is to implement a new data type (which we will call *number list*) that represents a list of non-negative integer numbers using a string (**char***). In Problem 1 you are asked to implement several functions for this data type and in Problem 2 you are asked to implement a short program that uses these functions to read numbers as text and perform arithmetic operations on them.

First, we consider each string as a list of words separated by space characters (`' '`). For this purpose, a word is defined as any non-empty sequence of non-space characters. A string is said to be a valid number list if all words in the list are made up of digit characters (`'0'-'9'`), implying that they represent non-negative integer numbers. Two numbers in a number list can be separated by more than one space character.

Examples of strings that are valid number lists:

- The string `"1000"` represents a list with a single number: 1000.
- The string `" 0120  789  00084 "` represents a list with three numbers: 120, 789, and 84. Note that there may be (extra) spaces before the first number, after the last number, and between numbers. Also note that numbers can have leading zeros that do not affect their value (e.g., the first 0 in `0120`).
- The strings `""`, `" "`, and `"  "` all represent empty lists.
- The string `" 0  00 000"` represents a list with three identical numbers (0).

Any string that has a character that is not a digit or space character is not a valid number list.

General coding guidelines:

- Follow the implementation guidelines specified for each function. Not all guidelines are covered by the automatic tests, but they will be checked manually by the graders, so make sure to stick to the guidelines.
- Write clear and readable code. Use appropriate indentation and try to follow the coding style in file `numList.c` as much as possible. You should also add brief documentation for the critical parts of each function's implementation. Your code will be reviewed by the graders and 10 grade points will be allocated for code style.
- Make sure that your code compiles without errors or warnings and passes all the tests specified for each task. Note that Problem 1 and Problem 2 have different compilation instructions.

Problem 1:

The purpose of this question is to implement six functions for basic operations on number lists. Copy the source file `numList.c` from the shared directory `/share/ex_data/ex5/` to your exercise directory `~/exercises/ex5/`. Review the code and documentation written in this file. You will notice that the file contains empty (and faulty) implementations of six functions. In sections 1-6 below you are asked to fix these implementations.

In this assignment you may implement each function using any series of expressions and a combination of operators. Make sure that each implementation is short and simple and write clear documentation explaining your code. In each implementation you may use calls to functions you implemented in previous sections, when appropriate. Do not make use of functions from other standard C libraries (such as `stdio` or `string`). Furthermore, make sure to modify the code of `numList.c` only in the marked places.

Compilation and testing: Testing your solutions requires you to compile your code in `numList.c` with the test code file `/share/ex_data/ex5/test_ex5.c`. This file contains a main function, which runs tests for every function you implement. You are encouraged to examine this code when you implement each function. You "turn on" a specific test by compiling your source file together with `test_ex5.c` and using the `-D TEST_1_<SECTION>` option, where `<SECTION>` is the section number (1-6). For example, to test Problem 1.3, you set `<SECTION>` to `3` and use the following compilation command:

```
==> gcc /share/ex_data/ex5/test_ex5.c numList.c \
          -Wall -D TEST_1_3 \
          -o test_ex5_1_3
```

[We discuss compilation of multi-file programs in detail in Lecture #11]

Compile your code using the flags mentioned above (including `-Wall`) and make sure you do not get any error or warning messages. After you successfully compiled your code, run the resulting executable program (`test_ex5_1_3` in the example above) and compare the output with the expected output provided in file `/share/ex_data/ex5/test_ex5_1_<SECTION>.out` , as follows:

```
==> ./test_ex5_1_3 | diff - /share/ex_data/ex5/test_ex5_1_3.out
```

Run this test after you complete each of the tasks in sections 1-6 below by replacing the '3' above with the appropriate section number.

1. Function **`isValidNumList`** receives a string as parameter (**`str`**) and it returns 1 if this string is a valid number list, and 0 otherwise. See page 1 for the formal definition of a valid number list, as well as the execution examples below. Implement the function without adding any variables other than the **`str`** parameter (use this pointer to scan the input string).

   Execution examples:

   - `isValidNumList("1000")` returns 1
   - `isValidNumList("10o0")` returns 0
   - `isValidNumList(" 0120  789   00084 ")` returns 1
   - `isValidNumList("1.23")` returns 0
   - `isValidNumList("")` returns 1
   - `isValidNumList("-76")` returns 0
   - `isValidNumList("  ")` returns 1
   - `isValidNumList("0120 -13")` returns 0
   - `isValidNumList("0")` returns 1
   - `isValidNumList(" 0 00   0000")` returns 1

2. Function **`numWordsInList`** receives a string as parameter (**`str`**) and it returns the number of words in this string. As specified on page 1, a word in this context is a non-empty sequence of consecutive non-space characters that is flanked on both sides by space characters (or the beginning/end of the string). Implement the function without adding any variables other than the local variable **`numWords`** (which is already defined) and the **`str`** parameter (use this pointer to scan the input string).

   Execution examples:

   - `numWordsInList("1000")` returns 1
   - `numWordsInList("  ")` returns 0
   - `numWordsInList(" 0120 hello  wor.ld ")` returns 3
   - `numWordsInList(" 00 1 02 3 4 50")` returns 6
   - `numWordsInList("0 1 2 3 4 5 6 7 8 9 10")` returns 11
   - `numWordsInList(" 001023450 ")` returns 1
   - `numWordsInList("")` returns 0
   - `numWordsInList("-76")` returns 1
   - `numWordsInList("0120 -13-12")` returns 2
   - `numWordsInList("0")` returns 1
   - `numWordsInList(" 0 00   0000")` returns 3

3. Function **`get1stNumberValue`** receives a string as parameter (**`numList`**) and it returns the numerical value (in type **`double`**) of the first word specified in **`numList`**. If the first word in **`numList`** is not a number (meaning that it has a non-digit sharacter), the function returns -1.0. If the list is empty, the function returns -2.0. The function returns a **`double`** value since the string **`numList`** can represent large integers that are outside the representation range of **`int`**. Implement the function without adding any variables other than the local variable **`value`** (which is already defined) and the **`numList`** parameter (use this pointer to scan the input string).

   Execution examples:

   - **`get1stNumberValue("1000")`** returns 1000.0
   - **`get1stNumberValue("  0 1")`** returns 0.0
   - **`get1stNumberValue("034 .")`** returns 34.0
   - **`get1stNumberValue("34.")`** returns -1.0
   - **`get1stNumberValue("   ")`** returns -2.0
   - **`get1stNumberValue(" 99 88 77 66 55")`** returns 99.0
   - **`get1stNumberValue("-3")`** returns -1.0
   - **`get1stNumberValue("")`** returns -2.0
   - **`get1stNumberValue("98765432109876543210")`**
     returns 9876543210987654**5168**

     Note that the last four digits in this number (highlighted in bold) are not represented accurately. This is because type **`double`** guarantees accurate representation "only" for the 15 most significant decimal digits of a number. Still, unlike the overflow that we get in integer types, this value is very close to the real one. If you get different values for the last four digits, this is fine.

4. Function **`compactNumList`** receives a string as parameter (**`numList`**) and it modifies this string by removing consecutive space characters and leading zeros. After invocation of this function, the string should have no spaces in its beginning or end and exactly one space between two consecutive words. Moreover, it should contain the same words as in the original string, with leading zeros removed (even if the words are not numbers; see examples below). The function returns a pointer to the beginning of the string (this should be the same address pointed to originally by **`numList`**). Note that the compact version of a string is never longer than the original one, so there is sufficient space for writing. Make sure to write the terminating `'\0'` in the appropriate position, and do not modify characters in the input string beyond this position. Try to use a simple linear-time implementation that avoids multiple copying of the same character.

Execution examples:

- **`compactNumList(" 1  034030   23 ")`** modifies string to **`"1 34030 23"`**
- **`compactNumList("1 34030 23")`** does not change the input string
- **`compactNumList("000102  000 ")`** modifies string to **`"102 0"`**
- **`compactNumList("00good 0luck  !0 ")`** modifies string to **`"good luck !0"`**
- **`compactNumList("    ")`** modifies string to `""` (the empty string)

All executions return a pointer to the beginning of the string.

5. Function **`getNextNumberValue`** receives a string as parameter (**`numList`**). When called sequentially, the function returns the value (type **`double`**) of the next word in **`numList`** according to these guidelines:

   - If this is the first time that **`getNextNumberValue`** was invoked during the execution of the program, or if in the last time that **`getNextNumberValue`** was invoked it was called with a different **`numList`** (a different number list means a list that starts at a different address), then the function returns the value of the first word in **`numList`**.
   - If in the last time that **`getNextNumberValue`** was invoked during the execution of the program it was called with the same **`numList`** (starting at the exact same address), then the function returns the value of the word following the word whose value was returned in the previous invocation.
   - If a word is not a sequence of digits, then its value is defined as -1.0.
   - If reached the end of the list, the function returns -2.0 and "resets" itself such that if the next invocation is done with the same **`numList`**, it will return the value of the first word in the list.
   - See execution example on the next page for clarifications.

   Note that implementing this function requires saving information from one call to the next using static local variables. The current implementation has two static variables defined (**`prevNumList`** and **`nextNumInList`**), which should be enough for this task. Avoid defining additional static variables, but you may define non-static variables as you wish. We recommend using **`get1stNumberValue`** from Problem 1.3 to extract the value of the next word.

Execution examples:

We initialize the two **char** arrays **numList1** and **numList2** as follows:

```
numList1[] = "9.05 3  -4  9876 .";
numList2[] = " 1  0230  45.6  78  ";
```

Now, we invoke **getNextNumberValue** sequentially as follows:

- **getNextNumberValue(numList1)** returns -1.0 [1st word in **numList1**]
- **getNextNumberValue(numList1)** returns 3.0 [2nd word in **numList1**]
- **getNextNumberValue(numList2)** returns 1.0 [1st word in **numList2**]
- **getNextNumberValue(numList2)** returns 230.0 [2nd word in **numList2**]
- **getNextNumberValue(numList2)** returns -1.0 [3rd word in **numList2**]
- **getNextNumberValue(numList2)** returns 78.0 [4th word in **numList2**]
- **getNextNumberValue(numList2)** returns -2.0 [end of **numList2**]
- **getNextNumberValue(numList2)** returns 1.0 [1st word in **numList2**]
- **getNextNumberValue(numList2)** returns 230.0 [2nd word in **numList2**]
- **getNextNumberValue(numList1)** returns -1.0 [1st word in **numList1**]
- **getNextNumberValue(numList1)** returns 3.0 [2nd word in **numList1**]
- **getNextNumberValue(numList1)** returns -1.0 [3rd word in **numList1**]
- **getNextNumberValue(numList1)** returns 9876.0 [4th word in **numList1**]
- **getNextNumberValue(numList1)** returns -1.0 [5th word in **numList1**]
- **getNextNumberValue(numList1)** returns -2.0 [end of **numList1**]
- **getNextNumberValue(numList1)** returns -1.0 [1st word in **numList1**]
- **getNextNumberValue(numList1)** returns 3.0 [2nd word in **numList1**]
- **getNextNumberValue(numList1 + 2)** returns 5.0
  This is because **numList1 + 2** is a different string than **numList1**, and its first word is **"05"**.
- **getNextNumberValue(numList1 + 2)** returns 3.0

6. Function **performOperation** receives as parameter a string (**numList**) representing a number list, and a character (**op**) representing an arithmetic operation (`'+'` or `'*'`). It returns the value (type **double**) obtained by applying the appropriate operation on all numbers in the list according to these guidelines:

   - If **op** is not `'+'` or `'*'`, then the function returns -1.0.
   - If one of the words in **numList** is not a number (sequence of digit characters), then the function returns -1.0.
   - Otherwise, if **op** is `'+'`, the function returns the sum of all the numbers in the list, and if it is `'*'`, the function returns the product of all the numbers.
   - If **numList** is an empty list (no words), then the function returns 0.0 when **op** is `'+'` and it returns 1.0 when **op** is `'*'`.

   Use function **getNextNumberValue** from Problem 1.5 to extract the values of all numbers in the list. Make sure that your function behaves appropriately even when function **getNextNumberValue** is called with the same **numList** argument right before or right after a call to **performOperation**. This requires an appropriate "resetting" of the static local variables of **getNextNumberValue**. Think how you can do this without having direct access to these variables.

   Execution examples:

   - **performOperation**`("1 2 3 4 5  6 ", '+')` returns 21.0
   - **performOperation**`("1 2 3 4 5  6 ", '*')` returns 720.0
   - **performOperation**`("1 2 3 4 5  6 ", '-')` returns -1.0
   - **performOperation**`("1", 'X')` returns -1.0
   - **performOperation**`("1", '+')` returns 1.0
   - **performOperation**`("1", '*')` returns 1.0
   - **performOperation**`("", '+')` returns 0.0
   - **performOperation**`("  ", '*')` returns 1.0
   - **performOperation**`("1 2 3 4 5 six 7", '*')` returns -1.0
   - **performOperation**`(" 1  034030   23 ", '+')` returns 34054.0
   - **performOperation**`(" 1  034030   23 ", '*')` returns 782690.0

Final testing for Problem 1: After you finished implementing all six functions, and you individually tested each function using the guidelines specified on page 2, you should ensure that your code provides the expected output for all functions by executing the test script `/share/ex_data/ex4/test_ex5.1` from the directory containing your `numList.c` source file. This script produces a detailed error report to help you debug your code.

Problem 2:

The purpose of this question is to implement a simple calculator program that uses the number list data type to perform simple arithmetic operations on numbers it reads from the standard input.

Program description:

- The program reads characters from the standard input and enters them sequentially into a number list (see implementation guidelines and execution examples below).
- When the program reads the characters `'+'` or `'*'` it performs the appropriate arithmetic operation (sum or product) on the current number list and proceeds according to the following three cases:
  - If the number list is empty (contains no words), then the program halts with status 0 (correct termination).
  - If the number list contains a word that is not a valid number (meaning that it contains a non-digit character), then the program prints the following message and halts with status 1:

    `Aborting because word #<N> in list is not a valid number`

    Where `<N>` is the index of the first word in the list that is not a valid number (1 for the first word, 2 for the second word, etc.).
  - If all words in the number list are valid numbers, then the program prints the result of the operation in the following format:

    `<NUM1><OP><NUM2><OP> ... <OP><NUMn>=<RES>`

    Where `<NUMi>` is the i^{th} number in the list without leading zeros, `<OP>` is the operator character (`'+'` or `'*'`), and `<RES>` is the result of the operation printed without a decimal point.

    In this case, the program empties the number list and continues reading characters.
- See the execution examples on pages 10-11 for specific demonstrations of the expected behavior of the program.

Implementation guidelines: The program should be implemented in a source file named `simpleCalc.c`. Copy the source file `simpleCalc.c` from the shared directory `/share/ex_data/ex5/` to your exercise directory `~/exercises/ex5/`. The initial version of `simpleCalc.c` contains `#include` directives for `<stdio.h>` and `<string.h>` (that will enable you to call **`printf`**, **`scanf`**, and library functions for strings), a `#define` directive for a symbolic constant (`MAX_CHARS_IN_LIST`; see below) and declarations of the six functions that you implemented in Problem 1. In the marked location in the file, you should write a definition for a main function that implements the program described above and follow the detailed guidelines on the next page.

- Your code may contain calls to functions that you implemented in Problem 1, as well as the **`printf`** and **`scanf`** functions, and (optionally) string library functions.
- Do not add any `#include` directives to `simpleCalc.c`.
- Read characters from the input stream one by one using a call to **`scanf`** with conversion specifier `%c`.
- Maintain a number list in a local array defined in function **`main`**. You may assume that the number list contains at most 100 characters. However, instead of using the literal 100, you should use the symbolic constant `MAX_CHARS_IN_LIST`, which is defined in `simpleCalc.c`. Note that the array that holds the number list should have space for the terminating `'\0'` after the number list.
- If the input stream contains more than 100 characters between two consecutive operator characters (`'+'` or `'*'`), then the program prints the following message and halts with status 2:

  ```
  Aborting because number list contains more than 100 characters
  ```

- Use function **`numWordsInList`** to check whether the number list is empty.
- Use function **`performOperation`** to compute the result of an operation.
- Use function **`compactNumList`** to compact the number list before printing an operation.
- Use sequential invocation of **`getNextNumberValue`** to find the index of the word that is not a number (when aborting with status 1).
- This program can be implemented using a relatively short **`main`**, so there is no need to implement additional functions. However, if you do choose to implement additional functions and call them from **`main`**, then declare these functions above the definition of **`main`** and define these functions below the definition of **`main`** (as we show in Lecture #6).
- Write clear and readable code. Use appropriate indentation and try to follow the style of `numList.c` as much as possible. You should also add brief documentation for the critical parts of your implementation. Keep in mind that your code is reviewed by the graders and 5 points will be allocated for code style.

Compilation and testing: Testing your solution requires you to compile the source files `numList.c` and `simpleCalc.c` together. This should be done using the following compilation command:

```
==> gcc -Wall simpleCalc.c numList.c -o simpleCalc
```

[We discuss compilation of multi-file programs in detail in Lecture #11]

Compile your code using the flags mentioned above and make sure you do not get any error or warning messages. If compilation is successful, it creates an executable program called `simpleCalc`, which you can run and test on basic execution examples provided below, as well as additional examples.

To help you test your code, we provide a working executable in `/share/ex_data/ex5/simpleCalc`. You should prepare a set of inputs and compare your output with the output of our program. You may use the execution examples from the next page as a base set of inputs.

Finally, when you are convinced that your program works well, execute the testing script `/share/ex_data/ex5/test_ex5.2` from the directory containing your `simpleCalc.c` and `numList.c` source files. The script produces a detailed error report that may help you find additional bugs in your code.

Execution examples:

```
==> echo "1 002 03  4 5 6 7+  8 9 10 11 12 13 14 15 16 *  *" | ./simpleCalc
1+2+3+4+5+6+7=28
8*9*10*11*12*13*14*15*16=4151347200

==> echo $?
0
```

(← exit status)

Explanation: the input stream consists of two valid number lists and an empty number list, which halts the program (without an error message).

```
==> echo " 0000 1 2 3  + 88 44 hello world * 1 2 3 + . . ." | ./simpleCalc
0+1+2+3=6
Aborting because word #3 in list is not a valid number

==> echo $?
1
```

Explanation: the 3rd and 4th words of the second list are not numbers ("hello" and "world"), which causes the program to abort with an error message.

```
==> echo "   + 42 422 84 *" | ./simpleCalc

==> echo $?
0
```

Explanation: the first number list (before the `'+'`) is empty, which halts the program without printing anything.

```
==> cat /share/ex_data/ex5/long_input.txt | ./simpleCalc
Aborting because number list contains more than 100 characters

==> echo $?
2
```

Explanation: file `long_input.txt` contains a single line with numbers 1…50 and two operator characters: "`1 2 … 50  * +`". The length of the number list exceeds 100 characters, so the program aborts with an error message.

```
==> tail -c104 /share/ex_data/ex5/long_input.txt | ./simpleCalc
18*19*20*21*22*23*24*25*26*27*28*29*30*31*32*33*34*35*36*37*38*39*40*41*42*43*44
*45*46*47*48*49*50=85507922966297847076363233144142401029507777560576
```

Explanation: the `tail -c104` command prints the last 104 characters of the file, which consist of the number list "`18 19 … 50  `" (100 characters) followed by "`* +`" (three characters) and a newline character. Since the number list contains exactly 100 characters, it is processed and the program halts without an error message.

```
==> tail -c105 /share/ex_data/ex5/long_input.txt | ./simpleCalc
Aborting because number list contains more than 100 characters
```

Explanation: similar to the previous example, but here the number list has 101 characters, which causes the program to abort with an error message.

Submission Instructions:

1. After you validated and tested your solution, make sure that your `~/exercises/ex5/` directory contains the following C source files, which include your implementation:
   - `numList.c`
   - `simpleCalc.c`
2. your `~/exercises/ex5/` directory should also contain a `PARTNER` file with the user id of the non-submitting partner. The non-submitting partner should also add a `PARTNER` file containing the user id of the submitting partner.
3. Check your solution by running **`check_ex ex5`**. The script should be executed from the account of the submitting partner, and it may be run from any directory. Clean execution of this script guarantees you 80% of the assignment's grade.
4. Once you are satisfied with your solution, you may submit it by running **`submit_ex ex5`**. The script should be executed from the account of the submitting partner, and it may be run from any directory. You may modify your submission any time before the deadline (**16/6 @ 21:00**) by running **`submit_ex ex5 -o`** from any location.
5. For more information on the submission process, see the **Homework submission instructions** file on the course website.