# Lab 3 (Oct. 23-Nov. 3)

## Strings. Dynamic Memory Allocation. C Structures.

**This lab is worth 5% of the course mark.**
**The *Bonus Question* is worth additional 1% of the course mark.**
**The number in square brackets at the beginning of each question shows the number of points the question is worth. The total number of points is 50 (+ 10 bonus).**

*You will receive a mark only for the codes that are demonstrated in front of a TA by the end of the lab. With that in mind, please complete as many problems as you can before you come to lab. Work not completed by the end of the lab will not be marked. TAs will leave the lab at 5:20pm sharp.*

*Additionally, you have to submit on Avenue a text file (with extension **txt**) for each program, containing the source code for that program (the main function and all other functions). Naming instructions for the files are provided on Avenue. The online submission has to be done by the end of the lab session.*

**Note:** See Lab 3 notes for info on how to open a file, read and write in a file. See lecture notes for Topics 8, 9 and 10 for information on dynamic memory allocation, and Topic 10 for structure types.

### General Requirements on Your Programs

A. Your programs should be written in a good programming style, including instructive comments and well-formatted, well-indented code. Use self-explanatory names for variables as much as possible. (~5% of the mark)
B. When outputting the results, include an explanatory message in the output. When inputting values display a message prompting the user to input the appropriate values. (~10% of the mark)

## Lab Question

1) [8] Implement a string processing function with the prototype

- `char *my_strcat( const char * const str1, const char * const str2);`

The function creates a new string by concatenating `str1` and `str2`. The function has to call `malloc()` or `calloc()` to allocate memory for the new string, i.e., for the total number of characters, plus the null character. The function returns the new string (i.e., the value of a pointer to the first array element, in other words, the address of the first array element). You are not allowed to call functions declared in the standard library header `<string.h>`, except for `strlen()` (`strlen(s)` returns the size of string `s`). After executing the following call to `printf()`:

```
printf ( "%s\n", my_strcat( "Hello", "world!" ));
```

the printout on the screen has to be

```
Helloworld!
```

Write a program to test the function.

2) [42] Write a program to manage the grades of students in a class. To represent the information of each student define a structure type named `student`. The members of this structure should be:

1. an integer that represents the student's ID number,
2. an array of 15 characters storing the first name,
3. an array of 15 characters storing the last name,
4. an integer that represents the project 1 grade,
5. an integer that represents the project 2 grade,
6. a floating point representing the final course grade.

To store the list of students use an array of pointers of type `student*`. Each pointer in the array has to point to a structure variable of type `student` representing the information of one student. The array has to be **SORTED** in increasing order of students' ID numbers.

## Define the following functions

- `student **create_class_list( char *filename, int *sizePtr )`

Function `create_class_list()` reads the students' ID numbers and names from the input file (a text file), **allocates** the memory necessary to store the list of students and initializes the students ID's and names. Note that variable `filename` represents a string which stores the name of the input file. Additionally, note that the input file is a text file which contains a positive integer representing the number of students at the beginning, then on each line the id of a student, blank, first name, blank, last name. The ID numbers appear in the file in increasing order. The students should appear in the class list in increasing order of their IDs, too. The function should also appropriately change the value of the variable in the caller that is supposed to store the number of students in the class. Note that parameter `sizePtr` is a pointer to that variable. The function has to additionally initialize all students' grades to 0 (note that this is done automatically if you allocate the memory for each `student` variable using `calloc()`). Finally, the function has to return a pointer to the beginning of the array of pointers to `student`. You may assume that each student's first name, respectively last name, has no more than 14 characters. Example of an input file containing the info of three students:

3
1200 Isaac Newton
4580 Alan Turing
9000 Elvis Presley

Note: Use the variable `filename` as the first argument of `fopen()` when opening the file.

- `int find( int idNo, student **list, int size )`

Parameter `list` is a pointer to the beginning of the array of pointers representing the class list. Parameter `size` represents the number of students in the list (you may assume it is larger than or equal to 0). Function `find()` determines if there is a student in the list whose ID number equals `idNo`. If there is such a student the function returns its position (i.e., index) in the list (note that indexing starts with 0). If such a student is not found the function returns -1.

- `void input_grades( char *filename, student **list, int size )`

Parameter `list` is a pointer to the beginning of the array of pointers representing the class list. Parameter `size` represents the number of students in the list. Parameter `filename` represents the name of a text file. Function `input_grades()` reads the project 1 and project 2 grades from the text file and inputs them in the class list (in other words, it modifies the information of each student accordingly). The file contains the information for each student on a separate line, first the ID number, then the project 1 grade followed by the project 2 grade, all separated by white spaces. Note that **the file is not sorted**, therefore the ID numbers do not necessarily appear in increasing order. You may need to use function `find()` to determine the position of a student in the list.

- `compute_final_course_grades()` - you need to figure out the prototype

Function `compute_final_course_grades()` computes the final course mark for each student as the arithmetic average of project 1 and project 2 grades. The function accordingly modifies the information in the class list.

- `output_final_course_grades()` - you need to figure out the prototype

Function `output_final_course_grades()` outputs the final course marks of all the students in a file whose name is passed to the function, as follows. The first line in the file must contain the number of

students. Each line after that should contain an ID number followed by the corresponding grade, separated by white spaces. The data has to be sorted in increasing order of ID numbers.

- `void print_list( student **list, int size)`

Parameter `list` is a pointer to the beginning of the array of pointers representing the class list. Parameter `size` represents the number of students in the list. The function has to print on the screen the information of each student in the list, one student per line. For instance, the info of a student may appear as follows:
ID: 1200, name: Isaac Newton, project 1 grade: 100, project 2 grade: 100, final grade: 100.00

- `void withdraw(int idNo, student **list, int *sizePtr)`

Parameter `list` is a pointer to the beginning of the array of pointers representing the class list. Parameter `sizePtr` is a pointer to the variable in the caller that stores the number of students in the list. Function `withdraw()` has to remove from the class list the student whose ID number equals `idNo`. If the student is not in the list the function has to print a message specifying that. When a student is removed from the class list the vacated spot in the array of pointers must be filled by shifting one position to the left all "students" between the vacated position and the end of the array. Additionally, **the structure variable storing the information of the student who was withdrawn has to be deallocated. NOTE:** Use the C library function `free()` to deallocate memory that was allocated with `calloc()` or `malloc()`.

- `void destroy_list(student **list, int *sizePtr)`

Parameter `list` is a pointer to the beginning of the array of pointers representing the class list. Parameter `sizePtr` is a pointer to the variable in the caller that stores the number of students in the list. Function `destroy_list()` deallocates all the memory used to store the class list and sets to 0 the variable in the caller that stores its size. **NOTE:** Use the C library function `free()` to deallocate memory that was allocated with `calloc()` or `malloc()`.

Write a program to manage the students' grades, which uses all the functions specified above. Your program has to invoke function `withdraw()` at least three times, one time corresponding to an unsuccessful withdrawal (in other words, when the student is not in the list) and two times corresponding to successful withdrawals. **NOTE:** The only C library functions you are allowed to use are those for input/output (including from/to files) and for memory allocation/deallocation.

To create an input file complying to the above specified format use simple text editors such as Notepad or Textpad. When using Netbeans you have to include the input file into the corresponding project by doing the following. First save the file in the directory containing the project. In the Netbeans Navigator select your project and right-click on "Important Files". Then right-click on "Add item to Important Files" and select the input file.

3) **[10] Bonus Question.** Write a program that reads a sequence of words from an input file, sorts the words in alphabetical order and then outputs them on the screen. For this exercise a word is a sequence of lower case letters. Note that the alphabetical order is consistent with the increasing order of the integer values of the characters. To read the data from the input file your program invokes the function `read_words()` with prototype

   - `char **read_words(const char *input_filename, int *nPtr).`

   Note that `input_filename` is a string representing the name of the input file. This function has to store the words in an array of strings. The memory for the array of strings and for each string has to be allocated dynamically (i.e., using `malloc()` or `calloc()` ). Do not allocate more memory **then necessary for the array of strings and for the individual words. The input file contains a positive integer representing the number of words, on the first line.** Then the words follow one per line.

Function `read_words()` has to store the number of words in the variable pointed to by `nPtr`. Additionally, the function returns a pointer to the beginning of the array of strings that was dynamically allocated.

Next your program has to invoke the function `sort_words()` and next invoke function `output_words()`. Function `sort_words()` has to sort the words in the array of strings in alphabetical order using the sorting algorithm known as "**insertion sort**". You have to determine the appropriate prototype for this function. Include a description of the sorting algorithm in a comment at the beginning of the function. Additionally, you have to write your own function to compare alphabetically two words. Function `output_words()` has to print all words on the screen in the order they appear in the array of strings, one word per line. Determine the appropriate prototype for this function.

Write another function `sort2_words()` which uses a different sorting algorithm (of your choice) to sort the words alphabetically. Include a description of the sorting algorithm in a comment at the beginning of the function. Test the function.

For this exercise you **are allowed** to use from the standard string processing library **only the functions `strlen()` and `strcpy()`. The only other C standard library functions that you are allowed to use are for input/output, for opening a file, and for dynamic memory allocation.**