CIS*2500 Assignment 1: GryphFlix

Due Wednesday January 27th 11:59pm
Weighting: 7%

1.0 Your Task

Congratulations! You have been hired as a developer for GryphCinemas in the creation and testing of their newest 'GryphFlix' movie rating software. As the sole software developer of this product, you will create a program that will allow ten (10) reviewers to rate three (3) movies. All data will be read in from external files and parsed into structs.

Each of the three movies will be rated by each of the ten judges on a binary scale:

- 'Y' or 'y' will represent the notion of recommending a movie
- 'N' or 'n' will represent the notion of not recommending a movie

The software will take the parsed reviews and determine the most "critical" reviewers as well as the highest-rated movie(s). As a means of futureproofing, this program will also predict how a movie will perform based on state-of-the-art (not really) algorithms.

This assignment stresses the importance of developing software that strictly conforms to specification—when you're in the industry, you'll face all sorts of strict clients.

You will write a .c file, `gryphflix.c` containing the function implementations detailed below as well as an accompanying header file, `gryphflix.h` containing the proper function prototypes, constants, and structures also detailed below.

Do not include a `main()` function in any of your submission(s)—grading will be based on the specified function definitions. You are still encouraged to make a robust testing file containing a `main()`. You are also free to create helper functions, though if they are used by the assignment functions below, include them in `gryphflix.c` and `gryphflix.h` so they are accessible to us during grading.

These constants must be defined, ideally go into your `gryphflix.h` file.

- `#define NUMBER_REVIEWERS 10`
  This will represent the number of reviewers.

- `#define NUMBER_MOVIES 3`
  This will represent the number of movies.

- `#define MAX_STR 50`
  This will represent the maximum length of a string.

- `#define MAX_WORDS 10`
  This will represent the max. number of words in a movie title.

```
struct reviewStruct {
    char reviewer[MAX_STR];
    int feedback[NUMBER_MOVIES];
}
```

This struct will represent a single review. The `reviewer` field is a string that will represent the name of the reviewer and the `feedback` field is an array of `int` that will represent each individual movie's rating where 1 represents 'Y' or 'y' and 0 represents 'N' or 'n'.

For example, if Jason recommends the first of the three movies but not the next two, the `reviewStruct` used would have `reviewer` = "Jason" and `feedback` = {1, 0, 0}

```
FILE *openFileForReading(char *fileName);
```

This function will take in a filename as a string and attempt to create a `FILE` pointer out of it. If the file opens successfully, a `FILE` pointer will be returned. Otherwise, `NULL`. You will only open the file for reading, not writing.

```
int readMovies(FILE *file, char movieNames[NUMBER_MOVIES][MAX_STR]);
```

This function will read movie titles from a `FILE` pointer variable named `file`, line by line, into the `movieNames` 2D array. The function will either return `1` (to indicate successful operation) or `0` to represent abnormal exit, which can be:

- Passing a `NULL` file pointer to the function
- Passing a file that has more lines than `NUMBER_MOVIES`
- Passing a file that has less lines than `NUMBER_MOVIES`

In normal operation, this function will read in exactly `NUMBER_MOVIES` lines.

For example, for this file:

```
Star Wars
Incredibles
Gone with the Wind
```

the `movieNames` array will contain the strings `"Star Wars"`, `"Incredibles"`, and `"Gone with the Wind"`. Make sure you test this thoroughly—this and the `readReviews` function represent the backbone of this application. Some implementations of line-by-line file reading may inadvertently introduce an extra newline (`'\n'`) in the string returned to the developer, which will mess up our testing. This is an error that can become "consistent" throughout the program (for example, if you mistakenly read the strings as {`"Star Wars\n"`, `"Incredibles\n"`, `"Gone with the Wind\n"`}, the rest of the program will likely run without error, only to fail the string comparison tests in grading. This is because the remaining functions will read and write the botched movie names and assume they are correct.

```
int readReviews(
    FILE *file, struct reviewStruct reviews[NUMBER_REVIEWERS]
);
```

This function will read movie reviews from a `FILE` pointer variable named `file`, line by line, into the `reviewStruct` array, `reviews`. Each line will be of the following form:

```
<name> <rating1> <rating2> <rating3>
```

Where each constituent part is separated by a space. For example, if Isabelle rated 'y', 'n', 'Y' for the three respective movies, her line would be:

```
Isabelle y n Y
```

It will be the responsibility of the programmer to parse these lines correctly. Reviewers' inputs are <u>case-insensitive</u> (in other words, to represent "Yes, I recommend this movie", the character could be 'Y' or 'y'. Likewise with 'N' and 'n'). Students may assume that each individual line is correct (i.e. no foreign characters/bad input) but are still responsible for regulating that a correct amount of lines are read, just like with `readMovies`.

As stated in the `struct` definition, in the `review_struct`'s `feedback` array, a 1 represents 'Y' or 'y' (i.e. "yes, I recommend this movie") and 0 represents 'N' or 'n' ("No, I do not recommend this movie").

This function will return `1` as a successful exit, and `0` in the event of:

- Passing a `NULL` file pointer
- Passing a file that has more lines than `NUMBER_REVIEWERS`
- Passing a that that has less lines than `NUMBER_REVIEWERS`

HINT You may find this built-in function named strtok on string tokenization useful. To read more on `strtok,` refer to textbook pages 620-621 of Chapter 23.

```
int getMostCriticalReviewers(
    struct reviewStruct reviews[NUMBER_REVIEWERS],
    char mostCriticalReviewers[NUMBER_REVIEWERS][MAX_STR]
);
```

This function will find a list of the most critical reviewers and store them in array `mostCriticalReviewers`. A critical reviewer is a reviewer who has the same amount of negative recommendations as the reviewer with the most negative recommendations. This function will return an `int` representing the number of critical reviewers and will store the names of each critical reviewer using the `mostCriticalReviewers` array parameter.

As the `reviews` array is used merely as an input (i.e. it will only be read from when writing to the `mostCriticalReviewers` list), you are not allowed to modify it. There are very few implementations of this function where you would want to do this, but this is still poor practice. As such, `reviews` will be made `const` (constant) for the compiler to warn you in the event of inadvertent modification.

For example, with the following file:

```
Ritu Y N y
Larry n n y
Dan y y y
Judi n y n
Eric y y n
Isabelle y n y
Manisha y n n
Terra Y Y N
Dora n y n
Nick n n y
```

the highest number of `n` reviews ("No, I do not recommend") for a given reviewer is 2. This is observed in the case of `Larry`, `Judi`, `Manisha`, `Dora`, and `Nick`, and their names would be written to the `mostCriticalReviewers` array in the original order as the file presents them. This means writing `{"Nick", "Dora", "Manisha", "Judi", "Larry"}` is incorrect and will result in lost marks.

```
int getMostRecommendedMovies(
    const char movies[NUMBER_MOVIES][MAX_STR],
    const struct reviewStruct reviews[NUMBER_REVIEWERS],
    char mostRecommendedMovies[NUMBER_REVIEWERS][MAX_STR]
);
```

This function will get a list of the most recommended movies. A recommended movie has the same amount of positive recommendations ('Y' or 'y') as the movie which has the most amount of positive recommendations. Like `getMostCriticalReviewers()`, this function will return an `int` representing the number of recommended movies as well as return the names of each recommended movie using `mostRecommendedMovies`.

In a similar vein of reasoning as `getMostCriticalReviewers()`, the `movies` and `reviews` arrays are used merely as input (i.e. they will only be read from when writing to the `mostRecommendedMovies` list), so you are not allowed to modify them. There are few reasons to (in this function) anyway.

To look at an example of this function, examine this file:

```
Ritu Y N y
Larry n n y
Dan y y y
Judi n y n
Eric y y n
Isabelle y n y
Manisha y n n
Terra Y Y N
Dora n y n
Nick n n y
```

of the three movies being reviewed (each being represented by a column in the file), the highest number of `'y'` reviews ("Yes, I recommend") is 6, and this occurs with the first movie (in other words, the first movie column of the three). The remaining two movies only have five `'y'` reviews each, meaning only the first movie is written to `mostRecommendedMovies`, which would be `"Star Wars"` given the original movies file. The function, as such, would return `1`.

```
int predictMovie(const char movie[MAX_STR]);
```

This function will predict the future performance of a movie's box office sales. This will be based solely off a proprietary *GryphCinemas*™ algorithm. If a movie title:

- **Has more than two words in it**, then sales will **decrease**
- **Has only one word**, then sales will **drastically decrease**
- **Has more than one word with an even number of characters**, sales **increase**
- **Has the same number of characters in each word**, sales **drastically increase**, as the title is "perfectly balanced as all things should be".

It is possible for more than one of these conditions to hold true (for example, "West Side Beef" has more than two words in it and has the same number of characters in each word, so there is a conflict between it decreasing and drastically increasing), so you will map these five predictions onto a five-point scale:

- **–2**: Box office sales will **drastically decrease**
- **–1**: Box office sales will **decrease**
- **0**: Box office sales will **remain unchanged**
- **1**: Box office sales will **increase**
- **2**: Box office sales will **drastically increase**

In the event of a conflict, you will combine the two different ratings into a single rating. For example, if the algorithm deduces that a given movie's sales will drastically increase (**2**) and decrease (**–1**), these two combined ratings reduce (**2 – 1 = 1**) to the movie's box office sales just increasing, as the two predictions almost cancel each other out.

This function will return one of the five values and nothing else. If a movie has a prediction that is lower than **–2**, return `-2`, and if a movie has a prediction higher than **2**, then return `2`.

Examples:

- `"Star Wars"` has more than one word with an even number of characters. It is predicted to increase, an event with a weight of **1**. However, each of its words is the same length, which means it is also predicted to drastically increase. This event has a weight of **2**. This, combined, results in a prediction of **3**, which is off the scale. As such, only `2` is returned.
- `"Incredibles"` has only one word, so its sales are predicted to drastically decrease (**–2**). However, one-word movies will always have the same number of characters in each word, so such movies are also predicted to drastically increase (**2**). After combining the two ratings, this means `"Incredibles"` will remain unchanged, and the function would return `0`.
- `"Gone with the Wind"` has more than two words (drastically decrease (**–2**)), and more than one word with an even number of letters (increase, (**1**)). Nothing else applies, so its final prediction will return `-1` to signify a decrease.

2.0 Extras – additional functionality

By completely the above task correctly, you have the potential to earn up to 90% of the grade for this assignment. By going beyond the material given and completely the extra function given below, you have the potential of earning another 10%.

```
int enumerateMovie(
    const char movie[MAX_STR], int charCounts[MAX_WORDS]
);
```

This function will enumerate a movie title's words. You will program functionality to count the number of words in each movie title as well as count the number of letters in each word. You can assume well-formatted movie titles consisting of only letters and spaces, where spaces are used to separate words—there will not be any movie titles with trailing space or any invalid input.

This function will return an `int` representing the number of words in the given movie title and will write to the `charCounts` array the character counts of each of the words.

For example, running `enumerateMovie()` on the movie `"Mean Girls"` would return `2` as well as write `{4, 5}` to the `charCounts` array; `"Incredibles"` would return `1` and write `{11}` to the `charCounts` array.

You must not modify the `movie` variable. There are some implementations of parsing a string that end up modifying it, but modifying input arguments is poor practice; modify a copy instead! In order to enforce this principle, `movie` will be made `const` (constant).

3.0 Compilation

Your program will be compiled using a makefile with the following flags:

```
-std=c99 -Wall
```

As stated above, the file containing your `main()` will not be submitted. In grading this assignment, we will use our own `main()` function to facilitate testing. Our main will be contained in a file called '*main.c*'. If you include a `main()` function in your files, it will not compile when we test, and you will likely get a 0.

Files required:

- `gryphflix.c` — your function implementations. You must submit this.
- `gryphflix.h` — your function definitions/prototypes, constants, library imports, student info, and declaration of academic integrity. You must submit this.
- `main.c` — your testing file/driver program used to verify outputs/test your functions. This file would contain a `main()`. You do not submit this.

Your makefile would include the following lines to be executed:

-----------start of makefile----------------------

```
gryphflix: gryphflix.o main.o

    gcc -Wall -std=c99 gryphflix.o main.o -o gryphflix

gryphflix.o: gryphflix.c gryphflix.h

    gcc -Wall -std=c99 -c gryphflix.c

main.o: main.c gryphflix.h
```

```
	gcc -Wall -std=c99 -c main.c
clean:
	rm *.o gryphflix
```

-------------end of makefile-------------

and running such a program would be done as follows:

```
    ./gryphflix
```

Your source code files can be anywhere and maintain any reasonable style of folder structure, but must include a makefile in the top-level directory that calls your sources files via relative path, as well as a main.c located in the same folder as the makefile.

4.0 Testing

Even though you won't submit a `main.c` file, you are still strongly encouraged to test throughout the entirety of the development lifecycle of this program.

One way to plan your test cases out on paper is to create tables representing movie ratings and then manually deduce ("trace") the function results in your head before turning them into test cases. This also helps you understand the program flow.

There are two important paradigms to note:

- Integration testing—where your program's functions are tested as a group.
- Unit testing—where the smallest possible parts of a program are tested in isolation to ensure they work on their own.

Integration testing the `getMostRecommendedMovies()` function may involve calling `openFileForReading()` on the movie and review files, calling `readReviews()` and `readMovies()` on them in order to derive the `movies` and `reviews` arrays, and then verifying the outcome from calling `getMostRecommendedMovies()` (which are the return value and `mostRecommendedMovies` array) with the two arrays as input. This ensures that these functions play well together.

In contrast, unit testing the `getMostRecommendedMovies()` function involves manually creating a `movies` and `reviews` array and then verifying the function's outputs (i.e. return value and `mostRecommendedMovies` array) using said arrays. This gets rid of dependency annoyances (if an earlier function like `readReviews()` or `readMovies()` fails, it would also likely cause `getMostRecommendedMovies()` to fail). Unit testing lets a developer know that a function works in isolation.

One reliable strategy for developing a program in the face of dependencies is to work on the modules (functions) that have no dependencies first. For example, although it is a good start to unit test `getMostCriticalReviewers()`, you will eventually need to integration test it with the output of `readReviews()`, which itself relies on `openFileForReading()`, which has no dependencies. `openFileForReading()` is a good place to start the project, as a result.

In a similar vein, `predictMovie()` relies on `enumerateMovie()`, but the function `enumerateMovie()` does not rely on anything. `enumerateMovie()`, therefore, is also a good function to work on and in the early stages of this assignment.

You know what they say: "*Test early and test often!*" (or was it '*vote*'?). Grading will employ a combination of unit and integration testing, so don't think that because you're only submitting functions for this assignment that they only need to work in isolation.

5.0 Submission

You will submit your assignment files (`gryphflix.c, makefile,` and `gryphflix.h`) to the submission box for A1 submission folder in GitLab.

************Instructions for Gitlab – Start (repeated from Lab1)**************

Step 1: Computer setup – this is an optional step. You can avoid this step by connecting to the school server via noMachine or (portkey.socs.uoguelph.ca + linux.socs.uoguelph.ca)

- make sure git is installed (https://git-scm.com/downloads)

- Mac users can use the terminal mode (I have tested it on my mac)

- Windows users can use powershell, WSL (windows subsystem for linux) or git bash.

- decide where cis2500 work will go and make a directory (e.g. mkdir CIS2500)

- cd to that directory from the terminal application

Step 2: From the chosen directory, now type:

```
git clone https://git.socs.uoguelph.ca/2500w21/<your username>/a1.git
```

At this point, you have a directory to work with on your local system.

Step 3: Do the work

- (remember to) cd to the new directory

- create the file that you are working on

- after first save, type: git add filename

ONLY ADD THE FILE ONCE!!! //do not do: git add .

Loop every 20-30 minutes:

```
   git commit -am "write something here about what you just did"
```

Once per day:

```
git push   // this is what stores local work back to the server.
```

To learn more about Gitlab, go to this link on moodle
https://moodle.socs.uoguelph.ca/course/view.php?id=169

************Instructions for Gitlab – End *******************

As explained earlier, do not submit a file containing a `main()` function.

If you do not plan on implementing every function, stub it out (create the definitions in your `gryphflix.c` and `gryphflix.h` files and return a meaningless value if it requires one). Otherwise, your submission will not compile with our grader and you will get a 0.

Your `gryphflix.h` header file will contain a program header comment signifying your academic integrity. It will be of the following form:

```
/*
  Student Name: Firstname Lastname
  Student ID: #######
  Due Date: Mon Day, Year
  Course: CIS*2500

  I have exclusive control over this submission via my password.
  By including this header comment, I certify that:
   1) I have read and understood the policy on academic integrity.
   2) I have completed Moodle's module on academic integrity.
   3) I have achieved at least 80% on the academic integrity quiz
  I assert that this work is my own. I have appropriate acknowledged
  any and all material that I have used, be it directly quoted or
  paraphrased. Furthermore, I certify that this assignment was written
  by me in its entirety.
*/
```

6.0 Marking

- Programs that don't compile receive an immediate zero (0).
- Programs that produce compilation warnings will receive a deduction of 1 mark per unique warning (i.e. 3 'unused variable' warnings will only deduct 1 mark).
- Loss of marks may also result from poor style (e.g. lack of comments, structure)
- Programs that use goto statements receive an immediate zero (0)
- Programs that use global variables statements receive an immediate zero (0)