Bern University
of Applied Sciences

BTF1230 - Grundlagen der Softwareentwicklung in C

Laboratory III – Student Management Application

TABLE OF CONTENTS

1 The exercise

This exercise requires different methods as well as syntax constructs seen in the class "Fundamentals of Software Development by using C – BTF1230".
Write a C program to keep records and perform statistical analysis for a class of 20 students – optional; do not limit the number of students. The information of each student contains :

- ID
- First-name
- Family-name
- E-Mail address
- Sex
- Quizze_1
- Quizze_2
- Mid-term score
- Final score (average)
- Total score
- Final grade

The program provides different features. The basic features are shown in a sample menu dialogue later in this document. The menu dialogue contains the following entries.

1. Add student records
2. Delete student records
3. Update student records
4. View all student records

5. Calculate an average of a selected student's scores
6. Show student who gets the max total score
7. Show student who gets the min total score
8. Find student by ID
9. Sort records by total scores

As optional features add three new functions to the menu. The optional functions require File-IO skills as described in chapter 17 of the companion book[1]. The additional three functions allow a user to create, load/modify/store and delete a data-base file. The file is used to save added information. It is like a simple data-base for previously added entries. This feature makes the program useful over a longer period of time. To keep things simple start by a initial version of the software without the File-IO feature.
To keep track of the entire development process use Git. Read available resources on the web and/or ask your course advisor (below some useful URLs are given).

2 Example Ctrl Menu

In the figure 1 a example of the basic control menu is shown. To make the menu easy modifiable, later, isolate the menu in a separate function.

```
**************************************************************
**       Control Menu - Student Management System           **
**************************************************************

1 : Add student records
2 : Delete student records
3 : Update student records
4 : View all student records
5 : Calculate an average of a selected student's scores
6 : Show student who gets the max total score
7 : Show student who gets the min total score
8 : Find student by ID
9 : Sort records by total scores
```

Figure 1: Example Ctrl Menu dialogue

3 Implementation Advice

To get started as well as for guidance follow the hints given in that section. First-of-all, split this program into sub-problems by using the method of structured programming as describe in chapter 5. Group functions and sub-function into "modules" each module has a separate source and header file. This method gives you one essential advantage during development. It allows you to test modules and its sub-functions. Therefore each module has its own test file name "test_<MODULE_NAME>". As for simplicity one use either a IDE or simple build-system based on make. Because we use Git as version control system one can use branch feature to develop each module in its on feature branch. Later we merge module branches together into integration branch and for deployment we merge integration branch into master branch[2]. The described workflow tend to be complex at the

[1] Sams Teach Yourself C Programming in One Hour a Day seventh edition
[2] To get familiar with Git workflows and implicit become a Git Pro, read online about Git WF:
- https://git-scm.com/book/en/v2/Git-Branching-Branching-Workflows

beginning. Do not worry about complexity. Use a much simpler workflow with one single master branch to begin with.

3.1 Guideline for a possible implementation

Step 1: Declaring an array of the structure called *studList* to store the records. Each array element is a student record. Keep in mind, this is one possible implementation. It is a easy method to begin with but in reality it is not flexible and will be improved in future. Anyway, the data "set of data objects" (instance of the structure) is based on the *record* structure defined at beginning.

Step 2: Defining the *displayMenu()* function to display the menu. The initial menu implementation provides nine choices from 1 to 9. This menu consists entries to manipulate records and to do evaluation of data-set.

Step 3: Defining the *addRec(studRec_t studList[], int *itemCount)* function to add a new record to the array of student objects. This function takes two arguments. The first argument is the array (set of data objects) of a student objects record (studRec_t) and the second argument is the number of items in the array (pointer to that memory location). This function firstly checks the new record, hence it calls a sub-function to read-in record and second the search function to avoid duplicated records, before it is allowed to be appended to the array. When the new item is added the value of *itemCount* variable increases by 1 that means the number of records in the list increases.

Step 4: Defining the *searchEntry(studRec_t studList[], char id[], int itemCount)* function to search for the index of a target record. This function is useful as we need it to find the location of the target record in the data-set of student objects. It can help us to make sure the record does exist before we allow the record for deleting or updating. If the target element is found, the function returns the index of this element. It return -1, if the target element is not found in the data-set.

Step 5: Defining the *viewAll(studRec_t studList[], int itemCount)* function to display the list of all records in the set. To display all records, we need a while-loop to traverse through the array of student objects.

Step 6: Defining the *delRec(studRec_t studList[], int *itemCount, int itemIdx)* function to delete a target record from the array of student objects. The user will be prompted to enter the id of student record that one wants to delete. Then this id will be checked to make sure it does exist in the list. If the target record or element really exists, the deletion process can be made. The deletion process starts by checking whether the target record is the last record, beginning or middle record. If the target record is the last record in the list, we simply delete the record. The last record is the record that has it index equal to *itemCount-1*. If the target record stays at the beginning or in the middle of the list, we need to use a loop to allow the previous element to take over the next element. This process continues until it reaches the end of the *studList[itemCount-1]*. Then the *cleanRec(studRec_t studList[], int index)* method is called to clean the last element of the list that should not exit. After the element is cleaned, the *itemCount* variable decreases by 1. This means that the number of valid elements in the list decreases.

Step 7: Defining the *updateRec(studRec_t studList[], int itemIdx)* function to update a specified record. The update process starts by asking the user to input the id of the record to be changed. The id value is checked to make sure it really exists. If it exists the change to the target record can be made after asking the user to input the new value of the field that need change.

Step 8: Defining the *avrScore(studRec_t studList[], int itemIdx)* function to calculate the average score of a selected student. The function also starts by asking the user to input the id of the target student. This id is checked to make sure it really exists. The average score can be calculated by dividing the sum of quizz_1 score, quizz_2 score, assignment score, mid-term score, and final score by 5, if all scores are available. If there are scores missing hence initialized by -1 we skip it for average calculation. If we skip a score we decrease the divisor accordingly.

▶ `http://nvie.com/posts/a-successful-git-branching-model/`

Step 9: Defining the *searchRecMaxScore(studRec_t studList[], int itemCount)* and *searchRecMinScore(studRec_t studList[], int itemCount)* functions show about the student who gets the maximum score and the student who gets the minimum score. To find the highest total score or lowest total score, we need to compare every total score of each element. Therefore we calculated average score first.

Step 10: Defining the *searchRec(studRec_t studList[], char id[ID_SIZE], int itemCount)* function to find the record in the list. This function asks the user to enter the id of the student record. Then this id is checked to make sure it really exists. If the record is found, the information of the target student will be displayed. If the record is not found the message *"The record doesn't exist."* will be displayed.

Step 11: Defining the *bubbleSort(studRec_t studList[], int itemCount)* function to sort the records in ascending order by total scores. For the sort technique we use bubble sort algorithm [3]. See the pseudo code below as possible implementations of bubble sort. First listing shows a simple implementation without any optimization, whereas second consists optimization to improve sort cycle time.

```
proc bubbleSort(A : list of sortable items)
  n = length(A)
  repeat
    swapped = false
    for i = 1 to n-1 inclusive do
     /* if this pair is out of order */
     if A[i-1] > A[i] then
       /* swap them and remember something
            changed */
       swap( A[i-1], A[i] )
       swapped = true
     end if
    end for
  until not swapped
end procedure
```

```
proc bubbleSort(A : list of sortable items)

   n = length(A)
   repeat
     newn = 0
     for i = 1 to n-1 inclusive do
       if A[i-1] > A[i] then
         swap(A[i-1], A[i])
         newn = i
       end if
     end for
     n = newn
   until n = 0

end procedure
```

Step 12: Now combine the C code together and test your program.

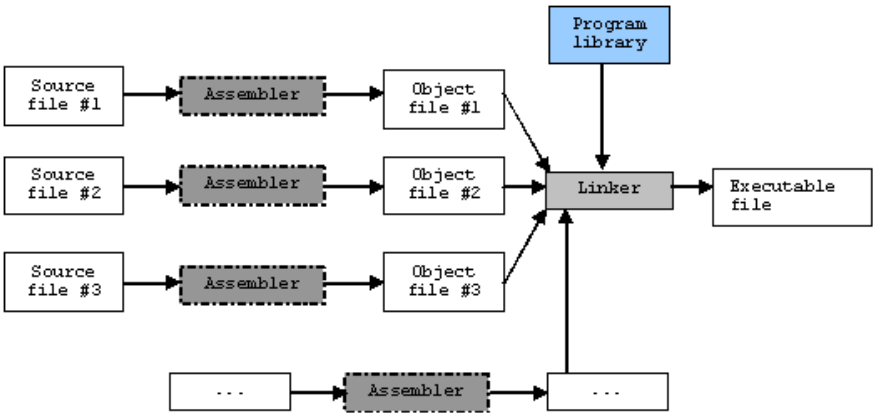

Figure 2: Bring it all together

Do not copy-and-past possible solutions from the Internet. You definitely will find similar exercises and possible solutions. This exercise is thought to be a finale-exam preparation.

[3] Bubble sort, sometimes referred to as sinking sort, is a simple sorting algorithm that repeatedly steps through the list to be sorted, compares each pair of adjacent items and swaps them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted. The algorithm, which is a comparison sort, is named for the way smaller elements "bubble" to the top of the list. Although the algorithm is simple, it is too slow and impractical for most problems even when compared to insertion sort. It can be practical if the input is usually in sort order but may occasionally have some out-of-order elements nearly in position.