System Programming in C – Homework Exercise 6

Publication date: *Thursday, June 13, 2024*
Due date: *Sunday, July 7, 2024* @ 21:00

In this assignment you will implement a **grade list** "class". This new data type enables maintaining a list of course participants and their grades. In Problem 1 you will implement seven functions for this "class", including a procedure that sorts participants by grades using the *merge-sort* algorithm (that you saw in the Data Structures course). In Problem 2 you will use a grade list object to read student grades from a file and print them sorted.

The grade list "class" uses a linked list to hold information about course participants. Each node in this list is represented by following two types:

```
/*** pointer to grade node struct defined below ***/
typedef struct GradeNode_st* GradeNode;

/*** grade node struct ***/
struct GradeNode_st {
   char* firstName;   // first name of student
   char* lastName;    // last name of student
   char grade;        // grade held as an integer [0,100]
   GradeNode next;    // pointer to next grade node
};
```

Each grade node holds two strings for the first and last name of the course participant, as well as a grade. Since the course grade is an integer in the range [0,100], we hold it using a **`char`** (and not an **`int`**, which takes up more space). The two strings for the participant's name are physically allocated outside of the structure defined for the grade node (see Problem 1.1). Finally, as with the simple linked list that we see in Lecture #11, each grade node holds a pointer to the next node in the list. The **`next`** field of the last node in the list is set to NULL. A grade list is represented by a pointer to the first grade node in the list (type **`GradeNode`**), and an empty grade list is represented by NULL.

An important feature of the grade list "class" is the ability to sort participants based on their grades. For this purpose, we need to define a complete order between all grade nodes. This is done by sorting grade nodes first based on grades, then (when the grades are identical) by last names, and then (when grades and last names are identical) by first names. This is all defined by the "member" functions of this "class".

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.
- This time, we do not provide you with template files. You will write your code in files that you create based on the specification for each Problem. Write clear and readable code.
- Use appropriate indentation and try to follow the coding style we demonstrated in HW assignment #4 and #5 (see, e.g., `numList.c` from HW #5). 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.

The virtual heap:

We implemented a specialized set of functions for you to use in all dynamic memory allocations that you are required to use in this assignment: **`ourMalloc()`** and **`ourFree()`**. These functions should be applied exactly like their counterparts from the standard library `stdlib` (**`malloc()`** and **`free()`**). You can see their declarations in the header file `/share/ex_data/ex6/virtualHeap.h` and their implementation in `/share/ex_data/ex6/virtualHeap.c`. If you review the implementation, you will notice that these functions simply call their standard counterparts, while keeping track of the status of the allocated memory using global variables. This enables us (and you) to ensure that your memory allocation is correct and without memory leaks. **Your implementation should use our virtual heap functions instead of the memory allocation functions in `stdlib`.**

Calling functions implemented in other source files:

In both problems of this assignment, you will be required to make use of functions implemented in other source files. This includes functions from standard libraries (`stdio` and `string`), functions that we implemented for you in the virtual heap, and in Problem 2 also functions that you implemented in Problem 1. In Lecture #11 we discuss the concept of code modules and how `gcc` compiles multi-file programs. For the purpose of this assignment, simply make sure that your source files in Problem 1 and Problem 2 contain the following include directives at the top:

```
#include "/share/ex_data/ex6/gradeList.h"
#include "/share/ex_data/ex6/virtualHeap.h"
#include <stdio.h>
#include <string.h>
```

These directives "import" the declarations of functions in these libraries into your source file, enabling the compiler to know their interface.

Problem 1:

The purpose of this question is to implement functions for grade nodes and lists, as specified on Page 1. The interface for this "class" is specified in the header file `/share/ex_data/ex6/gradeList.h`. This header file contains a definition of the two types described on Page 1 and the declarations of seven functions. In sections 1-7 below you are required to implement these functions in a source file `gradeList.c` in your `~/exercises/ex6/` directory. Write your source code from scratch, following the coding and style guidelines above.

Compilation and testing: As in previous assignments, testing your implementation requires you to compile your code in `gradeList.c` with the test code file `/share/ex_data/ex6/test/test_ex6.c`. You "turn on" a specific test by compiling your source file together with `test_ex6.c` and using the `-D TEST_1_<SECTION>` option, where `<SECTION>` is the section number. Because your code makes use of the virtual heap, you should compile it together with the `virtualHeap.c` source file. The complete compilation line for the test to run after you implemented Problem 1.3 is:

```
==> gcc /share/ex_data/ex6/test/test_ex6.c gradeList.c \
        /share/ex_data/ex6/virtualHeap.c \
        -Wall -D TEST_1_3 \
        -o test_ex6_1_3
```

After you successfully compile your code, run the resulting executable program and compare the output with the expected output, which we provide:

```
==> ./test_ex6_1_3 > my_test_ex6_1_3.out
==> diff my_test_ex6_1_3.out \
        /share/ex_data/ex6/test/test_ex6_1_3.out
```

1. Implement the constructor function **`newGradeNode`**, which creates a new grade node object (grade list with a single node) according to the guidelines below:
   - Function signature: **`GradeNode newGradeNode(const char* firstName, const char* lastName, int grade)`**
   - If **`grade`** is not in the range [0,100], the function does nothing and returns NULL. No need to check the other two parameters (**`firstName`** and **`lastName`**).
   - Otherwise, the function allocates the space on the heap required for holding the grade node object. Note that this includes the structure as well as copies of the two strings **`firstName`** and **`lastName`**, which are stored outside of the structure.
   - Use **`ourMalloc`** from the virtual heap to allocate memory.
   - Allocate the exact number of bytes required to hold the new object.
   - If memory allocation fails for some reason, the function returns NULL. If some allocation succeeds and some fails, you should free all allocated memory before you return NULL (use **`ourFree`** from the virtual heap).

- Before returning the grade node object, the function sets all structure fields to their appropriate values, including copying **`firstName`** and **`lastName`** to the newly allocated space for these strings.

For **execution examples**, you should review the expected output of our tests in file `/share/ex_data/ex6/test_ex6_1_1.out`.

2. Implement the destructor function **`freeGradeList`**, which frees all space allocated by a grade list starting from a given node according to the guidelines below:
   - Function signature: **`void freeGradeList(GradeNode gradeList)`**
   - If **`gradeList`** is an empty grade list (NULL), the function does nothing.
   - Otherwise, the function frees all memory allocated by all nodes in the given grade list. This includes space allocated by the structures and the strings for the first and last names.
   - Use **`ourFree`** from the virtual heap when freeing memory.
   - The function does not change the contents of the memory before or after freeing it (there is no need for that).
   - Hint: there is a simple recursive implementation for this function.

   For **execution examples**, you should review the expected output of our tests in file `/share/ex_data/ex6/test_ex6_1_2.out`.

3. Implement the function **`largerGradeNode`**, which compares two grade node objects and returns the "larger" one according to the guidelines below:
   - Function signature: **`GradeNode largerGradeNode(GradeNode gradeNode1, GradeNode gradeNode2)`**
   - If the **`grade`** fields of the two grade nodes are different, then the function returns the node with the larger **`grade`**.
   - If the **`grade`** fields of the two grade nodes are identical but their **`lastName`** fields contain different strings, then the function returns the node whose **`lastName`** string has greater lexicographical value.
   - If the **`grade`** and **`lastName`** fields of the two grade nodes are identical but their **`firstName`** fields contain different strings, then the function returns the node whose **`firstName`** string has greater lexicographical value.
   - If the **`grade`**, **`lastName`**, and **`firstName`** fields of the two grade nodes are identical, then the function returns **`gradeNode1`** (arbitrary choice).
   - Use the standard library function **`strcmp`** to determine the lexicographic order of two strings. See complete documentation here.
   - The function does not allocate (or free) any space. It simply returns one of the objects it received as arguments.

   For **execution examples**, you should review the expected output of our tests in file `/share/ex_data/ex6/test_ex6_1_3.out`.

4. Implement the function **`printGradeList`**, which prints a grade list according to the guidelines below:
   - Function signature: **`void printGradeList(GradeNode gradeList)`**
   - If **`gradeList`** is an empty list, the function simply prints the following line:
     `Empty grade list`
   - Otherwise, the function prints the following line for every grade node in the list (in order from first node to last):
     `<firstName><TAB><lastName><TAB><grade>`
     Where `<firstName>`, `<lastName>`, and `<grade>` are the fields of the grade node and `<TAB>` is the tab character (`'\t'`).
   - When the list is not empty, then after printing all the grade nodes, the function prints the following summary line:
     `Average of <N> grades is <AVG>`
     Where `<N>` is the number of nodes in the list and `<AVG>` is the average grade printed with precision of one digit after the decimal point (averages of integers may result in a non-integer value).

   For **execution examples**, you should review the expected output of our tests in file `/share/ex_data/ex6/test_ex6_1_4.out`.

5. Implement the function **`appendGradeNode`**, which appends a single grade node to a grade list according to the guidelines below:
   - Function signature: **`GradeNode appendGradeNode(GradeNode gradeList, GradeNode gradeNode)`**
   - If **`gradeNode`** is not a single grade node (meaning that it is not a grade list of length 1), the function does nothing and returns the original **`gradeList`**.
   - Otherwise, the function adds **`gradeNode`** to the end of **`gradeList`** and returns the modified grade list.
   - The function does not allocate (or free) any space. It simply reconnects the existing grade nodes.

   For **execution examples**, you should review the expected output of our tests in file `/share/ex_data/ex6/test_ex6_1_5.out`.

The final two functions of the grade list "class" implement *merge-sort*, which efficiently sorts a list of elements (you discuss this algorithm in Data Structures). Recall that *merge-sort* is based on the following recursive procedure:

- Split the list into two separate lists of equal length (if the list contains an odd number of elements, then one list will have one less node).
- Recursively apply *merge-sort* to each half list separately.
- Merge the two sorted (half) lists.

The efficiency of this algorithm stems from the fact that merging two sorted lists can be done in a single scan of both lists. In Problem 1.6 you will implement this efficient merging procedure, and in Problem 1.7 you will use this merging procedure to implement the merge-sort algorithm.

6. Implement the function **`mergeSortedGradeLists`** which merges two sorted grade lists according to the guidelines below:
   - Function signature: **`GradeNode mergeSortedGradeLists(GradeNode gradeList1, GradeNode gradeList2)`**
   - The function combines the grade nodes in the two lists **`gradeList1`** and **`gradeList2`** into one grade list whose nodes are sorted from "small" to "large" according to the definition of "large" implied by function **`largerGradeNode`** from Problem 1.3. Your implementation should call this function to determine node order.
   - The function returns the merged list (the first of the grade nodes in that list).
   - The function does not allocate (or free) any space. It simply reconnects the existing grade nodes.
   - Your implementation may (and should) assume that **`gradeList1`** and **`gradeList2`** are already sorted. If they are not, then the function returns a merged list that may not be sorted.
   - There is a relatively simple recursive implementation for this function.

     Hint: find the "smallest" node, then call the function recursively to merge the remaining nodes and attach the merged list to the "smallest" node.

   For **execution examples**, you should review the expected output of our tests in file `/share/ex_data/ex6/test_ex6_1_6.out`.

7. Implement the function **`mergeSortGradeList`** which sorts a given grade list according to the guidelines below:
   - Function signature: **`GradeNode mergeSortGradeList(GradeNode gradeList)`**
   - The function sorts the nodes in **`gradeList`** from "small" to "large" according to the definition of "large" implied by function **`largerGradeNode`** from Problem 1.3.
   - The function returns the sorted list (the first of the grade nodes in that list).
   - The function does not allocate (or free) any space. It simply reconnects the existing grade nodes.
   - Implement the (recursive) *merge-sort* procedure described in the previous page, and call **`mergeSortedGradeLists`** from Problem 1.6 to merge the two sorted (half) lists.

   For **execution examples**, you should review the expected output of our tests in file `/share/ex_data/ex6/test_ex6_1_7.out`.

Final testing for Problem 1: After you finished all the implementations above, and you manually ran the individual tests for each item, you should ensure that your code provides the expected output for all functions by executing the test script `/share/ex_data/ex6/test_ex6.1` from the directory containing your `gradeList.c` source file. This script produces a detailed error report.

Problem 2:

The purpose of this question is to implement a program called `processGrades`, which reads a list of course participants and their grades from a file, and then it prints a summary to the standard output. Implement the program in a source file named `processGrades.c` in your `~/exercises/ex6/` directory. As in Problem 1, we do not give you a template for this file, so write it from scratch following the coding and style guidelines on page 2, and the specific instructions given below.

Program usage:

The program should be executed with one input argument specifying the input file:

```
==>  ./processGrades <inFile>
```

Checking input arguments:

The input argument line should be checked for the cases listed below (in that order):

- If no arguments are specified, then the program prints the following error message to the standard error:

  ```
  Error: no file name provided
  ```

  The program then aborts with exit status 1.

- If more than one argument is specified, then the program prints the following error message to the standard error:

  ```
  Error: expecting only one input argument
  ```

  The program then aborts with exit status 2.

- If the (only) input argument does not correspond to a path of a readable file, then the program prints the following error message to the standard error:

  ```
  Error: <inFile> cannot be open for reading
  ```
  (where `<inFile>` is the input argument)

  The program then aborts with exit status 3.

Reading course participants' names and grades from file:

- Maintain a grade list that holds all information read from the file.
- Read the text from the input file line by line using the **`fgets()`** function from the standard library `stdio`. Use a local character array for this purpose and set its size according to the assumption that each line in the input file has no more than 200 characters (not including the newline character). Use a `#define` directive to define a symbolic constant representing the maximum line length. You should not assume any limit on the number of lines in the input file.
- Read the first three words in the line as the first name, last name, and grade of a course participant. A word here is a consecutive sequence of non-space

characters that is flanked by space characters (or the beginning/end of the line). There may be space characters before the first word and more than one space between words in the line.

- If a line has fewer than three words, or if its third word is not a number in [0-100], the line is considered to have a faulty format, and the program prints the following error message to the standard error:

  `Line <N> has faulty format`
  (where `<N>` is the line number in the input file)

  The program then continues to read the file.
- Otherwise, the program constructs a new grade node object based on the information read from the line and adds this grade node to the end of the grade list (using **`appendGradeNode`**). The text in the line needs to be processed to prepare the three arguments for **`newGradeNode`**. Try to do this without using additional character arrays (other than the one used to hold the line).
- If at any point, memory allocation failed, then the program prints the following error message to the standard error:

  `Out of memory after reading <N> lines`
  (where `<N>` is the line number in the input file where allocation failed)

  The program then aborts with exit status 11.

Final steps:

- After the file has been read, the grade list should be sorted (using **`mergeSortGradeList`**) and printed (using **`printGradeList`**).
- If there were faulty lines in the file (see above), the program halts with exit status 4. Otherwise, it halts with exit status 0. In both cases, no additional message is printed to the standard error or output.
- Before the program halts (in any of the cases listed above) it should make sure to free all dynamically allocated space, including the space taken by the FILE* object used to read the input file.

Other implementation notes:

- Your implementation should use functions that you implemented in Problem 1. The declarations of these functions are specified in the header file `gradeList.h`, which you should include in `processGrades.c`, (see instructions on Page 3).
- Your program may also call functions from the standard libraries `stdio` and `string`, but not any other library (e.g., `stdlib`). This program can be implemented without directly calling functions from the virtual heap, so including `virtualHeap.h` is optional.
- If you implement functions other than **`main`**, you should declare them at the top of the source file and define them below **`main`**.

Compilation and testing:

Testing your solution requires you to compile your code in `processGrades.c` together with a with a working implementation of functions declared in `gradeList.h` and `virtualHeap.h`. Use the following compilation command for this purpose:

```
==> gcc -Wall /share/ex_data/ex6/virtualHeap.c       \
          gradeList.c  processGrades.c  -o processGrades
```

To help you test your code, we provide a working executable in `/share/ex_data/ex6/test/processGrades`. You can compare your program to this one on various inputs. You may use the execution examples below as a base set of inputs. For more rigorous validation, you may copy over the sample file `grade-list.txt` to your exercise directory, modify it in various ways, and execute your program and our program on the modified file. You should examine the output, error messages, and exit status.

Finally, when you are convinced that your program works well, execute the testing script `/share/ex_data/ex6/test_ex6.2` from the directory containing your `processGrades.c` file. The script produces a detailed error report that may help you find additional bugs in your code.

Execution examples:

Execution on sample file `grade-list1.txt`, which contains ten lines, all with valid format:

```
John   Smith  72
Israel Israeli 84
Homer  Simpson 00000 Homer didn't do all that well
 Eden    Golan 84    Fifth place is pretty good
Johny Smithers 72    Johny appealed his grade
 Yoni  Israeli 84
Israel  Smithy 72    Didn't submit last assignment
  Jane     Doe 84
Mickey Mouse10 0000000072
  Mark Twain  100
```

```
==> ./processGrades /share/ex_data/ex6/grade-list1.txt > sorted-grades.txt

==> echo $?
0

==> cat sorted-grades.txt
Homer    Simpson   0
Mickey   Mouse10   72
John     Smith     72
Johny    Smithers  72
Israel   Smithy    72
Jane     Doe       84
Eden     Golan     84
Israel   Israeli   84
Yoni     Israeli   84
Mark     Twain     100
Average of 10 grades is 72.4
```

Since all lines in the input file are valid, then nothing is printed to the standard error and the exit status is 0.

Execution on sample file `grade-list2.txt`, which contains 12 lines, six of which have a faulty format (explained in red below):

```
John      Smith      67
Israel    Israeli    93
                                  ← no words in line
Eden      Golan      82
Johny     Smither    81   Johny appealed his grade
Lebron    James    1000           ← grade > 100
Johny____Israeli    81   Johny be good  ← 3rd word (Johnny) is not number
Israel    Smith      67
JaneDoe              93           ← only two words in line
Marcus    Marcus   84.5           ← 3rd word (84.5) is not integer number
Mickey    Mouse10    73
MarkTwain100                      ← only one word in line
```

```
==> ./processGrades /share/ex_data/ex6/grade-list2.txt > sorted-grades.txt
Line 3 has faulty format
Line 6 has faulty format
Line 7 has faulty format
Line 9 has faulty format
Line 10 has faulty format
Line 12 has faulty format

==> echo $?
4

==> cat sorted-grades.txt
Israel            Smith  67
John              Smith  67
Mickey            Mouse10     73
Johny             Smither     81
Eden              Golan  82
Israel            Israeli     93
Average of 6 grades is 77.2
```

Lines with faulty format are indicated in error messages printed to the standard error (which is not redirected), and the exit status is 4.

Below are three executions with invalid input arguments. In all cases, the output file `out.txt` is empty because the error message is printed to the standard error.

No input arguments:

```
==> ./processGrades > out.txt
Error: no file name provided
==> echo $?
1
==> cat out.txt
==>
```

More than one input arguments:

```
==> ./processGrades /share/ex_data/ex6/grade-list1.txt myFile > out.txt
Error: expecting only one input argument
==> echo $?
2
==> cat out.txt
==>
```

Input argument is not a path of a readable file:

```
==> ./processGrades /share/no-dir > out.txt
Error: /share/no-dir cannot be open for reading
==> echo $?
3
==> cat out.txt
==>
```

Submission Instructions:

1. After you validated and tested your solution, make sure that your `~/exercises/ex6/` directory contains the following C source files:
   - `gradeList.c`
   - `processGrades.c`
2. your `~/exercises/ex6/` 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 ex6`**. 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 ex6`**. 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 (**7/7 @ 21:00**) by running **`submit_ex ex6 -o`** from any location.
5. For more information on the submission process, see the **Homework submission instructions** file on the course website.