Project 3, Program Design

Write a C program that asks the user to two fractions separated by an operator and calculate the result. The program should allow add, subtract, multiply, or divide two fractions by entering either +, -, *, or / between the fractions:

```
Enter two fractions separated by the operator: 4/7 – 1/3
The difference is: 5/21
```

1) Use a switch statement to process the operator and the math.
2) There might be space(s) between the fractions and the operator in the input.
3) The program does NOT need to reduce the resulting fraction to lowest terms.
4) If the operator entered is not one of the acceptable operators, display an error message as the result.

Before you submit

1. Compile both programs with –Wall. –Wall shows the warnings by the compiler. Be sure it compiles on ***circe*** with no errors and no warnings.

   gcc –Wall fraction.c

2. Be sure your Unix source file is read & write protected. Change Unix file permission on Unix:

   chmod 600 fraction.c

3. Test your fraction program with the shell script try_fraction on Unix:

   chmod +x try_fraction

   ./try_fraction

4. Submit fraction.c on Canvas.

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%

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.