Project 8, Program Design

USF is creating an online store to sell t-shirts from its student organizations. They are hiring someone to code the backend of the system, which will be used by a staff to enter the information about the available t-shirt into the store database. You really want to get this position, so you decided to show off your skills by implementing a prototype in C using **dynamically allocated linked lists**. Each t-shirt model will have the following information stored in the database:

- student organization name: string with spaces and at most 50 characters
- size: strings of at most 3 characters (e.g., "XS", "S", "M", "L", "XL", "XXL")
- price: real number with at most two digits after the decimal point
- quantity in inventory: non-negative integer number smaller than or equal to 1000

You already have a sketch for the code in the file **`tshirt_store.c`**, now you must complete the following functions:

1. **`add_to_inventory`**:
    a. Ask the user to enter the student organization name and the size.
    b. Check if the inventory has a t-shirt for this organization with this specific size. If yes, your program should print a message stating that this t-shirt already exists and exit the function. Otherwise, ask the user to enter the price and the quantity, allocate memory for a new t-shirt and save the entered information, **add it to the end of the linked list**, and then exit the function.
    c. When exiting this function, return a pointer to **the first element of the linked list**.
2. **`search_by_organization`**: search by a student organization name. Ask the user to enter the name of the organization. Find all t-shirts on inventory for this organization. Display organization name, size, price and quantity. If no t-shirt is found, print a message.
3. **`search_by_size`**: search by t-shirt size. Ask the user to enter the desired size. Find all t-shirts on inventory with this size. Display organization name, size, price and quantity. If no t-shirt is found, print a message.
4. **`search_by_price`**: search by t-shirt price. Ask the user to enter the maximum price. Find all t-shirts on inventory with price smaller than or equal to the maximum price. Display organization name, size, price and quantity. If no t-shirt is found, print a message.
5. **`print_inventory`**: print organization name, size, price and quantity for all t-shirts on inventory.
6. **`clear_inventory`**: when the user exits the program, deallocate all the memory used for the linked list.

Note: use the **`read_line`** function to read the name of the student organization.

Testing guidelines:

1. Download the files *try_tshirt_store* and *tshirt_store.c* from Canvas and upload them to the **student cluster (sc.rc.usf.edu)**. Change the file permissions of the test script with the following command:

    chmod +x try _tshirt_store

2. Complete the missing functions of *tshirt_store.c*
3. Compile and test your solution for Project 8:

    gcc -Wall -std=c99 tshirt_store.c

    ./try_tshirt_store

Submission instructions:

1. Download the program *tshirt_store.c* from student cluster and submit it on Canvas>Assignments->Project 8.

Grading

Total points: 100

1. A program that does not compile will result in a zero.
2. Runtime error and compilation warning 5%
3. Commenting and style 15%
4. Functionality 80%
    a. Function implementation meets the requirements
    b. Function processes the linked list using **`malloc`** and **`free`** functions properly

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. In most cases, a function should have a brief comment above its definition describing what it does. Other than that, comments should be written only *needed* in order for a reader to understand what is happening.
3. 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.
4. Use consistent indentation to emphasize block structure.
5. Full line comments inside function bodies should conform to the indentation of the code where they appear.
6. Macro definitions (#define) should be used for defining symbolic names for numeric constants. For example: **#define PI 3.141592**
7. Use names of moderate length for variables. Most names should be between 2 and 12 letters long.
8. Use underscores to make compound names easier to read: **tot_vol** or **total_volumn** is clearer than totalvolumn.