System Programming in C – Homework Exercise 6

Publication date: *Friday, June 26, 2020*
Due date: *Sunday, July 12, 2020 @ 21:00*

General overview: The purpose of this assignment is to implement a program that **generates expense reports** for expenses listed in a text file. In Problem 1, you will implement an ExpenseReport object by defining a structure type and a series of functions (methods) in file `expenseReport.c`. In Problem 2, you will write a separate source file named `processExpenses.c` that use ExpenseReport objects to process expense items listed in a file.

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 ☺.

Including other modules / standard libraries in your source files:

- In both problems, you are required to include two header files that we provide in `/share/ex_data/ex6/`:
  - `expenseReport.h` – header file for the expense report module that you implement in Problem 1.
  - `virtualHeap.h` – header file for the virtual heap module that we implemented for you to use for all memory allocation functions.

  All dynamic memory allocation should be done using our virtual heap!
- In addition to these two modules, you may also include the two standard libraries `stdio` and `string`. Do not include any other libraries, especially not the standard library `stdlib`.
- Use the following include directives to include these header files:

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

  (Note that there is no need to include the virtual heap header in Problem 2)

Coding and compilation 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 style of source files in previous assignments (such as `numberString.c` from HW #5). You should also add brief documentation for the critical parts of each function's implementation. Keep in mind that your code is reviewed by the graders and 10 grade points will be allocated for code style.
- Your code should compile without errors or warnings. Note that the different problems have slightly different compilation instructions.

Problem 1:

The purpose of this question is to implement an ExpenseReport object for holding an expense report for a given period of time. An expense report holds the following information:

- Total amount of all expenses specified in NIS (₪) as a positive **`double`** value.
- Start date and end date of the period for which expenses are specified. Each of the two dates is held by an 8-character array in the following format: `YYYYMMDD`. For example, the date of June 25, 2020 is specified using the eight characters `20200625`. This format was chosen because it associates chorological order of dates with lexicographical order of the date string, or the numeric value of the 8-digit integer it represents (see Problem 1.4).
- Detailed description of expenses specified as a free-text string. The different expense items in the list are separated by the newline character (`'\n'`), such that when the description is printed, they appear in different lines.

Two examples of valid expense reports:

Report #1: three expenses from 27/12/99 to 13/1/00 amounting to 1284 NIS.

- Total amount: `1284.0`
- Start date: `19991227`
- End date: `20000113`
- Description: `"flight to NY\nHotel 1+1 night off\n Taxi"`

Report #2: a single expense from 31/5/11 amounting to 34.56 NIS.

- Total amount: `34.56`
- Start date: `20110531`
- End date: `20110531`
- Description: `"lunch with visitors, 20% off"`

Coding instructions: In the six tasks below, you are required to implement a structure type for the ExpenseReport object and five (method) functions for manipulating it. The function declarations and public object data type are specified in the header file `/share/ex_data/ex6/expenseReport.h`, and you are required to write your implementation for the tasks below in the source file `expenseReport.c` in your exercise directory. Follow the general coding guidelines on page 1.

Testing and validation: We provide two series of tests (1A and 1B) for you to run to test your code after you solved Problems 1.3 and 1.6. Note that there isn't a test after every task. If you wish to see execution examples for reference when you are implementing your functions, we encourage you to examine the expected test outcomes in files `test_ex6_1A.out` and `test_ex6_1B.out` in the shared directory `/share/ex_data/ex6/`). **We do not specify detailed execution examples in this document**.

1. Define the structure type **`struct ExpenseReport`**, which constitutes the basis of the ExpenseReport object. The structure should have four fields representing the total amount, start date, end date, and description, as specified on the previous page. Note that this new type is "private" since it is defined in the source file and not the header file. You may thus choose any name for the fields, since they are not part of the module's interface. The public type used to refer to the ExpenseReport object is the pointer type **`ExpenseReport`** defined in the header file. Follow these guidelines to ensure the validity of your structure type:
   - Define the four fields of the structure in the order they are specified in the two examples on the previous page. This is to ensure that you get a structure of the expected size (see **Test 1A** below).
   - The field holding the total amount should be of type **`double`**.
   - The start and end dates should be held in two 8-character arrays within the structure. Note that since we know the size of each array (8), we can include them within the structure. Also, we do not need to terminate the date strings with `'\0'` characters.
   - Since we do not know the length of the description string in advance, the characters of this string are not kept inside the structure, and the structure accesses them using a free pointer (similar to the vector example we saw in Lecture #11).

2. Implement the constructor function **`newExpenseItem`**, which creates a new ExpenseReport object corresponding to a single expense item. Follow these guidelines:
   - Function signature:
     ```
     ExpenseReport newExpenseItem ( double amount,
                                    const char* date, const char* description )
     ```
   - In the following cases, the function should do nothing and return NULL:
     - If **`amount`** is negative or 0.
     - If not all of the eight characters in **`date`** are digits.
     - If the year coded in **`date`** is not in the range 1950 – 2020.
     - If the month coded in **`date`** is not in the range 01 – 12.
     - If the day coded in **`date`** is not in the range 01 – 31. You do not have to check compatibility of the day and month. For example, date `20010231` (February 31st, 2001) is considered a valid date for this purpose.
     - If the **`description`** string contains a newline character (`'\n'`), indicating that it consists of more than one item. You may find the function **`strchr`** from the standard library `string` useful for this purpose.
   - You should not assume that the date array provided as input is terminated by a `'\0'` character. You simply access its first eight elements.
   - If the parameters meet all the conditions above, then the function should allocate the appropriate space for the ExpenseReport structure and the description string. Recall that the string characters are not part of the

structure, so a separate memory block must be allocated for them. The function should allocate the exact space required, as allocating extra space will lead to faulty tests.

- Use the function **`ourMalloc`** defined in the virtual heap module in all your allocations. This function has the same signature as the standard **`malloc`** function. If allocation failed, your function should return NULL.
- Make sure allocation succeeds before you access allocated space. If some allocation succeeds and some fails, you should free all allocated memory before you return NULL (use the **`ourFree`** function from the virtual heap to do this).
- If the parameters meet all the conditions above and all allocation succeeds, then set all structure fields to their appropriate values. Copy over the **`date`** and **`description`** strings to the appropriate allocated space. Note that the start and end dates should both hold the same date. You may find the functions **`strcpy`** and **`strncpy`** from the standard library `string` useful for this purpose.
- Return the new ExpenseReport object.

3. Implement the destructor function **`freeExpenseReport`**, which frees all space allocated by a given ExpenseReport object. Follow these guidelines:
   - Function signature: **`void freeExpenseReport(ExpenseReport expenseRep)`**
   - Use the function **`ourFree`** from the virtual heap module to free the space allocated by the structure and the description string (see the allocation instructions for **`newExpenseItem`**).
   - Your function should not change the content of the ExpenseReport members before freeing them (there is no need for that).
   - Make sure to free the structure and string in the correct order to avoid accessing freed memory in the heap.

Manual inspection: Your function should contain two calls to **`ourFree`** (in the correct order) and a void **`return`** statement, and no additional statements.

Validation: Test 1A checks the size of the structure type you defined in Problem 1.1 as well as your application of dynamic memory allocation and de-allocation in Problems 1.2 and 1.3. For information on the test, containing execution examples and expected output, please examine `/share/ex_data/ex6/test_ex6_1A.out`.

To run Test 1A, compile your code with the test code file `/share/ex_data/ex6/test_ex6_1A.c` as follows:

```
==> gcc  expenseReport.c  /share/ex_data/ex6/test_ex6_1A.c \
         /share/ex_data/ex6/virtualHeap.c  -Wall -o test_ex6_1A
```

After your source has been **successfully compiled** (no errors or warning), you should run the program `test_ex6_1A` and compare the output (using `diff`) with the expected output provided in `/share/ex_data/ex6/test_ex6_1A.out`.

4. Implement the function **`mergeExpenseReports`** which merges two expense reports. Follow these guidelines:
   - Function signature: **`ExpenseReport mergeExpenseReports(`**
     `ExpenseReport expenseRep1, ExpenseReport expenseRep2)`
   - The function should merge the expense report represented by **`expenseRep2`** into the expense report represented by **`expenseRep1`**.
   - To achieve this, the function modifies the object **`expenseRep1`** as follows:
     - The total amount is the sum of the total amounts of the two expense reports.
     - The start date is the minimum start date of the two expense reports.
     - The end date is the maximum end date of the two expense reports.
     - The detailed description of expense items should consist of the detailed description for **`expenseRep1`** followed by the detailed description for **`expenseRep2`**.
   - You may find the function **`strncmp`** from the standard library `string` useful for the purpose of determining the minimum / maximum dates. Note that the way we formatted the dates (`YYYYMMDD`) implies a direct correspondence between chronological order of dates and their lexicographic order (or order of numeric value as an 8-digit integer).
   - To merge the detailed descriptions of the two expense reports, you will need to re-allocate the space for the detailed description string of **`expenseRep1`**. In principle, this could simply be done by using the function **`realloc`** from the standard library `stdlib`. However, since you only have access to allocation functions from our virtual heap (which does not have realloc functionality), you should manually reallocate the space by allocating new space using **`ourMalloc`** and freeing the old space using **`ourFree`**.
   - Make sure to allocate sufficient space for the merged string, including the terminating `'\0'` and the newline character `'\n'` separating the two detailed descriptions. Do not allocate extra space, as this will result in testing errors.
   - Copy over the contents of the two description strings from **`expenseRep1`** and **`expenseRep2`** to the newly allocated space. You may find the functions **`strcpy`** and **`strcat`** from the standard library `string` useful for this purpose.
   - Free the space allocated by the previous description string of **`expenseRep1`**.
   - If the merge operation succeeded, then the function should return the merged expense report object (**`expenseRep1`**).
   - If the allocation of the merged detailed description failed, then the function should not modify the **`expenseRep1`** object and return NULL.
   - The object **`expenseRep2`** should remain unchanged throughout this process.

Detailed execution examples are provided in **Test 1B**, which you should run after implementing Problems 1.5 and 1.6. For a basic example, consider a case where **`expenseRep1`** and **`expenseRep2`** hold the expense reports #1 and #2 described on page 2, and we invoke **`mergeExpenseReports(expenseRep1, expenseRep2)`**. This will modify **`expenseRep1`** to hold the merged expense report described below, and it will not modify the object **`expenseRep2`**.

Merged expense report (new contents of **`expenseRep1`**):

- Total amount: `1318.56`
- Start date: `19991227`
- End date: `20110531`
- Description: `"flight to NY\nHotel 1+1 night off\n Taxi\n lunch with visitors, 20% off"`

5. Implement the function **`numExpenseItems`**, which returns the number of items in a given expense report. Follow these guidelines:
   - Function signature: **`int numExpenseItems(ExpenseReport expenseRep)`**
   - The number of items is defined by the number of "lines" in the detailed description of the expense report. The number of lines may be determined by iterative invocation of function **`strchr`** from the standard library `string`.

6. Implement the function **`printExpenseReport`**, which prints a given expense report using the following format:

```
<N> expense(s) reported between DD/MM/YYYY and DD/MM/YYYY:
< ITEM 1 >
< ITEM 2 >
. . .
< ITEM N >
Total amount: <TOTAL_AMOUNT> NIS
```

   - Function signature: **`void printExpenseReport(ExpenseReport expenseRep)`**.
   - The number of expense items **`<N>`** (determined by calling **`numExpenseItems`**) is printed as a decimal integer using a field width of 3 (aligned to the right).
   - The start and end dates are printed in the format **`DD/MM/YYYY`**. For example, the date July 28, 2020, should pe printed as **`28/07/2020`**.
   - The descriptions of each expense item should be printed in a separate line.
   - The total amount should be printed in precision of one position after the decimal point (in resolution of 10 אגורות).
   - Print a newline after the total amount.
   - For detailed examples, see the execution examples in **Test 1B**.

Validation: Test **1B** checks the merge function you implemented in Problem 1.4 as well as the two simpler functions from Problems 1.5 and 1.6. For information on the test, containing execution examples and expected output, please examine `/share/ex_data/ex6/test_ex6_1B.out`.

To run Test 1B, compile your code with the test code file `/share/ex_data/ex6/test_ex6_1B.c` as follows:

```
==> gcc  expenseReport.c  /share/ex_data/ex6/test_ex6_1B.c \
         /share/ex_data/ex6/virtualHeap.c  -Wall -o test_ex6_1B
```

After your source has been **successfully compiled** (no errors or warning), you should run the program `test_ex6_1B` and compare the output (using `diff`) with the expected output provided in `/share/ex_data/ex6/test_ex6_1B.out`.

Final testing for Problem 1: Execute script `/share/ex_data/ex6/test_ex6.1` from the directory containing your `expenseReport.c` source file.

(the script produces a detailed error report to help you debug your code)

Problem 2:

The purpose of this problem is to implement a program that reads a list of expense items from an input file, generates reports for them, and prints a summary to the standard output.

Coding instructions:

- Implement the two functions specified below in file `processExpenses.c`.
- To enable use of the ExpenseReport object, you should include `/share/ex_data/ex6/expenseReport.h`. at the top of the file. You may also include the standard libraries `stdio` and `string` , but no other libraries (see guidelines on page 1). No need to include the header file for the virtual heap in `processExpenses.c`, because you will not directly be invoking its memory allocations function.
- Since the source file `processExpenses.c` contains a **`main`** function, it is not considered a code module that can be used by another program. Therefore, the declarations of the function that you implement in Problem 2.1 (or additional "helper" functions you implement) should appear at the top of the file and not in a header file.

1. Implement the function **`strToExpenseItem`**, which creates an expense report based on information specified in a given string. Follow these guidelines:
   - Function signature: **`ExpenseReport strToExpenseItem(const char* str)`**
   - The input string **`str`** is assumed to have the following structure:

     "<SPACE1>**<TOKEN1>**<SPACE2>**<TOKEN2>**<SPACE3>**<TOKEN3>**"

     Where:

     - <SPACE1>, <SPACE2>, and <SPACE3> correspond to sequences of space characters (`' '`), with possibly more than one space per sequence. The first sequence (<SPACE1>) may be empty, but the other two contain at least one space character. Note that other white space characters (e.g. `'\t'` and `'\n'`) are not considered as spaces for this purpose.
     - **<TOKEN1>** and **<TOKEN2>** correspond to (non-empty) sequences of characters that do not contain a space (`' '`).
     - **<TOKEN3>** corresponds to the remaining tail of the string **`str`**. In particular, it starts with a non-space character, but may contain space characters.
   - The function should create a new ExpenseReport object with a single expense item, using **<TOKEN1>** to set the date of the expense, **<TOKEN2>** to set the expense amount, and **<TOKEN3>** to set the description for the expense item.
   - Invoke the **`newExpenseItem`** function from the `expenseReport` module to create the new object. The function parameters for should be set as follows:
     - For the **`amount`** parameter, compute the appropriate **`double`** value by applying a similar procedure to the one you implemented for function

numStringToDouble in HW #5. Note that the implementation is not identical, because the number string is not terminated here with a `'\0'`.

- For the **date** and **description** parameters, pass pointers to the appropriate locations within the input string **str**. Do not copy characters to another buffer for this purpose!

- If the ExpenseReport object was successfully created, the function should return it. Otherwise, the function should return NULL.
- The function should not invoke any dynamic allocation functions other than **newExpenseItem**.
- Note that the parameter **str** is defined as **const char***, so your function cannot (and should not) modify its content in any way.

Examples for valid input strings, and the resulting expense items:

- **strToExpenseItem("20110405 3.45 phone bill + internet .");**
  - amount=3.45
  - date=20110405
  - description="phone bill + internet ."
- **strToExpenseItem(" 2011040511\t11 7 snack");**
  - amount=7.0
  - date=20110405
  - description="snack"
- **strToExpenseItem("201104051\nhello 0007000 refreshments ");**
  - amount=7000.0
  - date=20110405
  - description="refreshments "

Examples for invalid input strings:

- **"20110311 3.45"**
  → <SPACE3> is empty
- **" hello "**
  → **<TOKEN2>** is missing (also, **<TOKEN1>** is not a valid date)
- **" 20110941 3.45 office supplies"**
  → **<TOKEN1>** is an invalid date (day=41; checked by **newExpenseItem**).
- **" 20131023 3.45-7 office supplies \t !"**
  → **<TOKEN2>** is an invalid number (invalid character `'-'`).
- **" 20001212 3.45 office supplies\n"**
  → **<TOKEN3>** contains a new line character (checked by **newExpenseItem**).

Validation: Run test **2A** to test your implementation by compiling your code with the test code file `/share/ex_data/ex6/test_ex6_2A.c` as follows:

```
==> gcc  processExpenses.c expenseReport.c
         /share/ex_data/ex6/test_ex6_2A.c \
         /share/ex_data/ex6/virtualHeap.c  -Wall -o test_ex6_2A
```

After your source has been **successfully compiled** (no errors or warning), you should run the program `test_ex6_2A` and compare the output (using `diff`) with the expected output provided in `/share/ex_data/ex6/test_ex6_2A.out`.

Note: this validation can only be executed before you implement the **main** function.

2. Implement a **`main`** function for the program, by following the guidelines below:
   - The program should be executed with a single input argument corresponding to an input file (which is expected to specify expense items):
     `==> ./processExpenses <inFile>`
   - If no arguments are specified, or the user entered a dash (`-`) instead of a file name for the input argument, the program should read the expense items from the **standard input** (see execution examples below).
   - The program should ignore anything entered in the command line after the first input argument.
   - If the (first) input argument does not correspond to a readable file (or `-`), the program should print the following message to the standard error, and return an exit status of 1:
     `Cannot open <inFile> for reading`
   - You may assume that each line in the input file has no more than 300 characters (not including the newline). Define a symbolic constant representing the maximum line length.
   - You may not assume any limit on the number of lines in the input file.
   - The file contains lines describing expense items and "break lines". Break lines start with the hashtag character (`'#'`) , with no spaces before it, and an arbitrary sequence of characters after it. These are valid break lines:
     ```
     #
     #   report for trip to Hawaii
     #Renovation on 5th floor !!!
     ```
   - Any line that does not start with a hashtag character (`'#'`) is assumed to contain information on an expense item, in the format specified in Problem 2.1 above.
   - The program should read the file, line by line, using the function **`fgets`** from the standard library `stdio`. Use a local character array for this purpose, and set its size according to the assumption on line length mentioned above.
   - As the program reads lines from the file, it should hold an ExpenseReport object containing the relevant information for all expense items specified in the file since the last break line (or the since beginning of the file, if no break lines have been read yet). The order of items specified in the detailed description of the report should correspond to the order of their appearance in the file.
   - When the program reads a line that it does not identify as a break line, it should attempt to construct an expense item from this line by invoking the function **`strToExpenseItem`** from problem 2.1. Recall that function **`fgets`** writes the newline character at the end of the string it writes to the line buffer (before the terminating `'\0'`).
   - The expense items should be combined in the order that they appear in the file using the function **`mergeExpenseReport`**. If the merge operation fails (due

to insufficient memory), the program should halt, free all objects, print the following message to the standard error, and return an exit status of 2:

```
Out of memory
```

- When the program reads a break line, it should print the expense report corresponding to all items specified in the file since the last break line. The report should be printed to the standard output using the function **`printExpenseReport`**. If there were no valid expense items listed since the previous break line (and this is not the first break line), the program prints:

  ```
  No expenses to report
  ```

- After printing the expense report (or the message above), the function should print the current break line without the first hashtag character (`'#'`).
- Finally, the expense report should be emptied at that stage, to initialize for the next report. Think how best to do this using the functions you implemented in Problem 1.
- Lines that are not break lines and do not correspond to a valid expense item should be skipped, but counted for the final error message at the end (see below).
- Make sure to properly destroy (free) every ExpenseReport object that you create once you no longer need it to avoid memory leaks in your program.
- When your program is done reading from the file, it should print the last expense report (or the `No expenses to report` message if the report is empty), and properly close the input file. No need to close the standard input, if that is where you are reading from (argument `'-'` or no arguments).
- If every line of the input file is either a break line or a line that was used to successfully create an expense item, then the program ends by returning an exit status of 0.
- If, however, the input file contained lines that are not break lines but also could not be used to create a valid expense item, the program should print the following message to the standard error, and return an exit status of 3:

  ```
  File <inFile> contains <N> invalid lines, which were ignored
  ```

  (`<N>` is the number of invalid lines)

Compilation guidelines and testing:

- To create the executable for your program, compile your code as follows:

```
==> gcc -Wall \
       expenseReport.c processExpenses.c \
       /share/ex_data/ex6/virtualHeap.c    -o processExpenses
```

- To help you test the resulting executable (`processExpenses`), we provide a working executable `processExpenses` and an example of an input file `sample-expenses.txt`, both of which you may find in the directory `/share/ex_data/ex6/`. You should copy the sample file to your home directory, modify it in different ways, and make sure that your program produces identical output to our program. In particular, apply modifications that will challenge your program by testing different end cases.

Basic execution examples:

1. Execute on sample file, redirect output to `report1.txt`, check exit status, and compare output to the expected output

```
==> ./processExpenses /share/ex_data/ex6/sample-expenses.txt > report1.txt
File  /share/ex_data/ex6/data/sample-expenses.txt  contains  5  invalid
lines, which were ignored

==> echo $?
3
==> /share/ex_data/ex6/processExpenses
       /share/ex_data/ex6/sample-expenses.txt > report1-expected.txt
File /share/ex_data/ex6/data/sample-expenses.txt contains 5 invalid
lines, which were ignored

==> diff report1-expected.txt report1.txt
              ←no output from diff, indicating that file is as expected
```

2. Execute on sample file by piping it to standard input.

```
==> cat /share/ex_data/ex6/sample-expenses.txt | \
         ./processExpenses > report2.txt
File stdin contains 5 invalid lines, which were ignored

==> echo $?
3

==> diff report1.txt report2.txt
              ←no output from diff, indicating outcome is identical to scenario #1
```

3. Execute on sample file without the last 6 lines (5 invalid lines and a break).

```
==> head -n-6 /share/ex_data/ex6/sample-expenses.txt | \
         ./processExpenses - > report3.txt

==> echo $?
0             ←no invalid lines

==> diff report2.txt report3.txt
23,24d22      ←the last two lines of report2.txt should not appear in report3.txt
< final report should be empty
< No expenses to report
```

4. Execute on non-existing input file

```
==> ./processExpenses --
Cannot open -- for reading

==> echo $?
1
```

You should continue to validate your program by copying over the sample file to your exercise directory, modifying it in various ways, executing your program and our program on the modified file, and comparing the outputs, error messages, and exit status, as demonstrated in the first execution example.

Final testing for Problem 2: Execute script `/share/ex_data/ex6/test_ex6.2` from the directory containing your `expenseReport.c` and `processExpenses.c` source files. The script compares the output of your program to the output of our program on a list of inputs, using the procedure described above. As always, it produces a detailed error report to help you debug your code).

Submission Instructions:

1. After you validated and tested your solution, make sure that your `~/exercises/ex6/` directory contains the following C source files:
   - `expenseReport.c`
   - `processExpenses.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 (**12/7 @ 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.