CSCI 1730 – Programming Assignment 1 – 50 pts.
Due Date: Thursday, January 29, 2015

What you need to turn in:

- **Code listing:** A printed copy of your C++ code solution for each problem. Remember to include your name on every page you turn in. Be sure to follow the "Code Style Guidelines" specified in the "Assignment Information and Guidelines" handout that was distributed on the first day of class (these guidelines are also available at the class web page).
- **Code files:** E-mail me a copy of your C++ code. Copy/paste all code into one e-mail message – please, do not send attachments. **Enter your name in the subject line of your email.**
- **Working in Pairs:** If you want to work with one other student in our class on this assignment, this is acceptable provided that both members of the pair make a contribution to the solution. If you decide to work in a pair, turn in only one copy of the solution – clearly identify the name of each pair member on everything that you turn in.
- **Late Assignments:** Assignments are due **by the end of class** on the specified due date (both the paper copies and the e-mail copies). Assignments turned in late will be assessed a 20% penalty per day late.

1. (12 pts.) Before the 1950's, Great Britain used a monetary system based on pounds, shillings, and pence. There were 20 shillings to a pound, and 12 pence to a shilling. The notation for this old system used the pound sign, £, and two decimal points, so that, for example, £5.2.8 meant 5 pounds, 2 shillings, and 8 pence. We'll call this system old-pounds. The new monetary system, introduced in the 1950's, consists of only pounds and pence, with 100 pence to a pound (like U.S. dollars and cents). We'll call this system decimal-pounds. So, £5.2.8 in the old-pounds is £5.13 in decimal-pounds.

Write a program that asks the user to enter two money amounts expressed in old-pounds and will then add the two amounts and display the answer both in old-pounds and in decimal-pounds.

Program requirements:

a) Read the numbers into integer variables and then make use of the integer division and remainder (%) operators to do the needed arithmetic for both calculation of the old-pounds sum as well as the decimal-pounds equivalent.

b) To read in the old-pound amounts, make use of the fact that the extraction operator (>>) can be chained to read in more than one quantity at once. For example:

```
cin >> pound >> dot1 >> shill >> dot2 >> pence;
```

will read an old-pound amount into integer variables pound, shill, and pence, and read the "dots" into the character variables `dot1` and `dot2`.

Here is an example of a user's interaction with the program (user input in **bold**):

```
Enter first old-pound amount: 5.10.11
Enter second old-pound amount: 3.19.5
Old-pound total = 9.10.4
Decimal-pound total = 9.51
```

2. (12 pts.) The Fibonacci sequence is 0, 1, 1, 2, 3, 5, 8, 13, 21, … where the first two terms are 0 and 1, and each term thereafter is the sum of the two preceding terms.

Write a program that repeatedly prompts for and reads a positive integer value *n* and then calculates and displays the *n*th number in the Fibonacci sequence (the program should force reentry if the user enters a non-positive *n*). For example, if *n* = 8, then the program would display 13.

Program requirements:

a) Use a `do` loop to ensure that the user enters a positive value for *n*.
b) Use a `for` loop to perform the calculations needed to determine the *n*th Fibonacci number.
c) Use a `do` loop to control the program repetition for finding more Fibonacci numbers.

Here is an example of a user's interaction with the program (user input in **bold**):

```
Enter n>0: 8
Fibonacci number 8 = 13
Continue (y or n)? y
Enter n>0: 20
Fibonacci number 20 = 4181
Continue (y or n)? n
```

3. (14 pts.) Write a program that implements a four-function (+, –, *, /) calculator for fractions. The program should prompt for and read the first fraction, an arithmetic operator, and a second fraction, perform the required calculation, display the result, and then ask if the user wants to continue. The result should be displayed as a fraction - it does not need to be in reduced form.

Program requirements:

a) Write a program that does not make use of any floating point data nor variables. Read the numerator and denominator of each fraction into separate integer variables (using the same technique given in the first problem). Then produce the numerator and denominator of the resulting fraction using basic fraction arithmetic. For example, a/b + c/d = (ad+bc)/bd.
b) Make use of a `switch` statement to organize the calculations associated with the different arithmetic operations.
c) Use a `do` loop to control the calculator repetition.

Here is an example of a user's interaction with the program (user input in **bold**):

```
Enter first fraction: 1/2
Enter operation: +
Enter second fraction: 1/3
Sum = 5/6
Continue (y or n)? y
Enter first fraction: 3/4
Enter operation: -
Enter second fraction: 1/6
Difference = 14/24
Continue (y or n)? y
Enter first fraction: 2/3
Enter operation: *
Enter second fraction: 5/6
Product = 10/18
```

```
Continue (y or n)? y
Enter first fraction: 6/7
Enter operation: /
Enter second fraction: 8/9
Quotient = 54/56
Continue (y or n)? n
```

4. (12 pts.) Given a positive real number *n*, and an approximation for its square root, *approx*, a closer approximation to the actual square root, *new-approx*, can be obtained using the formula:

$$new\text{-}approx = \frac{\frac{n}{approx} + approx}{2}$$

Using this information, write a program that prompts the user for a positive number and an initial approximation to the square root of the number, and then approximates the actual square root of the number accurate to 0.00001.

Hint: Using the initial approximation as the first value of *approx*, repeatedly calculate a *new-approx*, using the above formula, until the difference between the *new-approx* and *approx* is between –0.00001 and 0.00001 (i.e., –0.00001 < *new-approx* – *approx* < 0.00001). If this condition is not met, then calculate another *new-approx*, using its previous value as the value of *approx*. Continue this process until the condition is met.

Here is an example of a user's interaction with the program (user input in **bold**):

```
Enter n: 31
Enter first approximation: 5
The square root of 31 = 5.56776
```