College of Computer Studies
De La Salle University
CCPROG2 Machine Project | Term 2 AY '24-'25
(from January 31, 2025 of 12 AM to March 31, 2025 of 7:30 AM)

Foodie App

The definition of Foodie in the Cambridge dictionary is "*a person who loves food and is very interested in different types of food*". Whether this foodie happens to be fond of eating or cooking (or even both), this helps broaden their horizons and serves as a camaraderie with the people around them. Nice way to promote world peace. 😊

For your project, you are to create a **foodie app** that will be used as a module in an Android and iOS mobile application. Who said you won't be able to use your C code on a mobile platform? This is possible and this part can be done at another time (if you will be lucky). For now, we are to focus on the development of the foodie app module using C. This foodie app will be a companion in taking notes on one's food journey and can give specific insights such as one's favorite foods along with keeping track of must-keep recipes (from family or experience).

Your program should allow the user to manage the following entities with **all inputs to be validated on run-time and repeat capturing the input of a specific property or variable until the input becomes valid**:

□ ***User***

The sole user of the foodie app module. This user will have access to the food log and recipes with added security measures such as log-in. The following data will be stored for this user:

- Username – string between 8 and 50 alphanumeric characters (A-Z, a-z, 0-9)
- Password – string between 8 and 20 characters containing at least 1 uppercase character, 1 lowercase character, 1 number, and 1 special character (!, @. #,, $, %, &, *, and . only)
- Full Name – string between 5 and 80 characters containing alphabet characters (A-Z, a-z)
- Email Address – non-empty string with 30 characters at most and follows the email pattern
- Mobile Number – string that should contain 11 numbers only and must start with the 0 prefix

□ ***Food Log***

The food log will be 50 records maximum and will be stored using the following data structure:

- Food Name – string between 3 and 50 alphanumeric characters (A-Z, a-z, 0-9)
- Food Type – a character fixed to the following: 'a'-appetizer, 'm'-main course, 'd'-dessert
- Times Eaten – a positive number containing times the food was eaten
- Date First Tried – string of 8 number characters and 2 '/' with this date format: mm/dd/yyyy
- Location First Tried – non-empty string of size 30 on the place where the food was first tried
- Description – non-empty string of 300 characters on the overall impression of the food

□ ***Recipe***

The recipe will be 20 records maximum and will contain the following data structure:

- Recipe Name – string between 3 and 50 alphanumeric characters (A-Z, a-z, 0-9)
- Recipe Description – non-empty string of size 160 on the overall impression on the recipe
- Time to prepare – a positive number containing the time to prepare in minutes
- Time to cook – a positive number containing the time to cook in minutes
- Number of ingredients - a positive number of ingredients for the List of Ingredients part
- List of ingredients – non-empty string of size 80 for each ingredient (max 20) for this recipe
- Number of instructions - a positive number of instructions for the Instructions part
- Instructions – non-empty string of size 100 each instruction (max 20) on how to prepare and cook the food based from the recipe

Your program should also allow the user to **Manage Data**, by providing a user-friendly interface, as well as the functionality to do the following:

- ***Enter User Profile***
  The first run of the program will get user credentials from the username up to the mobile number. The password will be inputted twice on creating the user credentials, one for the intended password and another for confirmation. Succeeding runs of the program will ask for the provided username and password after successful user registration. If the provided user credentials are correct, show the program menu (referring to the set of bulleted features from Add Food Log to Exit). If not, make the user try again and display how many unsuccessful logins have been made so far. Upon reaching 3 unsuccessfully login attempts, show a message for the user to log in again after some time and terminate the program after letting the user press any key to confirm the message received.

- ***Add Food Log***
  The program will be able to create a food log record and store the information in the food logs file. The food name should be unique and no duplicate shall exist.

- ***Add Recipe***
  The program will be able to create a recipe record and store the information in the recipes file. The recipe name should be unique, and no duplicates should exist.

- ***Modify Food Log***
  The program will be able to edit a food log one at a time by searching for the food name. When the food name is in the records, display the food log information and proceed with editing the information. Before saving the modifications, the program must seek confirmation if the intended changes are to be applied. If the user confirms the modification, proceed with updating the food log. Otherwise, leave as is.

- ***Modify Recipe***
  The program will be able to edit a recipe one at a time by searching for the recipe name. When the recipe name is in the records, display the recipe information and proceed with editing the information. Before saving the modifications, the program must seek confirmation if the intended changes are to be applied. If the user confirms the modification, proceed with updating the recipe. Otherwise, leave as is.

- ***Delete Food Log***
  The program will be able to delete a food log one at a time by searching for the food name. When the food name is in the records, display the food log information to be deleted. Before deleting, the program must seek confirmation if the intended record is to be deleted. If the user confirms the deletion, proceed with removing the food log. Otherwise, leave as is.

- ***Delete Recipe***
  The program will be able to delete a recipe one at a time by searching for the recipe name. When the recipe name is in the records, display the recipe information to be deleted. Before deleting, the program must seek confirmation if the intended record is to be deleted. If the user confirms the deletion, proceed with removing the recipe. Otherwise, leave as is.

- ***Display User***
  The program will show the details of the registered user. Do not include the password.

- ***Display All Food Log and Recipe by Username***
  The program will search for a username which will show all the information by the Display User module along with all the food logs and recipes the specified user has created assuming the username searched happens to have some records within the program session or lifecycle. If the username is not the one logged in the program, display the full name and username only yet continue to display all the food logs and recipes made by that username.

- ***Display All Food Log***
  The program will show all the food logs in descending order (based from the date first tried).

- ***Display All Recipe***
  The program will show all the recipes in ascending order (by recipe name alphabetically).

- ***Search Food Log***
  The program will be able to search the food logs by food name. If the name of the food log happens to be available, print the food log data. Otherwise, print “No such food has been logged.”. Afterwards, let the user press any key to continue and go back to the main menu.

- ***Search Recipe***
  The program will be able to search the recipes by recipe name. If the name of the recipe happens to be available, print the recipe data. Otherwise, print “No such recipe has been entered.”. Afterwards, let the user press any key to continue and go back to the main menu.

- ***Export***
  Your program should allow all data to be saved into either a text file or a binary file. The data stored in the text file can be used later on. The system should allow the user to specify the filename. Filenames have at most 30 characters including the extension. If the file exists, the data will be overwritten (same applies when exiting the program).
  - User will be saved in the *user.dat* as a text file and the following file format can be used:
    ```
    <username><new line>
    <password><new line>
    <full name><new line>
    <email address><new line>
    <mobile number><new line>
    ```
  - Food Logs will be saved in the *foodlogs.txt* file as a text file with the following file format:
    ```
    <food name1><new line>
    <food author username1><space><food author fullname1><newline>
    <food type1><space><times eaten1><new line>
    <date first tried1><new line>
    <location1><new line>
    <description1><new line>
    <food name2><new line>
    <food author username2><space><food author fullname2><newline>
    <food type2><space><times eaten2><new line>
    <date first tried2><new line>
    <location2><new line>
    <description2><new line>
    …
    <food nameN><new line>
    <food author username2><space><food author fullname2><newline>
    <food typeN><space><times eatenN><new line>
    <date first triedN><new line>
    <locationN><new line>
    <descriptionN><new line>
    ```
  - Recipes will be saved in the *recipes.txt* file as a text file with the following file format:
    ```
    <recipe name1><new line>
    <recipe author username1><space><recipe author fullname1><new line>
    <recipe description1><new line>
    <time to prepare1><space><time to cook1><new line>
    <number of ingredients1><new line><list of ingredients1>=END=<new line>
    <number of instructions1><new line><instructions1>=END=<new line>
    <recipe name2><new line>
    <recipe author username2><space><recipe author fullname2><new line>
    <recipe description2><new line>
    <time to prepare2><space><time to cook2><new line>
    <number of ingredients2><new line><list of ingredients2>=END=<new line>
    <number of instructions2><new line><instructions2>=END=<new line>
    ```

```
…
<recipe nameN><new line>
<recipe author usernameN><space><recipe author fullnameN><new line>
<recipe descriptionN><new line>
<time to prepareN><space><time to cookN><new line>
<number of ingredientsN><new line><list of ingredientsN>=END=<new line>
<number of instructionsN><new line><instructionsN>=END=<new line>
```

- ***Import***
  Your program should allow the data stored in the text file to be added to the list of entries in the program (for both food logs and recipes). The user should be able to specify which file (input filename) to load. If entries have been added (or loaded previously) in the current run, the program shows one entry loaded from the text file and asks if this is to be added to the list of entries (in the array). If yes, it is added as another entry. If no, this entry is skipped. Whichever choice is made by the user, the program proceeds to retrieve the next entry in the file and asks the user again if this is to be included in the array or not, until all entries in the file are retrieved. The data in the text file will follow the format indicated in Export.

- ***Exit***
  The exit option will just allow the user to quit the program. The information in the lists should be cleared after this option, only through the Export module will specified data be saved in a file.

Bonus

A **maximum** of **10 points** may be given for features **over & above** the requirements, like (1) producing top 5 of the number of times eaten based from the food log (not necessarily with equal count); (2) producing top 3 recipes of the shortest time to prepare and cook; or other features not conflicting with the given requirements or changing the requirements) subject to **evaluation** of the teacher. **Required features** must be **completed first** before bonus features are credited. Note that use of conio.h, or other advanced C commands/statements may **not** necessarily merit bonuses.

Submission & Demo

- **Final MP Deadline: March 31, 2025 (M), 0730H via Canvas**. No project will be accepted anymore after the submission link is locked and the grade is automatically 0.

- **Requirements:** Complete Program

    - Make sure that your implementation has considerable and proper use of arrays, strings, structures, files, and user-defined functions, as appropriate, even if it is not strictly indicated.

    - It is expected that each feature is supported by at least one function that you implemented. Some of the functions may be reused (meaning you can just call functions you already implemented [in support] for other features. There can be more than one function to perform tasks in a required feature.

    - Debugging and testing was performed exhaustively. The program submitted has
        a. NO syntax errors
        b. NO warnings - make sure to activate -Wall (show all warnings compiler option) and that C99 standard is used in the codes
        c. NO logical errors -- based on the test cases that the program was subjected to

Important Notes:

1. Use **gcc -Wall** to compile your C program. Make sure you **test** your program completely (compiling & running).
2. Do not use brute force. Use **appropriate conditional** statements **properly**. Use, **wherever appropriate**, **appropriate loops** & **functions properly**.
3. You **may** use topics outside the scope of CCPROG2 but this will be **self-study**. Goto *label*, exit(), break (except in switch), continue, global variables, calling main() are **not allowed**.
4. Include **internal documentation** (comments) in your program.
5. The following is a checklist of the deliverables:

Checklist:

❐ Upload via AnimoSpace submission:
  - ❐ source code*
  - ❐ test script**
  - ❐ sample text file exported from your program

❐ email the softcopies of all requirements as attachments to **YOUR** own email address on or before the deadline

Legend:

***** Source code exhibit readability with supporting inline documentation (not just comments before the start of every function) and follows a coding style that is similar to those seen in the course notes and in the class discussions. The first page of the source code should have the following declaration (in comment) [replace the pronouns as necessary if you are working with a partner]:

```
/*********************************************************************************************************
This is to certify that this project is my own work, based on my personal efforts in studying and applying the concepts
learned. I have constructed the functions and their respective algorithms and corresponding code by myself. The
program was run, tested, and debugged by my own efforts. I further certify that I have not copied in part or whole or
otherwise plagiarized the work of other students and/or persons.
                                                                    <your full name>, DLSU ID# <number>
*********************************************************************************************************/
```

Example coding convention and comments before the function would look like this:

```
/* funcA returns the number of capital letters that are changed to small letters
   @param strWord - string containing only 1 word
   @param pCount - the address where the number of modifications from capital to small are
                   placed
   @return 1 if there is at least 1 modification and returns 0 if no modifications
   Pre-condition: strWord only contains letters in the alphabet
*/
int     //function return type is in a separate line from the
funcA(char strWord[20] ,    //preferred to have 1 param per line
      int * pCount)            //use of prefix for variable identifiers
{  //open brace is at the beginning of the new line, aligned with the matching close brace
   int    ctr;        /* declaration of all variables before the start of any statements –
                         not inserted in the middle or in loop- to promote readability */

   *pCount = 0;
   for (ctr = 0; ctr < strlen(strWord); ctr++)  /*use of post increment, instead of pre-
                                                 increment */
   {  //open brace is at the new line, not at the end
      if (strWord[ctr] >= 'A' && strWord[ctr] <= 'Z')
      {    strWord[ctr] = strWord[ctr] + 32;
           (*pCount)++;
      }
      printf("%c", strWord[ctr]);
   }

   if (*pCount > 0)
      return 1;
   return 0;
}
```

Test Script should be in a table format. There should be at least 3 categories (as indicated in the description) of test cases **per function. There is no need to test functions which are only for screen design (i.e., no computations/processing; just printf).

Sample is shown below.

Function	#	Description	Sample Input Data	Expected Output	Actual Output	P/F
sortIncreasing	1	Integers in array are in increasing order already	aData contains: 1 3 7 8 10 15 32 33 37 53	aData contains: 1 3 7 8 10 15 32 33 37 53	aData contains: 1 3 7 8 10 15 32 33 37 53	P
	2	Integers in array are in decreasing order	aData contains: 53 37 33 32 15 10 8 7 3 1	aData contains: 1 3 7 8 10 15 32 33 37 53	aData contains: 1 3 7 8 10 15 32 33 37 53	P
	3	Integers in array are combination of positive and negative numbers and in no particular sequence	aData contains: 57 30 -4 6 -5 -33 -96 0 82 -1	aData contains: -96 -33 -5 -4 -1 0 6 30 57 82	aData contains: -96 -33 -5 -4 -1 0 6 30 57 82	P

Test descriptions are supposed to be unique and should indicate classes/groups of test cases on what is being tested. Given the sample code in page 6, the following are four distinct classes of tests:

i.) testing with strWord containing all capital letters (or rephrased as "testing for at least 1 modification")
ii.) testing with strWord containing all small letters (or rephrased as "testing for no modification")
iii.) testing with strWord containing a mix of capital and small letters
iv.) testing when strWord is empty (contains empty string only)

The following test descriptions are **incorrectly formed**:

- Too specific: testing with strWord containing "HeLlo"
- Too general: testing if function can generate correct count OR testing if function correctly updates the strWord

- Not necessary -- since already defined in pre-condition: testing with strWord containing special symbols and numeric characters

6. Upload the softcopies via Submit Assignment in Canvas. You can submit multiple times prior to the deadline. However, only the last submission will be checked. Send also to your **mylasalle account** a **copy** of all deliverables.
7. Use **<surnameFirstInit>.c** & **<surnameFirstInit >.pdf** as your filenames for the source code and test script, respectively. You are allowed to create your own modules (.h) if you wish, in which case just make sure that filenames are descriptive.
8. During the MP **demo**, the student is expected to appear on time, to answer questions, in relation to the output and to the implementation (source code), and/or to revise the program based on a given demo problem. Failure to meet these requirements could result in a grade of 0 for the project.
9. It should be noted that during the MP demo, it is expected that the program can be compiled successfully and will run. If the program does not run, the grade for the project is automatically 0. However, a running program with complete features may not necessarily get full credit, as implementation (i.e., code) and other submissions (e.g., non-violation of restrictions evident in code, test script, and internal documentation) will still be checked.
10. The MP should be an HONEST intellectual product of the student/s. For this project, you are allowed to do this individually or to be in a group of 2 members only. Should you decide to work in a group, the following mechanics apply:

Individual Solution: Even if it is a group project, each student is still required to create his/her INITIAL solution to the MP individually without discussing it with any other students. This will help ensure that each student went through the process of reading, understanding, solving the problem and testing the solution.

Group Solution: Once both students are done with their solution: they discuss and compare their respective solutions (ONLY within the group) -- note that learning with a peer is the objective here -- to see a possibly different or better way of solving a problem. They then come up with their group's final solution -- which may be the solution of one of the students, or an improvement over both solutions. Only the group's final solution, with internal documentation (part of comment) indicating whose code was used or was it an improved version of both solutions) will be submitted as part of the final deliverables. It is the group solution that will be checked/assessed/graded. Thus, only 1 final set of deliverables should be uploaded by one of the members in the Canvas submission page. [Prior to submission, make sure to indicate the members in the group by JOINing the same group number.]

Individual Demo Problem: As each is expected to have solved the MP requirements individually prior to coming up with the final submission, both members should know all parts of the final code to allow each to INDIVIDUALLY complete the demo problem within a limited amount of time (to be announced nearer the demo schedule). This demo problem is given only on the day/time of the demo, and may be different per member of the group. Both students should be present/online during the demo, not just to present their individual demo problem solution, but also to answer questions pertaining to their group submission.

Grading: the MP grade will be the same for both students -- UNLESS there is a compelling and glaring reason as to why one student should get a different grade from the other -- for example, one student cannot answer questions properly OR do not know where or how to modify the code to solve the demo problem (in which case deductions may be applied or a 0 grade may be given -- to be determined on a case-to-case basis).

11. Any form of **cheating** (**asking other people not in the same group for help**, **submitting as your [own group's] work part of other's work**, **sharing your [individual or group's] algorithm and/or code to other students not in the same group, etc.**) can be punishable by a grade of **0.0** for the **course** & a **discipline case**.

Any requirement not fully implemented or instruction not followed will merit deductions.