Project 2, Program Design

Problem 1 – Stopping Panic Buying (50 points)

Panic buying is the act of buying large amounts of a product in anticipation of an emergency. The most recent example is the shortage of items such as face masks, food, bottled water, milk, and toilet paper in the beginning of the COVID-19 pandemic.

To avoid this problem, a local grocery store has adopted the following policy for certain items. For an item with cost **C**, the client will pay **C** dollars for the first unit purchased, **2×C** dollars for the second unit purchased, **3×C** dollars for the third unit purchased, and so on. Thus, a client must pay **i×C** dollars for the **i**-th unit purchased.

If a client must buy **N** units of an item with cost **C** and has **D** dollars in his wallet, will the client have enough money? If not, how many extra dollars will the client need to complete this purchase?

1. The program takes the price per unit (C) of a certain item, the desired number of units (N) and an amount of dollars (D) as input. It computes the total cost of N units and checks if D dollars are enough to complete the purchase. If yes, the program prints a message. Otherwise, it prints how many extra dollars are necessary to complete the purchase.
2. Input validation: the program validates the input price (greater than 0), number of units (greater than 0), and amount of dollars (greater than or equal to 0). If any of the three inputs is invalid, the program prints a message and exits.

Hint: use the counter of the loop to compute the price of the i-th unit.

Example #1:

```
Enter item price: 3
Enter number of units: 4
Enter money amount: 17
Needs 13 dollar(s) to complete the purchase!
```

Example #2:

```
Enter item price: 1
Enter number of units: 1
Enter money amount: 1
Has enough money!
```

Problem 2 – Characters (50 points)

In this program, we define an input to be in order if the characters of the input

1. are alphabetic letters, lower case or upper case.
2. any two neighboring letters (regardless of case) are in order, for example, ‘c’ and ‘k’ are in order but ‘s’ and ‘b’ is not in order because ‘c’ is less than ‘k’ and ‘s’ is greater than ‘b’, considering their ASCII values.
3. if two neighboring letters are same, they are considered in order.

Write a program that determines if the input is in order.

1) Assume the input contains two or more characters.
2) Convert input characters to lower case before comparison.
3) The user input ends with the user pressing the enter key (a new line character).
4) Use **getchar()** to read in the input. Character handling functions are allowed.

Hint: use two variables to keep track of two neighboring characters.

Example #1:

```
Input: “all”
Output: In order
```

Example #2:

```
Input: “littlepigs”
Output: Not in order
```

Example #3:

```
Input: "cS"
Output: In order
```

Example #4:

```
Input: "F28"
Output: Not in order
```

Other requirements and submission:

1. Program names:

   project2_panic.c
   project2_inOrder.c

2. Compile on ***student cluster (sc.rc.usf.edu)***:

   gcc –Wall project2_panic.c

   gcc –Wall project2_inOrder.c

3. Change Unix file permission on Unix:

   chmod 600 project2_panic.c

   chmod 600 project2_inOrder.c

4. Test your program with the shell script on the student cluster:

   chmod +x try_project2_panic

   ./try_project2_panic

   chmod +x try_project2_inOrder

./try_project2_inOrder

5. Download the program *project2_panic.c* and *project2_inOrder.c* from student cluster and submit it on Canvas>Assignments.

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 requirement 80%

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 either underscores or capitalization for compound names for variable: `tot_vol`, `total_volumn`, or `totalVolumn`.