Inventory Management System for a Retail Store

An inventory management system is a software application, or a set of tools designed to help businesses effectively manage their inventory. It provides a systematic approach to track, control, and organize inventory levels, stock movements, and related information.

The primary goal of an inventory management system is to ensure that businesses have the right amount of stock available at the right time to meet customer demand while minimizing costs and avoiding stockouts or overstock situations. It helps businesses streamline their operations, improve efficiency, and make informed decisions related to inventory management.

Requirements:

For your project, you are to create an inventory management program that should maintain (keep track of) at most 100 products. Below are the descriptions of functionalities required.

[Note that at the beginning of the program, the file inventory.txt, if existing, should be loaded to the main memory for processing]

1. Product Data Storage:
    - Each product in the store will have the following information associated with it:
        - Product ID: A unique identifier for each product (a string of at most 5 characters).
        - Product Name: The name or description of the product (a string of at most 30 characters).
        - Quantity in Stock: The number of units of the product available in the store (numeric).
        - Price per Unit: A numeric price of a single unit of the product.
    - The product data should be stored in a text file named "inventory.txt".
    - Each line in the file should represent a single product, with the attributes separated by a delimiter (such as a comma).
2. Authentication:
    - The user is assumed to be the administrator of the program. For verification, he/she is asked to input the password. For simplicity, let us assume that the correct password is “$up3rv1sor”.
    - If invalid log-in, a message “Unauthorized access not allowed” is shown, then program exits after 3 attempts.
3. Menu System:
    - The program should provide a user-friendly menu system with the following options:
        - View Inventory: Display the complete inventory list with product details.
        - Add Product: Add a new product to the inventory.
        - Update Product Quantity: Modify the quantity of a specific product.
        - Update Product Price: Change the price of a specific product.
        - Search Product: Find a product by its ID or name.
        - Exit: Terminate the program.
    - The menu should be displayed after each operation until the user selects the "Exit" option.
4. View Inventory:

- When the user selects the "View Inventory" option, the program should read the product data from the "inventory.txt" file and display it on the screen.
- The data should be presented in a clear and organized format, showing the product ID, name, quantity in stock, and price per unit for each product.

5. Add Product:
    - Choosing the "Add Product" option should prompt the user to enter the details of the new product, including the product ID, name, quantity, and price per unit.
    - The program should validate the input, ensuring that the product ID is unique and that all required fields are provided.
    - If the product ID is already present in the inventory, an appropriate error message should be displayed, and the user should be prompted to enter a unique ID.
6. Update Product Quantity:
    - Selecting the "Update Product Quantity" option should prompt the user to enter a product ID and the new quantity of the product.
    - The program should search for the product with the given ID in the "inventory.txt" file and update its quantity.
    - If the product is not found, an error message should be displayed to notify the user.
7. Update Product Price:
    - When the user selects the "Update Product Price" option, they should be asked to enter the product ID and the new price of the product.
    - The program should search for the product with the specified ID in the "inventory.txt" file and update its price.
    - If the product is not found, an appropriate error message should be displayed.
8. Search Product:
    - Selecting the "Search Product" option should prompt the user to enter either the product ID or name to search for.
    - The program should search for the product using the provided ID or name in the "inventory.txt" file and display its details if found.
    - If the product is not found, an error message should be displayed.
9. Exit:
    - The program saves into text file all current data on set of items. Then the program terminates properly.
10. Error Handling:
    - The program should handle potential errors, such as file reading/writing errors or invalid user input.
    - Appropriate error messages should be displayed to the user in case of any errors, guiding them on how to rectify the issue or proceed.
11. Code Organization and Modularity:
    - The program should be well-organized and modular, with separate functions responsible for different operations.
    - Code duplication should be avoided using functions and appropriate data structures.

Additional Considerations:

- Implement proper input validation to ensure that the user enters valid data and prevent unexpected program behavior.
- Use suitable data types and structures to store and manipulate the inventory data effectively.

- Comment your code thoroughly to explain the logic and make it easier to understand and maintain.
- Conduct comprehensive testing of your program to ensure it handles various scenarios correctly.
- Bonus
    - A maximum of 10 points may be given for features over & above the requirements (other features not conflicting with the given requirements or changing the requirements).
    - 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.

MP Checkpoint

June 24, 2023 (Saturday). This is a self-check and self-declaration on what has been accomplished so far. It is expected that by this date, at least 60 percent of all the features have been completed and tested already.

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

Submission & Demo

- Final MP Deadline: **July 29, 2023, 1200PM** via AnimoSpace. 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, 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 7, 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 AnimoSpace. 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. 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 with 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 to 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 AnimoSpace 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 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.

DEPARTMENT OF
SOFTWARE TECHNOLOGY