Project 4, Program Design

Important: the programs in this project will be graded based on whether the required functionality were implemented correctly instead of whether it produces the correct output, for the functionality part (80% of the total grade)

1. Modify Project 1 (dollar.c) (Write a C program that asks the user to enter a U.S. dollar amount and then shows how to pay that amount using the smallest number of $20, $10, $5, and $1 bills.)

   So it includes the following function:

   ```
   void pay_amount(int dollars, int *twenties, int *tens, int
   *fives, int *ones);
   ```

   The function determines the smallest number of $20, $10, $5, and $1 bills necessary to pay the amount represented by the `dollars` parameter. The `twenties` parameter points to a variable in which the function will store the number of $20 bills required. The `tens`, `fives`, and `ones` parameters are similar. Modify the `main` function so it calls `pay_amount` to compute the smallest number of $20, $10, $5, and $1 bills. The `main` function should display the result.

2. Modify Project 2 (fraction.c) so it reduces the fraction to lowest terms:

   ```
   Enter two fractions separated by the operator: 4/9 - 1/9
   The result is: 1/3
   ```

   The program should include the following function:

   ```
   void reduce(int numerator, int denominator, int
   *reduced_numerator, int *reduced_denominator);
   ```

   `numerator` and `denominator` are the numerator and denominator of a fraction. `Reduced_numerator` and `reduced_denominator` are pointers to variable in which the function will store the numerator and denominator of the fraction once it has been reduced to lowest terms. Modify the `main` function so it calls `reduce` before displaying the result.

   To reduce a fraction to lowest terms, first compute the GCD (greatest common divisor) of the numerator and denominator. Then divide the numerator and denominator by the GCD. Add the following function to your program that finds the GCD of two numbers m and n:

   ```
   int find_gcd(int m, int n)
   {

     if(n == 0) return m;
     return find_gcd(n, m%n);
   }
   ```

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 dollar2.c

   gcc –Wall fraction2.c

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

   chmod 600 dollar2.c

   chmod 600 fraction2.c

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

   chmod +x try_dollar

   ./try_dollar

   chmod +x try_fraction2

   ./try_fraction2

4. Submit *dollar2.c* and *fraction2.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. Information to include in the comment for a function: name of the function, purpose of the function, meaning of each parameter, description of return value (if any), description of side effects (if any, such as modifying external variables)
4. 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.
5. Use consistent indentation to emphasize block structure.
6. Full line comments inside function bodies should conform to the indentation of the code where they appear.
7. Macro definitions (#define) should be used for defining symbolic names for numeric constants. For example: `#define PI 3.141592`
8. Use names of moderate length for variables. Most names should be between 2 and 12 letters long.
9. Use underscores to make compound names easier to read: `tot_vol` or `total_volumn` is clearer than totalvolumn.