CS 240: Programming in C
Fall 2020
Homework 5

Prof. Turkstra

Due October 7, 2020 9:00 PM

1 Goals

In this assignment, you will learn about random file access, reading and writing to a file in binary format, and more about structures. We'll also get a small peek at using an enumerated data type. You'll also (finally) start using assertion checks.

2 Getting Started

You know what to do!

3 The Big Idea

At this point you understand how structures work, what they're used for, and how to manipulate arrays of them. This time, we're going to make things a bit more interesting by using a structure with many fields in it and manipulating them in a binary file instead of an array.

Binary files are a great way to store structs, because the programmer doesn't have to deal with the hassle of coming up with tricky format strings for `scanf()`/`printf()`. All you need is a single `fwrite()`/`fread()` at the correct offset and you have the whole struct filled for you! The trade-off, of course, is that these files are not human-readable.

For this homework, we're going to create a structure that will be used to represent a professor at Purdue University. There are lots of pieces of information for a professor that should be kept together in one group. The information to be maintained will be as follows:

The information to be maintained will be as follows:

- **ID Number**: this is an integer
- **First Name**: this is an array of 40 characters
- **Last Name**: this is an array of 40 characters
- **Age**: this is an integer
- **Funding**: this is a float
- **Gender**: this is an enum type corresponding to the professor's gender, which will be one of `female`, `male` or `other`.

- **Class**: this is a two dimensional array of characters. The first (row) dimension represents the day of the workweek (M - F). The second dimension (column) represents the hour of the day (8AM - 9PM). If the professor has class during that hour, the corresponding array entry contains the character "Y", and if the professor has no class during the hour, the array entry contains the character "N".

The structure should look like this (we combine it with a `typedef` so that we can refer to it with one name):

```
typedef struct {
  int             id_number;
  char            first_name[MAX_NAME_LEN];
  char            last_name[MAX_NAME_LEN];
  int             age;
  float           funding;
  enum gender_t   gender;
  char            class[N_DAYS][N_HOURS];
} professor_t;
```

4 The Assignment

You will be given a binary file representing Purdue University, which will contain a series of `professor_t` structs. Using this file, you will implement a number of functions that process data from the file, including some which modify the contents of the file itself.

Important note: These functions will take in the `FILE *` of a file that has already been opened. However, the position of the file pointer will not be altered by the test or main modules, meaning that each of your functions will start with the file pointer wherever the function called previously left it.

4.1 Functions You Will Write

You will write the following functions:

`professor_t read_professor(FILE *, int);`

This function will read the N^{th} professor record (where N is the second argument) from the file (the first argument), which has already been opened. The function should always return a `professor_t` structure.

If there is any kind of error, it should instead return `BAD_PROFESSOR`, which is a structure that has its fields set to known invalid values. Errors include trying to read beyond the length of the file or any situation where `fread()` fails.

This function should assert that the file pointer is non-NULL and that N is greater than or equal to 0.

`int write_professor(FILE *, professor_t, int);`

This function will write the given professor information to the specified (already open) file at the N^{th} position in the file.

For instance, if the third argument is 0, the structure will be written at the beginning of the file. If the third argument is 55, the structure will be written to the 55^{th} position in the file (i.e. at a byte offset of `55 * sizeof(professor_t)`).

If the operation is successful, the function should return `OK`. If an error occurs, including trying to seek beyond the end of the file, or if an `fwrite()` call fails, the function should return `WRITE_ERROR`.

~~This function should assert that the first argument is non-NULL and that the third argument is greater than or equal to 0. Furthermore, it should also assert that the information in the second argument is valid. A professor is valid if the following conditions hold:~~

1. ~~The `last_name` and `first_name` fields are not empty strings.~~
2. ~~The~~ `id_number` field is greater than 0, the `gender` field holds a valid enumerated value, the `age` field is greater than 0, the `funding` field is greater than 0, and all entries in the `class` array are either "Y" or "N".

`float compute_female_percent(FILE *);`

Determine the percentage of the professor who identify as female. Make sure your returned value is in the form of a decimal from 0 and 1.

You can assume the number of female professor is not zero.

~~This function should assert that the first argument is non-NULL.~~

`float total_funding_by_male(FILE *);`

Sum the funding for all male professors in the file and return it.

~~You can assume the number of male professor is not zero.~~

~~This function should assert that the first argument is non-NULL.~~

`int find_professor_by_name(FILE *, char *, char *);`

This function should search the file for a `professor_t` whose first and last names match the second and third arguments, respectively. Once found, the function should return the position of the professor in the file. If the professor cannot be found, the function should return `NO_PROFESSOR`.

If multiple entries are found, return the position of the first match.

~~This function should assert that the first argument is non-NULL, and the second and third arguments are not empty strings.~~

`int find_professor_by_id(FILE *, int);`

This function should search the file for a `professor_t` whose ID number matches the value of the second argument. The function should return the position of the professor in the file. ~~If the professor cannot be found, the function should return `NO_PROFESSOR`.~~

~~This function should assert that the first argument is non-NULL and the second argument is greater than zero.~~

`float average_funding_by_age(FILE *, int);`

This function should return the average funding among professors whose age is older than the value of the second argument. If no professors are older than the given age, the function should return `NO_PROFESSOR`.

~~This function should assert that the first argument is non-NULL and the second argument is greater than zero.~~

`int fund_student(FILE *, int);`

This function should search the file for a `professor_t` whose ID number matches the second argument. Once found, 10000.0 should be subtracted from the funding field. The record should then be written to the file at its original position.

If this operation would result in a negative funding, do not alter the amount, and instead return `NO_FUNDING`. If no professor matches the ID number, this function returns `NO_PROFESSOR`, otherwise, it returns `OK`.

This function should assert that the first argument is non-NULL and the second argument is greater than zero.

`int schedule_meeting(FILE *, int, int);`

This function will accept the `id_numbers` of two professors and find the earliest time in the week that both are available. ~~If either professor is not found, return `NO_PROFESSOR`~~. If there are no commonalities between the employee schedules, return `NO_OVERLAP`. Otherwise, the function should return an integer that represents the day of the week (M-F) times 100 plus the hour of the day (8-21). For instance, Monday at 2pm would be represented by 0*100 + 14 = 14. Tuesday at 4pm would be represented by 1*100 + 16 = 116.

This function should assert that the first argument is non-NULL and that the other arguments are greater than zero.

4.2 Input Files

We are going to be nice and open the files for you this time, so their (opened) pointers will be passed as parameters in each function.

Remember: these are binary files. You cannot simply inspect them using a text editor. It is possible to view their contents with something like `xxd`. It's probably not worth the trouble.

Instead, think carefully about the offset, and make sure you get it right. At least you don't have to think about format strings!

4.3 Header Files

We provide a header file, "hw5.h", for you. It contains prototypes for each of the functions that you will write as well as #definitions for the constants. You should not alter this file. We will

replace it with the original when grading.

4.4 Error Codes

- `OK`: no errors occurred
- `BAD_PROFESSOR`: a `professor_t` structure that indicates a failed operation for functions that return professors
- `NO_PROFESSOR`: the desired professor could not be located in the file
- `NO_OVERLAP`: there are no times common in the schedules of two professors
- `NO_FUNDING`: professor does not have enough funding to fund a student
- `WRITE_ERROR`: the desired write operation was unable to be completed

These Standard Rules Apply

- You may add any `#include`s you need to the top of your hw5.c file;
- You may not create any global variables other than those that are provided for you. Creation of additional global variables will impact your style grade;
- Do not look at anyone else's source code. Do not work with any other students.

Submission

To submit your program for grading, type:

```
$ make submit
```

In your hw5 directory. You can do this as often as you wish. We encourage you to submit your code as often as possible. Only your final submission will be graded.

5 Grading

The operation of your functions will be graded out of 100 points. The point breakdown will be determined by the test program.

The test program will be run many times when grading. It is your job to do the same. The lowest score will be your final grade.

This homework will also have a style grade based on 20 points, with 2 points deducted for each code standard violation found.

Your code must compile successfully using `-Wall -Werror -std=c99` to receive any credit. Code that does not compile will be assigned an automatic score of 0.

If your program crashes (e.g., segmentation fault) at any point during the testing process, your score will be reduced by 25 points.