

## Lab7

**Deadline: 11:59 PM on Nov 10**

### Objectives:

In this lab, you will write a **C program** that manages **student information** and performs operations such as **sorting students** using the MergeSort algorithm and **filtering students** based on their average grades. The goal is to learn how to manage complex data, implement sorting algorithms, and work with file input and output.

## Lab Instructions

### 1. Lab Structure

1. You will use:

- **Structures** to store student data.
- **MergeSort** to sort students based on their **last name, first name, student number, and grades**.
- **File I/O** to read and write student data.
- **Command-line arguments** to handle user input.
- A **Makefile** to compile and clean your code.

2. Create the following files:

- Makefile
- main.c
- student.c
- student.h
- input.txt (sample input file)

### 2. Requirements

Each student will have the following attributes:

- **Last Name:** A string representing the student's last name.
- **First Name:** A string representing the student's first name.

- **Student Number:** In the format A1234567.
- **Midterm Grade:** An integer between 0 and 100.
- **Final Grade:** An integer between 0 and 100.

The program will:

- Sort students by last name (and other fields in case of ties).
- Filter students based on their average grade.
- Read input data from a file.
- Write filtered student data to an output file.

### 3. Program Flow

1. **Compile the Code:** Use the provided **Makefile** to compile your code.
2. **Run the Program with Command-Line Arguments:** The program takes **3 command-line arguments**:

`./lab7 <input_file> <output_file> <filter_option>`

- **<input\_file>:** The name of the input file containing student data.
- **<output\_file>:** The name of the output file where filtered data will be saved.
- **<filter\_option>:** An integer (1-5) representing the grade filter:
  1. Average > 90%
  2.  $80\% \leq \text{Average} \leq 90\%$
  3.  $70\% \leq \text{Average} < 80\%$
  4.  $60\% \leq \text{Average} < 70\%$
  5. Average < 60%

### 4. Files to Create

#### a. student.h

Header Guard:

```
#ifndef STUDENT_H
#define STUDENT_H
```

Prevents the contents of the student.h file from being included multiple times, which could cause redefinition errors.

If STUDENT\_H has not been defined yet, the contents between #ifndef and #endif will be included. If it is already defined, the contents are skipped.

### **Defining the Student Structure**

```
typedef struct {  
    char last_name[50];  
    char first_name[50];  
    char student_number[10];  
    int midterm_grade;  
    int final_grade;  
} Student;
```

Defines a structure named Student to store relevant information about a student.

#### **Attributes:**

**last\_name:** The student's last name, stored as a string with a maximum size of 50 characters.

**first\_name:** The student's first name, stored as a string with a maximum size of 50 characters.

**student\_number:** A 10-character string for the student number, such as "A1234567".

**midterm\_grade:** The student's midterm exam grade as an integer between 0 and 100.

**final\_grade:** The student's final exam grade as an integer between 0 and 100.

### **Function Prototypes**

```
int compare_students(const Student *a, const Student *b);  
void merge_sort(Student students[], int left, int right);  
float calculate_average(const Student *student);  
int read_students(const char *filename, Student students[], int max_students);  
void write_to_file(const char *filename, const Student students[], int count, int option);
```

The **compare\_students()** function compares two Student structures by following a specific order of criteria. It first compares the students based on their **last names**. If the last names are identical, it compares their **first names**. If the first names are also the same, it proceeds to compare their **student numbers**. If the student numbers are identical, it then compares the **midterm grades**. Finally, if all previous criteria are the same, it compares the **final grades**. This function is used in the sorting algorithm to determine the correct order of students in the list.

The **merge\_sort()** function sorts an array of Student structures using the MergeSort algorithm based on the criteria in compare\_students().

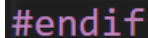
The **Calculate\_average** function calculates the **average grade** of a student based on the formula:

$$Average = \frac{Midterm\ Grade + Final\ Grade}{2}$$

The **read\_students()** function reads student data from a specified input file and stores it in an array of Student structures. It takes three parameters: filename, which is the name of the input file; students, an array to hold the student data; and max\_students, the maximum number of students to read. The function returns the total number of students successfully read from the file.

The **write\_to\_file()** function writes filtered student data to an output file based on the selected filter option. The filter options are: average greater than 90%, between 80% and 90%, between 70% and 80%, between 60% and 70%, or less than 60%. It takes four parameters: filename, the name of the output file; students, an array of student records to filter and write; count, the total number of students in the array; and option, the filtering option selected by the user.

## Ending the Header Guard



```
#endif
```

Marks the end of the header guard to prevent multiple inclusions of this header file.

### **b. Student.c**

Let's now implement various functions to manage student data. It provides operations to compare students, sort them using **MergeSort**, calculate their average grades, read data from an input file, and write filtered data to an output file.

First, we want to create a function called `compare_students()` that compares two students based on several attributes. This function is essential for ensuring that our sorting algorithm can properly order students according to the specified criteria.

We aim to compare two students by following a **priority order**:

1. **Last Name:** The students will first be compared by their last names using `strcmp()`. If the last names are different, the function will return the result immediately.
2. **First Name:** If the last names are the same, we move on to compare their first names using `strcmp()`.
3. **Student Number:** If both names are identical, the student numbers will be compared (`strcmp()`).
4. **Midterm Grade:** If the names and numbers are the same, the midterm grades will be checked.
5. **Final Grade:** If all other criteria are the same, the final grades will determine the order.

Next, we want to create a **`merge()`** function as part of the MergeSort algorithm. The purpose of this function is to merge two sorted halves of an array into a single sorted array. The two halves are temporarily stored in separate arrays, which ensures that the merge operation maintains the correct order.

```
// Compare two students based on multiple criteria
int compare_students(const Student *a, const Student *b) {
    int cmp = strcmp(a->last_name, b->last_name); // Compare last names
    if (cmp != 0) return cmp;                      // Return if not equal

    cmp = strcmp(a->first_name, b->first_name);    // Compare first names
    if (cmp != 0) return cmp;                      // Return if not equal

    cmp = strcmp(a->student_number, b->student_number); // Compare student numbers
    if (cmp != 0) return cmp;                      // Return if not equal

    if (a->midterm_grade != b->midterm_grade)      // Compare midterm grades
        return a->midterm_grade - b->midterm_grade; // Return difference

    return a->final_grade - b->final_grade; // Compare final grades
}
```

The size of each half is calculated using  $\text{int } n1 = \text{mid} - \text{left} + 1$ ; for the left half and  $\text{int } n2 = \text{right} - \text{mid}$ ; for the right half, where  $n1$  and  $n2$  represent the sizes of the left and right halves of the array, respectively.

Memory for the temporary arrays is allocated using `Student *L = (Student *)malloc(n1 * sizeof(Student));` for the left half and `Student *R = (Student *)malloc(n2 * sizeof(Student));` for the right half. These arrays, L and R, are used to temporarily store the elements of the left and right halves, respectively.

The elements from the original array are copied into two temporary arrays, L and R. The first temporary array, L, stores the elements from the left half of the original array, while the second array, R, holds the elements from the right half. This separation allows the two halves to be processed independently during the merging phase.

The two halves are merged back into the original array by comparing their elements using the **compare\_students()** function. The smaller element from either half is placed back into the original array, ensuring the correct order is maintained. After placing the element, the corresponding index is incremented to continue the comparison process. This merging continues until all elements from both halves have been processed.

```
void merge(Student students[], int left, int mid, int right) {
    int n1 = mid - left + 1; // Size of the left half
    int n2 = right - mid;    // Size of the right half

    // Allocate memory for temporary arrays
    Student *L = (Student *)malloc(n1 * sizeof(Student));
    Student *R = (Student *)malloc(n2 * sizeof(Student));

    // Copy data into the temporary arrays
    for (int i = 0; i < n1; i++)
        L[i] = students[left + i];
    for (int j = 0; j < n2; j++)
        R[j] = students[mid + 1 + j];

    // Merge the two halves back into the original array
    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) {
        if (compare_students(&L[i], &R[j]) <= 0) {
            students[k++] = L[i++];
        } else {
            students[k++] = R[j++];
        }
    }
}
```

If any elements remain in either half after all comparisons are complete, they are directly copied into the original array. First, the remaining elements from the left half are copied, followed by any remaining elements from the right half. This ensures that all elements are included in the final merged array.

After the merge is complete, the dynamically allocated memory for the temporary arrays is released to prevent memory leaks.

```
    // Copy remaining elements from the left half
    while (i < n1) {
        students[k++] = L[i++];
    }

    // Copy remaining elements from the right half
    while (j < n2) {
        students[k++] = R[j++];
    }

    // Free the allocated memory
    free(L);
    free(R);
}
```

This function implements the MergeSort algorithm to recursively divide and sort the student array. The array is divided into two halves by calculating the middle index, ensuring that it is split evenly for the sorting process. The **merge\_sort()** function is then called recursively on the left half of the array to sort it. Similarly, it is also called on the right half to sort that portion. Once both halves are sorted, the **merge()** function merges them back into the original array in the correct order. This process continues recursively until the entire array is fully sorted.

```
void merge_sort(Student students[], int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2; // Calculate mid-point
        merge_sort(students, left, mid);     // Sort the left half
        merge_sort(students, mid + 1, right); // Sort the right half
        merge(students, left, mid, right);    // Merge the two halves
    }
}
```

The **calculate\_average()** function computes the average grade for a student based on their midterm and final grades. The average is calculated using the formula:



The function returns the average as a float to allow for decimal precision, ensuring accurate calculations. Together, these functions work seamlessly: **merge\_sort()** ensures that the student data is correctly sorted, while **calculate\_average()** provides the necessary grading information for filtering and generating the final output.

```
// Calculate the average grade of a student
float calculate_average(const Student *student) {
    return (student->midterm_grade + student->final_grade) / 2.0;
}
```

The **read\_students()** function reads student data from an input file and stores it in an array of Student structures. It begins by attempting to open the specified file in read mode using **fopen()**. If the file cannot be opened, an error message is printed, and the function returns 0 to indicate that no students were read. A counter, **count**, is initialized to track the number of students successfully read. The function then uses a while loop to read the data line by line, ensuring that the total number of students does not exceed the maximum allowed (**max\_students**). Each line of the file is parsed using **fscanf()**, which extracts the student's last name, first name, student number, midterm grade, and final grade. If all five fields are successfully read, the counter is incremented. Once all the data is processed, the file is closed using **fclose()** to free resources, and the function returns the total number of students read, as indicated by the **count** variable.

```
// Read student data from an input file
int read_students(const char *filename, Student students[], int max_students) {
    FILE *file = fopen(filename, "r"); // Open the file for reading
    if (!file) {                        // Check if the file opened successfully
        printf("Error: Could not open file %s\n", filename);
        return 0;
    }

    int count = 0; // Counter for the number of students
    // Read data line by line from the file
    while (count < max_students &&
           fscanf(file, "%s %s %s %d %d",
                  students[count].last_name,
                  students[count].first_name,
                  students[count].student_number,
                  &students[count].midterm_grade,
                  &students[count].final_grade) == 5) {
        count++;
    }

    fclose(file); // Close the file
    return count; // Return the number of students read
}
```

The **write\_to\_file()** function writes filtered student data to an output file based on the selected filtering option. It begins by attempting to open the specified file in write mode using **fopen()**. If the file cannot be opened, an error message is displayed, and the function returns without further execution. The function then iterates through the array of students, calculating the average grade for each student using the **calculate\_average()** function. A write flag is initialized to 0 for each student, and a switch statement is used to determine whether the student meets the criteria based on the selected option. If the student's average falls within the specified range, the write flag is set to 1.

If the write flag is set, the student's data is written to the file using **fprintf()**, including their last name, first name, student number, midterm grade, final grade, and calculated average. This ensures that only students matching the selected criteria are written to the output file. After processing all students, the file is closed using **fclose()** to free resources. This function ensures that the filtered student data is properly saved to the specified output file.

```
void write_to_file(const char *filename, const Student students[], int count, int option) {
    FILE *file = fopen(filename, "w"); // Open the file for writing
    if (!file) {                        // Check if the file opened successfully
        printf("Error: Could not open file %s\n", filename);
        return;
    }

    // Loop through each student and filter based on the selected option
    for (int i = 0; i < count; i++) {
        float avg = calculate_average(&students[i]); // Calculate the average grade
        int write = 0; // Initialize the write flag

        // Determine if the student meets the criteria for the selected option
        switch (option) {
            case 1: if (avg > 90) write = 1; break;
            case 2: if (avg >= 80 && avg <= 90) write = 1; break;
            case 3: if (avg >= 70 && avg < 80) write = 1; break;
            case 4: if (avg >= 60 && avg < 70) write = 1; break;
            case 5: if (avg < 60) write = 1; break;
        }

        // Write the student's data if it meets the criteria
        if (write) {
            fprintf(file, "%s %s %s %d %d %.2f\n",
                    students[i].last_name, students[i].first_name,
                    students[i].student_number,
                    students[i].midterm_grade, students[i].final_grade, avg);
        }
    }

    fclose(file); // Close the output file
}
```

### c. Main.c

#### 1. Including Necessary Headers

- `#include <stdio.h>` provides access to input/output functions like `printf()` and `fopen()`.
- `#include <stdlib.h>` allows the use of functions like `atoi()` and `malloc()`.
- `"student.h"` is included to access the `Student` structure and related functions such as `read_students()`, `merge_sort()`, and `write_to_file()`.

```
#include <stdio.h>
#include <stdlib.h>
#include "student.h"
```

## 2. Main Function Signature and Argument Validation

- This function starts the program and receives **command-line arguments**.
- It checks if **exactly three arguments** (besides the program name) are provided: the **input file**, the **output file**, and the **filter option**.
- If the number of arguments is incorrect, it prints the usage instructions and exits with an error.

```
int main(int argc, char *argv[]) {  
    if (argc != 4) { // Check if the correct number of arguments is provided  
        printf("Usage: %s <input file> <output file> <option>\n", argv[0]);  
        return 1;  
    }  
}
```

## 3. Extracting and Converting Command-line Arguments

- The **input file** and **output file names** are retrieved from the command-line arguments.
- The **filter option** is converted from a string to an integer using `atoi()`.

```
const char *input_file = argv[1];  
const char *output_file = argv[2];  
int option = atoi(argv[3]);
```

## 4. Validating the Filter Option

```
if (option < 1 || option > 5) {  
    printf("Error: Option must be between 1 and 5.\n");  
    return 1;  
}
```

- This block ensures that the filter option is within the **valid range (1 to 5)**.
- If the option is invalid, it prints an error message and exits the program.

## 5. Reading Student Data from the Input File

```
Student students[100];  
int student_count = read_students(input_file, students, 100);
```

- An array of Student structures is declared to hold up to **100 students**.

- The `read_students()` function is called to **read student data** from the specified input file and store it in the `students` array.
- The function returns the **number of students** read, which is stored in `student_count`.

## 6. Checking if Any Students Were Read

- If no students were successfully read from the input file, an error message is printed, and the program exits.

```
if (student_count == 0) { // Check if any students were read
    printf("No students found in the input file.\n");
    return 1;
}
```

## 7. Sorting Students Using MergeSort

- The `merge_sort()` function is used to **sort the students** based on multiple attributes (last name, first name, student number, and grades) in ascending order.

```
merge_sort(students, 0, student_count - 1);
```

## 7. Writing the Sorted and Filtered Data to the Output File

- The `write_to_file()` function saves the **filtered student data** to the specified output file based on the selected option.

```
write_to_file(output_file, students, student_count, option);
```

## 9. Confirming Successful Write Operation & Exiting the Program Successfully

- A message is printed to confirm that the data was successfully written to the output file.
- The program returns 0, indicating that it completed successfully without errors.

```
printf("Data successfully written to %s\n", output_file); // Confirm successful write
return 0; // Exit the program successfully
```

### 5. Bonus Marks: Implement QuickSort [2 marks]

For bonus marks, replace MergeSort with QuickSort.

**Hint:**

- Use partitioning logic similar to how MergeSort divides arrays.
- A pivot element helps divide the array into two halves.

### Submission Instructions

1. Submit the following files to the learning platform before the deadline:
  - main.c
  - student.c
  - student.h
  - Makefile
2. Ensure that your code compiles and runs successfully using the provided Makefile.
3. Late submissions will not be accepted, so plan ahead and submit all required files on time.
4. Bonus Marks Opportunity: If you replace MergeSort with QuickSort, you can earn up to 2 bonus marks, which will not affect the total score but will be added to your overall performance evaluation.

### Grading Criteria for the Lab (Out of 10 Marks)

Criteria	Points
<b>Completion of the Code:</b> All required functionalities are correctly implemented, including reading students from a file, sorting with MergeSort, calculating averages, filtering data, and writing to an output file.	<b>8 marks</b>
<b>Makefile Creation:</b> A working Makefile is provided, allowing the code to compile and run without errors.	<b>2 marks</b>

**Bonus Marks (will be adjusted later on in overall course weightage)**

Bonus Criteria	Points
Implement QuickSort: Replace MergeSort with QuickSort to sort the students.	2 bonus marks