Project #3

Due Wednesday by 11:59pm **Points** 100 **Available** until Sep 21 at 2am

Task #1 - Modify project 1 (50 points)

Task

Modify your project 1 solution to use arrays to represent 4-hour, per day, and per week rates for the four equipments, and update the calculation using the arrays. The program should have one calculation for all equipments. All other requirements are the same.

https://usflearn.instructure.com/courses/1829998/assignments/14363017

If you have already done the above in your project 1 submission, you can submit the same program for this task.

Submission instructions

1. Develop and test your program on the student cluster
   1. To compile your program, run: **gcc -std=c99 -Wall your_program.c**
   2. To execute your program, run: **./a.out**
2. Name your program **project1_rental.c**
   1. To rename your program, run: **mv your_program.c project3_rental.c**
3. Test your program with the shell script on Unix: try_project3_rental
   1. upload the script *zip* file to the student cluster in the same directory as your program.
   2. run: **$ unzip try_project3_rental.zip**
   3. move your program to the same folder as try_project1_rental file
   4. run: **chmod +x try_project3_rental**
   5. run: **./try_project3_rental**
4. Download the program from student cluster and submit it on **Canvas->Gradescope**. Make sure you submit a file with the correct name!
5. You can submit your program as many times as needed before the due date. Gradescope will indicate the test cases with incorrect output, if any exists.
6. Please note that GradeScope is used for testing some of the functionalities. It's part of the grading process. The grade you received on GradeScope only reflects the results of your program against the test cases, it is not your final project grade. Projects will be manually graded after running the test cases on GradeScope.

Task #2 (50 points)

Task

To save energy, the owner of a local shopping mall installed a sensor that verifies when there is someone on the escalator. When the sensor detects no presence, the escalator is deactivated, this way saving energy until the next person arrives. Write a program that calculates the number of active seconds that an escalator was active. The escalator is initially inactive. The amount of time that a person takes to go from the beginning to the end of the escalator is 10 seconds. The input are the number of people, and the arrival times of each person at the escalator.

In example #1 below, 2 people used the escalator, first person arrived at 12th second, and second person arrived at 25th second. The escalator is active for 20 seconds, 12th to 22 th, and 25th to 35th seconds.

In example #2 below, 3 people used the escalator, first person arrived at 12th second, and second person arrived at 15th second, and the third person arrived at 20th second. The escalator is active for 18 seconds, 12th to 30th seconds.

Requirements

1. Follow the format of the examples below.
2. The program should include the following function:

   int active_seconds(int arrival[], int n)

   The function calcuates the total number of active seconds. Array arrival[] represents the arrival time of each person, n is the length of array arrival, representing the number of people.
3. In the main function, call the **active_seconds** function and display the result.

Examples (your program must follow this format precisely)

Example #1

```
Enter number of people: 2
Enter arrival times: 12 25
Active seconds: 20
```

Example #2

```
Enter number of people: 3
Enter arrival times: 12 15 20
Active seconds: 18
```

Example #3

```
Enter number of people: 5
Enter arrival times: 12 15 20 34 41
Active seconds: 35
```

Submission instructions

1. Develop and test your program on the student cluster
   A. To compile your program, run: **$ gcc -std=c99 -Wall your_program.c**
   B. To execute your program, run: **$ ./a.out**
2. Name your program **project3_escalator.c**
   A. To rename your program, run: **$ mv your_program.c project3_escalator.c**
3. Test your program with the shell script on Unix: try_project3_escalator
   A. Upload the script to the same directory as your program.
   B. Run: **$ chmod +x try_project3_escalator**
   C. Run: **$ ./try_project3_escalator**
4. Download the program from student cluster and submit it on **Canvas->Gradescope**. Make sure you submit a file with the correct name!
5. You can submit your program as many times as needed before the due date. Gradescope will indicate the test cases with incorrect output, if any exists.

General instructions

Grading

Task #1

Total points: 50

- A program that does not compile will result in a zero.
- Runtime error and compilation warning 3 points

  1 points off, if a warning is present.

  3 points off, if multiple warnings are present.
- Commenting and style 8 points

  1 point off for not putting name and description at the beginning

  2 to 4 points off if the code didn't have clarifying comments.

  1 to 3 points off (depending on how much indentation is off) if the program is not indented properly.
- Functionality

  3 points off for each incorrect test case

  5 points off if array is not used to represent 4-hour, per day, or per week rates

  10 points off if array are not used in the calculation of cost

Task #2

Total points: 50

- A program that does not compile will result in a zero.
- Runtime error and compilation warning 3 points

  1 points off, if a warning is present.

  3 points off, if multiple warnings are present.
- Commenting and style 8 points

  1 point off for not putting name and description at the beginning

  2 to 4 points off if the code didn't have clarifying comments.

  1 to 3 points off (depending on how much indentation is off) if the program is not indented properly.
- Functionality

  8 points off for each incorrect test case

Programming Style Guidelines

The major purpose of programming style guidelines is to make programs easy to read and understand. Good programming style helps make it possible for a person knowledgeable in the application area to quickly read a program and understand how it works.

- Your program should begin with a comment that briefly summarizes what it does. This comment should also include your name.
- In most cases, a function should have a brief comment above its definition describing what it does. Other than that, comments should be written only needed in order for a reader to understand what is happening.
- Variable names and function names should be sufficiently descriptive that a knowledgeable reader can easily understand what the variable means and what the function does. If this is not possible, comments should be added to make the meaning clear.
- Use consistent indentation to emphasize block structure.
- Full line comments inside function bodies should conform to the indentation of the code where they appear.
- Macro definitions (**#define**) should be used for defining symbolic names for numeric constants. For example: **#define PI 3.141592**
- Use names of moderate length for variables. Most names should be between 2 and 12 letters long.
- Use underscores to make compound names easier to read: **tot_vol** and **total_volumn** are clearer than **totalvolumn**.