Assignment: Data Analysis Using the Heap

Learning Outcomes

- Reading values from files
- Dynamic memory management
- Processing arrays using pointer offsets
- Selection sort algorithm

Task

The objective is to develop a program that analyzes an unknown number of course grades stored in a data file. The program will open the data file, determine the count of grades in the file, allocate appropriate memory in the heap as a container for the grades, store grades read from the file in the previously allocated heap storage, and then analyze the data. The report provides the minimum, maximum, average, and median grades, variance and standard deviation of the grades, and a table indicating the percentage of grades in each letter grade category.

Functions to analyze data

Declare the following functions to solve the problem:

```c
double* read_data(char const *file_name, int *ptr_cnt);
double  max(double const *begin, double const *end);
double  min(double const *begin, double const *end);
double  average(double const *begin, double const *end);
double  variance(double const *begin, double const *end);
double  std_dev(double const *begin, double const *end);
double  median(double *base, int size);
void    selection_sort(double *base, int size);
void    ltr_grade_pctg(double const *begin, double const *end,
                       double *ltr_grades);
```

Your submission must satisfy the following criterion:

> ***You can use the C standard library to define the necessary functions except function `qsort`. Instead of using `qsort`, you will define function `selection_sort` to sort values.***

A brief summary of each of the five functions follows:

1. Function `read_data` opens the text file specified by parameter `file_name`, determines the number of double-precision floating-point values in the file and writes this count to the location pointed to by parameter `ptr_cnt`. Next, the function allocates the exact amount of heap memory to contain these values and copies the values from file to this heap memory. If for any reason, the function is unable to complete these tasks, it should return `NULL`; otherwise, the function should return the address of the first byte of the previously allocated heap memory containing the values read from the file.
2. Functions `max` and `min` return the maximum value and minimum value in a half-open range of values.

3. Functions `average`, `variance`, and `std_dev` return the average, variance, and standard deviation of a half-open range of values, respectively.

   The average or *mean* value is computed as:

$$\mu = \frac{\sum_{k=0}^{n-1} x_k}{n}$$

   where x represents an array of n values with

$$\sum_{k=0}^{n-1} x_k = x_0 + x_1 + x_2 + \cdots + x_{n-1}.$$

   The variance σ^2 for a set of data values stored in array x can be computed using the equation:

$$\sigma^2 = \frac{\sum_{k=0}^{n-1} (x_k - \mu)^2}{n-1}$$

   The standard deviation σ for a set of data values is defined as the square root of the data set's variance:

$$\sigma = \sqrt{\sigma^2}$$

   > ***Here, we've an opportunity to apply the DRY principle; clearly, function `std_dev` can simply call function `variance` as the operand to the standard library function `sqrt`.***

4. Function `median` returns the value in the middle of a group of sorted values. If there are an odd number of values, the median is the value in the middle; if there are an even number of values, the median is the average of the values in the two middle positions. For example, the median of values $\{1, 6, 18, 39, 86\}$ is the middle value 18; the median of values $\{1, 6, 18, 39, 86, 91\}$ is the average of the two middle values, or $(18 + 39)/2$, or 28.5. Assume that a group of sorted values are stored in an array and that n contains the number of values in the array. If n is odd, then the middle value's subscript can be represented by $\lfloor n/2 \rfloor$, as in $\lfloor 5/2 \rfloor$, which is 2. If n is even, then the subscripts of the two middle values can be represented by $\lfloor n/2 \rfloor - 1$ and $\lfloor n/2 \rfloor$, as in $\lfloor 6/2 \rfloor - 1$ and $\lfloor 6/2 \rfloor$, which are 2 and 3, respectively. This function will need a sorted sequence of values and will call the `selection_sort` function [explained below] to sort the values in the array whose first value is pointed to by parameter `base`.

5. Function `selection_sort` will implement the selection sort algorithm to sort in *ascending order* `size` number of values of an array whose first value is pointed to by parameter `base`.

   > ***Your submission must not use C standard library function `qsort` to sort values in array. Instead, you must use the selection sort algorithm described in this handout to roll out your own function to sort numerical values in an array.***

6. Function `ltr_grade_pctg` has the declaration:

```c
void ltr_grade_pctg(double const *begin, double const *end,
                    double *ltr_grades);
```

Each value in the half-open range specified by parameters `begin` and `end` represents a numerical grade in the range from 0 to 100. A value in range $[0, 100]$ is mapped to a letter grade using the following algorithm:

Value v	Letter Grade	Subscript
$v \geq 90$	A	A_GRADE
$80 \leq v < 90$	B	B_GRADE
$70 \leq v < 80$	C	C_GRADE
$60 \leq v < 70$	D	D_GRADE
$v < 60$	F	F_GRADE

Function `ltr_grade_pctg` will determine the percentage of values in the half-open range that map to a letter grade and writes these percentages to the array whose first element is specified by parameter `ltr_grades`.

Define an *anonymous* enumeration type with enumeration constants in the order `A_GRADE`, `B_GRADE`, `C_GRADE`, `D_GRADE`, `F_GRADE`, and `TOT_GRADE`. These enumeration constants must have default values starting from 0 [for `A_GRADE`] with subsequent constants having a value one larger than the previous constant. Therefore constant `TOT_GRADE` will have value 5 [that is, enumeration constant `TOT_GRADE` will specify the total number of letter grades that numerical grades must be mapped to]. This enumeration type must be defined in header file q.h since both your definition of function `ltr_grade_pctg` in source file q.c and the client's use of the letter grade percentages in qdriver.c will have to reference these enumeration constants.

How are the enumeration constants used? Enumeration constants `A_GRADE` through `F_GRADE` specify the subscript of the corresponding letter grade A through F which can then be used for referencing the array whose first element is pointed to by parameter `ltr_grades`. Study the use case of this function in the driver to avoid confusion about the purpose and use of this function.

Header and source files

As usual, you'll define the enumeration type and function declarations in q.h.

> ***Since neither the enumeration constants nor the function declarations make references to names declared in the C standard library, your implementation of header file q.h must not include any header files. Why? Simply because such includes are completely unnecessary and will only lead to wasteful copying and pasting of text by the preprocessor when clients such as qdriver.c include your header file q.h.***

Define the functions declared in q.h in source file q.c. You must include all the necessary C standard library headers files required to define the function. The only caveat is that q.c must not contain any references to C standard library function `qsort`.

Download driver source file qdriver.c, a *makefile* makefile, input files containing course grades, and corresponding [correct] output files. Use the strategy of providing *stub* functions to establish and verify linkage between source files qdriver.c and q.c, and implementing and verifying the correct behavior of one function at a time. Build executable q.out using makefile:

```
$ make
```

and use command-line parameters to specify the input and output file names to the program:

```
$ ./q.out grades1.txt myoutput1.txt
```

Compare the output from your implementation with the correct implementation in output1.txt using diff:

```
$ diff -y --strip-trailing-cr --suppress-common-lines myoutput1.txt output1.txt
```

Repeat the process for the other input files.

File-level and Function-level documentation

Every source and header file you submit *must* contain file-level documentation blocks whose purpose is to provide human readers [yourself and other programmers] useful information about the purpose of this source file at some later point of time

Every function that you declare in a header file [and define in a corresponding source file] must contain a function-level documentation block.

> ***Don't copy and paste documentation blocks from previous assignments. Annoyed graders will definitely subtract grades to the full extent specified in the rubrics below when they detect such copy-and-paste scenarios.***

Submission and automatic evaluation

1. In the course web page, click on the submission page to submit the necessary files.
2. Read the following rubrics to maximize your grade. Your submission will receive:
    1. F grade if your submission doesn't compile with the full suite of gcc options [shown above].
    2. F grade if your submission doesn't link to create an executable.
    3. $A+$ grade if the submission's output matches the correct output. Otherwise, a proportional grade is assigned based on how many incorrect results were generated by your submission.
    4. A deduction of one letter grade for each missing documentation block. Every submitted file must have one file-level documentation block. Every function that you declare in a header file must provide a function-level documentation block. A teaching assistant will physically read submitted source files to ensure that these documentation blocks are authored correctly. Each missing block will result in a deduction of a letter grade. For example, if the automatic grader gave your submission an $A+$ grade and the three documentation blocks are missing, your grade will be later reduced from $A+$ to $B+$. Another example: if the automatic grade gave your submission a C grade and the three documentation blocks are missing, your grade will be later reduced from C to F.

Selection Sort Algorithm

One of the most common applications in computer science is sorting - the process through which data are arranged according to their values. We're surrounded by data. If data were not ordered, we would spend hours trying to find a single piece of information. Imagine the difficulty of finding someone's telephone number in your cellphone's contact list that was not ordered! In this document, we present a sorting algorithm called the selection sort that is simple to understand and straightforward to code in a function.

Although the type of elements in the array is of no significance to the algorithm itself, assume an array whose elements are assigned arbitrary integral values. The selection sort algorithm begins by finding the minimum value and exchanging it with the value in the first position in the array. Then the algorithm finds the minimum value beginning with the second element, and it exchanges this minimum with the second element. This process continues until reaching the next-to-last element, which is compared with the last element; the values are exchanged if they are out of order. At this point, the entire array of values will be in ascending order. This process is illustrated in the following sequences:

`Original order:`

5	3	12	8	1	9

`Exchange minimum with value in subscript 0:`

1	3	12	8	5	9

`Exchange next minimum with value in subscript 1:`

1	3	12	8	5	9

`Exchange next minimum with value in subscript 2:`

1	3	5	8	12	9

`Exchange next minimum with value in subscript 3:`

1	3	5	8	12	9

`Exchange next minimum with value in subscript 4:`

1	3	5	8	9	12

`Array values are now in ascending order:`

1	3	5	8	9	12

A hand implementation of the algorithm for a deck of cards can be visualized here. An algorithm for the selection sort can be designed like this:

Algorithm Selection Sort

Input: Array A of n elements
Output: Array A sorted in ascending order

```
1. $i = 0$
2. while $(i <= n - 2)$
3.    find $min$ : the index of smallest element in unsorted subarray $A[i]$ through $A[n-1]$
4.    if $i \neq min$
5.        swap$(A[i], A[min])$
6.    $i = i + 1$
7. endwhile
```