Project 5

Due July 11th at 11:59 PM

You are a business owner and you keep information on your employees stored in a text file. Each line contains the name of an employee, their employee id, the number of hours they worked during this pay period, and the hourly rate at which they are paid. Write a program which

- Gets the name of input and output files from command line arguments
- Reads the file using `fscanf` and stores the information in an array of employee structs
- For each employee, calculates the net income and taxes withheld, storing it in their corresponding struct within the array
- For each employee, writes the employee name, id, net income, and taxes withheld to the output file using `fprintf`

An employee struct must have the following members:

- name — string
- id — int
- hours worked — double
- hourly rate — double
- net income — double
- taxes withheld — double

An example execution of the program:

```
programName employees.txt payday.txt
```

**Details**

1. Use `fscanf` and `fprintf` to read and write data

2. You should assume that each line of the input file has the following format:

   ```
   employee_last_name id hours_worked hourly_rate
   ```

   Example:

   ```
   rippetoe 324 23 12.50
   ```

3. Each line of the output file ***must*** have the following comma separated format:

   ```
   employee_last_name,id,net_income,taxes_whitheld
   ```

   Example:

```
rippetoe,324,244.38,43.12
```

4. A skeleton program is provided that is broken up into the following parts
   1. `employeePayroll.c`
   2. `employee.c` and `employee.h`

   You must create a makefile that builds your program. It should contain the following rules
   1. Build executable program payroll by linking `employeePayroll.o` and `employee.o` (this should be the first rule!)
   2. Build `employeePayroll.o` by compiling `employeePayroll.c`
   3. Build `employee.o` by compiling `employee.c`

   You should use the `-Wall` option for warnings. If you want to use GDB to debug, include the `-g` option. Use the slides and the makefile from previous project as examples.

**Extra Credit** (10pts)

Sort the employee array according to name in ascending order before writing to the output file. The recursive `selectionSort` function from earlier in the semester has been provided at the bottom of `employeePayroll.c`. Uncomment this code, adapt it to the project, and call it in `main()`. See the example output for what you should expect in comparison to not performing the sort. You can assume that employee names are made up entirely of lowercase letters

**Suggestions**

1. You can use a diff tool to compare the file output from your program to the expected, correct output. There is a `diff` command available on linux that will show you which lines in your file are different

   ```
   diff file1 file2
   ```

   will show you the lines which are different between file1 and file2. If they are identical, the command will appear to have done nothing won't do anything.

2. START ASAP AND ASK QUESTIONS.

**Before you submit:**

1. **Make sure you follow the rules for the input/output files**

2. Make sure your program begins with a comment that begins with your name

3. Your makefile should compile with `c99` and the `-Wall` flag. `-Wall` shows the warnings by the compiler. Be sure it compiles on the student cluster with no errors and no warnings.

4. Test your program thoroughly with multiple different input files. Use the diff command to compare your output with the expected output

5. Place all of your source files in a zip file. The files should be in the root of the zip; not in a folder inside the zip. You can zip them on the student cluster with the following command

```
zip project5 *.c *.h makefile
```

   This will place the makefile and all of the .c and .h files in your current directory into project5.zip. Submit project5.zip

**Grading Details**

1. A program that does not compile may result in a zero.
2. Programs which generate compiler warnings will lose 5%
3. Commenting and style worth 10%
4. Functionality worth 90%

**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.

1. Your program should begin with a comment that briefly summarizes what it does. This comment should also include your name.
2. 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.
3. Use consistent indentation to emphasize block structure.
4. Macro definitions (`#define`) should be used for defining symbolic names for numeric constants. For example: `#define PI 3.141592`
5. Use underscores or camel case notation to make compound names easier to read: `total_volume` or `totalVolume` is clearer than `totalvolume`.