Assignment 2: Arrays, Strings and Modules with Pointers and Memory Management

Computer Programming (TPTG11) Autumn 2021

1 Assignment Description

The intended learning outcomes of this assignment, includes using arrays, strings, modules, pointers and dynamic memory management. The assignment is organized into 3 problems, where the first two problems involve creating a library of two modules, and the final problem uses the library (two modules) to create a main program.

Start by creating a file `main.c`, with a `main()` function, which you will use to test your modules while developing them. Note! Be sure to put everything related to each module's interface in its *.h* file and everything related to its implementation in its *.c* file. Document everything in the *.h* file (with comments), and where appropriate in the *.c* file. Also make sure you follow the coding conventions used in the book.

1.1 Problem 1: The `intArray` Module (`intarray.h`, `intarray.c`)

In this problem, you will create a module `intArray` (in `intarray.c` and `intarray.h`) for working with integer arrays that are allocated on the program's *stack* memory. All elements in the array are positive integers, with the exception of one special element with the value `-1`, which acts as a `SENTINEL` (terminator) for the end of the array (compare to the null terminating character for strings).

In this problem, the arrays are not dynamic, i.e. once we set the array length, it will remain fixed throughout the entire program. We will differentiate between the actual length of an array, which we will call the array's `CAPACITY`, and the effective length of the array, which we will call the array's `size`. The `CAPACITY` is the actual length of the array, i.e. how many elements it can hold, whereas the `size` of the array is the number of elements we are using in the array. We will create *symbolic constants*, using the `#define` preprocessor directive, for the `SENTINEL` and `CAPACITY` (make sure the `CAPACITY` is large enough for its usage in the assignment).

We will also use the keyword `typedef` to define the alias `intArray` for our int array type, i.e. `typedef int intArray[CAPACITY]`. In the example below, the `intArray`'s indices (indexes) go from `0` to `CAPACITY-1`. We also see the `size` of the current array is `4`, since we are only using the first `4` elements in the array to store the values `5`, `3`, `2` and `8`. Also notice how the `SENTINEL` (with a value of -1) is used to indicate where the `effective length` of the array ends.

Value	5	3	2	8	-1	$< undefined >$	$\cdots$	$< undefined >$
Index	0	1	2	3	4	5	$\cdots$	CAPACITY-1

`intArray`

We can create a variable of our new alias type (e.g. `intArray array;`) and even use it to declare parameters in functions (e.g. `void function(intArray array);`). These expressions are just aliases for `int array[CAPACITY]` due to our `typedef` above. Now let's start coding!

1. Write a function `printIntArray()` that takes an `intArray` as a parameter, and prints the array's elements to standard output `stdout`, i.e. the terminal. Note! Make sure you don't read past, or print, the `SENTINEL`.

   Example:

   ```
   intArray a = {2,6,8,4,5,SENTINEL}; // notice the SENTINEL -1
   printIntArray(a);
   ```

   Output:

   ```
   [ 2 6 8 4 5 ]
   ```

2. Write a function `getIntArraySize()` that takes an `intArray` as a parameter, and returns its `size`. Remember, the `size` is the number of elements before the `SENTINEL`.

   Example:

   ```
   intArray a = {2,6,8,SENTINEL};
   printf("%u", getIntArraySize(a));
   ```

   Output:

   ```
   3
   ```

3. Write a function `getIntArray()` that takes an `intArray` as a parameter, and reads in a string from the user (via the terminal). The function should first check that the string has the format $< \textit{unsigned int} >, < \textit{unsigned int} >, < \textit{unsigned int} >$, i.e. a comma-separated list of positive integers. To be a valid string, it must not contain two consecutive commas, and may not start or end with a comma (and, of course, only contain comma-separated positive integers). If the string is valid, the function should extract the positive integers from the string, enter them into the `intArray`, including the terminating `SENTINEL`, and then return the value `1` (`true`) as the function's return value. If the string isn't valid, the function should just enter the terminating `SENTINEL` at index `0` and return the value `0` (`false`).

   Note! The header file `stdbool.h` defines three, so called, *macros* `bool`, `true` and `false` for working with boolean expressions. A `bool` is simply a kind of alias for `int`, and `true` and `false` simply have the values `1` and `0` respectively. So, if you want, you could `#include "stdbool.h"` and use these "aliases" instead. In that case, your function would return a `bool` with the value `true` or `false`.

   Example:

   ```
   intArray a = {0};
   bool b;

   printf("Please enter a comma-separated list of positive integers: ");
   b = getIntArray(a);

   printf("Please enter a comma-separated list of positive integers: ");
   b = getIntArray(a);
   ```

   Output:

   ```
   > Please enter a comma-separated list of positive integers: 1,2,3
   [ 1 2 3 ]
   > Please enter a comma-separated list of positive integers: 1,,2,3
   [ ]
   ```

4. Write a function `appendIntArray()` that takes 3 `intArrays` `a`, `b` and `c` as a parameters. When the function returns, `c` should contain the elements of `a` concatenated with the elements of `b` (i.e. so that all elements in `a` preceed all elements in `b`). If necessary, the function should truncate the result so that the `size` of `c` does not exceed the array's `CAPACITY`. Also note, that array `c` should only contain 1 `SENTINEL` as the last element (after the last useful element added to it)!

   Example:

   ```
   intArray a = {2,6,8,4,5,SENTINEL}, b = {3,4,6,7,9,SENTINEL}, c;
   appendIntArray(a,b,c);
   printIntArray(c);
   ```

   Output:

   ```
   [ 2 6 8 4 5 3 4 6 7 9 ]
   ```

5. Write a function `interleaveIntArray()` that takes 3 `intArrays` `a`, `b` and `c` as a parameters. When the function returns, `c` should contain the elements of `a` and `b` interleaved with each other (i.e. a[0], b[0], a[1], b[1], etc). If there aren't an equal amount of elements in `a` and `b`, the function should fill the remaining elements of `c` with the remaining elements in `a` or `b`, whichever is longer. If necessary, the function should truncate the result so that the `size` of `c` does not exceed the array's `CAPACITY`. Also note, that array `c` should only contain 1 `SENTINEL` as the last element (after the last useful element added to it)!

   Example:

   ```
   intArray a = {6,7,9,12,SENTINEL}, b = {3,5,6,9,1,2,3,SENTINEL}, c;
   interleaveIntArray(a,b,c);
   printIntArray(c);
   ```

   Output:

   ```
   [ 6 3 7 5 9 6 12 9 1 2 3 ]
   ```

6. Write a function `sortIntArray()` that takes an `intArray` as a parameter, and returns the sorted array. The user should be able to choose if the array should be sorted in ascending or descending order via another function `setSortOrder()`. Furthermore, the user should be able to choose if the array should only contain unique values or not via yet another function `setUniqueness()`. Note, if the function only returns unique values, the resulting array may contain fewer elements than its original size.

   The settings for the sort order and uniqueness should be implemented as private (`static` variables in the module, and the values assigned to these 2 variables should be from 2 different `enum` types. The values sent in as arguments to `setSortOrder()` and `setUniqueness()` should also be values from these 2 `enums`, respectively. As default, the sort order and uniqueness should be set to ascending and non-unique values in the module.

   Example:

   ```
   intArray a = {6,7,9,12,3,5,6,9,5,7,10,12,SENTINEL};

   setSortOrder(ASCENDING); setUniqueness(NONUNIQUE);
   sortIntArray(a); printIntArray(a);

   setSortOrder(ASCENDING); setUniqueness(UNIQUE);
   sortIntArray(a); printIntArray(a);

   setSortOrder(DESCENDING); setUniqueness(NONUNIQUE);
   sortIntArray(a); printIntArray(a);

   setSortOrder(DESCENDING); setUniqueness(UNIQUE);
   sortIntArray(a); printIntArray(a);
   ```

Output:

```
[ 3 5 5 6 6 7 9 9 10 12 12 ]
[ 3 5 6 7 9 10 12 ]
[ 12 12 10 9 9 7 6 6 5 5 3 ]
[ 12 10 9 7 6 5 3 ]
```

1.2 Problem 2: The `fMatrix` Module (`fmatrix.h`, `fmatrix.c`)

In this problem, you will create a module `fMatrix` (in `fMatrix.c` and `fMatrix.h`) for working with floating-point 3x3 matrices (2D arrays) that are allocated on the *heap*, also known as the *free store*. This involves managing (allocating and freeing) memory.

As in problem 1, we will create an alias for our matrix type using `typedef float fMatrix[ROWS][COLS]`, where `ROWS` and `COLS` are symbolic constants in the module. Note, that we are not using any `SENTINEL` for our matrix, but instead rely on the fixed sizes `ROWS` and `COLS`. We will not be expanding the matrix dynamically in this assignment (you will work with this in assignment 3).

1. Write a function `createMatrix()` that takes a `float` as a parameter. The function should allocate memory on the heap for a `fMatrix`, fill it (each element) with the value of the `float` input parameter, and finally return a pointer to the matrix, i.e. a pointer to a `fMatrix`. If the memory could not be allocated, for any reason, the function should return `NULL`.

   Example:

   ```
   fMatrix *m = createMatrix(0.0f);
   ```

   Output:

   ```
   There is no output, but use the debugger (Locals) in Visual Studio Code
   to check that the memory was allocated and filled with the correct value.
   ```

2. Write a function `destroyMatrix()` that takes a pointer to a `fMatrix` as a parameter, and deallocates (frees) the memory for the matrix on the heap. The function should not return a value.

   Example:

   ```
   fMatrix *m = createMatrix(0.0f);
   destroyMatrix(m);
   ```

   Output:

   ```
   There is no output, but use the debugger (Locals) in Visual Studio Code
   to check that the memory was deallocated (freed) properly.
   ```

3. Write a function `printMatrix()` that takes a pointer to a `fMatrix` and a prints its elements to standard output (`stdout`), i.e. the terminal.

   Example:

   ```
   fMatrix *m = createMatrix(0.0f);
   printMatrix(m);
   destroyMatrix(m);
   ```

   Output:

   ```
   0.00 0.00 0.00
   0.00 0.00 0.00
   0.00 0.00 0.00
   ```

4. Write a function `getMatrix()` that takes a pointer to a `fMatrix` as a parameter, and reads in a string from the user (via the terminal). The function should first check that the string has the format $< float >, < float >, < float >$, i.e. a comma-separated list of floating-point numbers. To be a valid string, it must not contain two consecutive commas, and may not start or end with a comma (and, of course, only contain comma-separated floating-point numbers). Also note that the string can contain punctuation marks ".", since the floating-point numbers may contain a decimal point. If the string is valid, the function should extract the floating-point numbers from the string, enter them into the `fMatrix`, and then return the value `1` (`true`) as the function's return value. If the string isn't valid, the function should not modify the `fMatrix` in any way and return the value `0` (`false`).

   Example:

```
fMatrix *m = createMatrix(0.0f);
printMatrix(m);
printf("Please enter a comma-separated list of 9 floats: ");
getMatrix(m);
printMatrix(m);
printf("Please enter a comma-separated list of 9 floats: ");
getMatrix(m);
printMatrix(m);
destroyMatrix(m);
```

   Output:

```
0.00 0.00 0.00
0.00 0.00 0.00
0.00 0.00 0.00
> Please enter a comma-separated list of 9 floats: 1,2,,a,4.0,;,6.00,abc
0.00 0.00 0.00
0.00 0.00 0.00
0.00 0.00 0.00
> Please enter a comma-separated list of 9 floats: 1,2,3,4.0,5,6.00,7,8,9
1.00 2.00 3.00
4.00 5.00 6.00
7.00 8.00 9.00
```

5. Write a function `matadd()` that takes two pointers to `fMatrix` as parameters. The function should add the two matrices and store the result in the first matrix (pointed to by the first input parameter).

   Example:

```
fMatrix *m1 = createMatrix(0.0f), *m2 = createMatrix(0.0f);
printf("Please enter a comma-separated list of 9 floats: ");
getMatrix(m1);
printMatrix(m1);
printf("Please enter a comma-separated list of 9 floats: ");
getMatrix(m2);
printMatrix(m2);
matadd(m1,m2);
printf("Result:\n");
printMatrix(m1);
destroyMatrix(m1); destroyMatrix(m2);
```

   Output:

```
> Please enter a comma-separated list of 9 floats: 10,20,10,4,5,6,2,3,5
10.00 20.00 10.00
 4.00  5.00  6.00
 2.00  3.00  5.00
> Please enter a comma-separated list of 9 floats: 3,2,4,3,3,9,4,4,2
 3.00  2.00  4.00
```

```
 3.00  3.00  9.00
 4.00  4.00  2.00
Result:
13.00 22.00 14.00
 7.00  8.00 15.00
 6.00  7.00  7.00
```

6. Write a function `matmul()` that takes two pointers to `fMatrix` as parameters. The function should multiply the two matrices (matrix multiplication) and store the result in the first matrix (pointed to by the first input parameter).

   Example:

```
fMatrix *m1 = createMatrix(0.0f), *m2 = createMatrix(0.0f);
printf("Please enter a comma-separated list of 9 floats: ");
getMatrix(m1);
printMatrix(m1);
printf("Please enter a comma-separated list of 9 floats: ");
getMatrix(m2);
printMatrix(m2);
matmul(m1,m2);
printf("Result:\n");
printMatrix(m1);
destroyMatrix(m1); destroyMatrix(m2);
```

   Output:

```
> Please enter a comma-separated list of 9 floats: 10,20,10,4,5,6,2,3,5
10.00 20.00 10.00
 4.00  5.00  6.00
 2.00  3.00  5.00
> Please enter a comma-separated list of 9 floats: 3,2,4,3,3,9,4,4,2
 3.00  2.00  4.00
 3.00  3.00  9.00
 4.00  4.00  2.00
Result:
130.00 120.00 240.00
 51.00  47.00  73.00
 35.00  33.00  45.00
```

1.3 Problem 3: The `Main` Program

In this final problem, you will create a main program that uses the functionality provided by your two modules from problems 1 and 2. For this, you will reuse the file `main.c` that you have been using to test your two modules above. In your main program file `main.c`, write a program that uses your two modules so that it gives the same output as below.

```
== Main Menu ==
1. Append two integer arrays.
2. Interleave two integer arrays.
3. Sort an integer array.
4. Add two 3x3 float matrices.
5. Multiply two 3x3 float matrices.
6. Print Main Menu
7. Quit

?: 1
Enter first arrary: 1,2,3
Enter second arrary: 4,5,6
Result:
[ 1 2 3 4 5 6 ]

?: 2
Enter first arrary: 6,7,9,12
Enter second arrary: 3,5,6,9,1,2,3
Result:
[ 6 3 7 5 9 6 12 9 1 2 3 ]

?: 3
Enter array: 4,7,5,2,5
Enter Sort order (0) ASC, (1) DESC: 1
Unique Values (0) No, (1) Yes: 1
Result:
[ 7 5 4 2 ]

?: 4
Enter first matrix (9 floats): 10,20,10,4,5,6,2,3,5
Enter second matrix (9 floats): 3,2,4,3,3,9,4,4,2
Result:
 13.00  22.00  14.00
  7.00   8.00  15.00
  6.00   7.00   7.00

?: 5
Enter first matrix (9 floats): 10,20,10,4,5,6,2,3,5
Enter second matrix (9 floats): 3,2,4,3,3,9,4,4,2
Result:
130.00 120.00 240.00
 51.00  47.00  73.00
 35.00  33.00  45.00

?: 6
== Main Menu ==
1. Append two integer arrays.
2. Interleave two integer arrays.
3. Sort an integer array.
4. Add two 3x3 float matrices.
5. Multiply two 3x3 float matrices.
6. Print Main Menu
7. Quit

?: 7
```

2 Requirements

- You must write the code yourselves. Code snippets from the textbook and/or lectures can of course be used, but it is not allowed to find code on the internet.
- Furthermore, you are only allowed to use the constructs covered so far in the course, i.e. no structs, files or such.
- The program that you submit must compile and run without modification. This means that any code that gives compilation or runtime errors must be commented out.

3 Grading criteria

You will receive the grade pass or fail. In order to pass, the program must be complete, i.e. have the full functionality described above. In addition, the program must be well-designed and well-coded, especially with regards to:

- The separation of interface and implementation into each module's .h and .c files respectively.
- How it is broken down into functions (especially if you use helper functions in you modules).
- Adequate data types and consistency in this between functions.
- Proper use of string functions, pointers and memory management.
- Naming of functions and variables must be in accordance with the principles presented in the course book.
- Comments for defines, typedefs, symbolic constants and function prototypes in header files (.h files),
- Comments describing non-trivial functionality in implementation files (.c files).

4 Submission

You must submit your solution via Canvas, in a zip file containing your Visual Studio workspace folder with all your source code (all documentation is done using comments in your source code). Don't submit anything else (just one zip file).

Deadline for submission is Friday 3rd of December at 17.15.

Re-examination deadline is January 31st, 2022.