**CSCI 1730 – Programming Assignment 1 – 100 pts.**

**Due Date: Feb 4, 2019**

1. 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:

1. Read the numbers into **integer variables** and then make use of the **integer division** and remainder (%) operators to do the needed arithmetic for both calculations of the old-pounds sum as well as the decimal-pounds equivalent.
2. 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 >> dot >> shill >> dot >> pence;

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

Here is an example of a user’s interaction with the program (user input is 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. 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:

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

Following is a sample output from the program (user input is bold):

Enter a positive integer n: **-3**

Enter a positive integer n: **0**

Enter a positive integer n: **8**

8th Finbonacci number 13

Continue (y or n)? **y**

Enter a positive integer n: **5**

5th Finbonacci number 3

Continue (y or n)? **n**

--------------------------------

Process exited after 37.34 seconds with return value 0

Press any key to continue . . .

3. 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:

1. 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.
2. Make use of a **switch statement** to organize the calculations associated with the different arithmetic operations.
3. Use a **do loop** to control the calculator repetition.

Here is an example of a user’s interaction with the program:

Enter first fraction: **1/2**

Enter operation: **+**

Enter second fraction: **1/3**

Sum = 5/6

Continue (y or n)? **y**

Enter first fraction: **1/2**

Enter operation: **-**

Enter second fraction: **1/3**

Difference = 1/6

Continue (y or n)? **y**

Enter first fraction: **1/2**

Enter operation: **/**

Enter second fraction: **1/3**

Quotient = 3/2

Continue (y or n)? **n**

--------------------------------

Process exited after 56.37 seconds with return value 0

Press any key to continue . . .

4. Recall that a quadratic equation is one of the form , where the coefficients *a*, *b*, and *c* are real numbers and . The solutions to a quadratic equation can found using the quadratic formula:



Write a program that will prompt for and read the values of the coefficients *a*, *b*, and *c*, and then calculate and display the solutions of the quadratic equation having those coefficients.

Program requirements:

1. Use floating point variables to hold the values of *a*, *b*, and *c*.
2. Your program should generate an error message and force reentry if the user enters a value of zero for *a*.
3. Your program should work whether the quadratic equation has two real number roots, one real number root, or two complex number roots.
4. The function **sqrt** should be used to calculate the needed square root. To find the square root of a variable x, include this code: **sqrt(x)** Note: To use this function, you need to add **#include <cmath>** after the other include statement at the beginning of your code. To find the square root of a variable x, include this code: sqrt(x)

Here are examples of a user’s interaction with the program:

Run 1:

Quadratic equation solver:

Enter a: **0**

a cannot be zero - reenter

Enter a: **1**

Enter b: **3**

Enter c: **2**

There are two real solutions: -1 and -2

Run 2:

Quadratic equation solver:

Enter a: **1**

Enter b: **-6**

Enter c: **9**

There is one real solution: 3

Run 3:

Quadratic equation solver:

Enter a: **1**

Enter b: **2**

Enter c: **10**

There are two complex solutions: -1+3i and –1-3i

5. 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:



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 absolute value of the difference between the *new*-*approx* and *approx* is less than or equal to 0.00001. If the difference is greater than 0.00001, then calculate another *new*-*approx*, using its previous value as the value of *approx*.

Note: The function **fabs** should be used to calculate the needed absolute value. To find the absolute value of a variable x, include this code: **fabs(x)** Note: To use this function, you need to add **#include <cmath>** after the other include statement at the beginning of your code.

A sample run:

Enter n: **5**

Enter first approximation: **2**

The square root of 5 = 2.23607

6. Euclid’s method for finding the greatest common divisor (GCD) of two positive integers is given by the following algorithm:

1. Divide the larger number by the smaller and retain the remainder.
2. Divide the smaller number by the remainder, again retaining the remainder.
3. Continue dividing the prior remainder by the current remainder until the remainder is zero, at which point the last nonzero remainder is the greatest common divisor.

For example, find the GCD of 72 and 114:

114/72 = 1 with remainder 42

72/42 = 1 with remainder 30

42/30 = 1 with remainder 12

30/12 = 2 with remainder 6

12/6 = 2 with remainder 0

So, the GCD of 72 and 114 is 6, the last nonzero remainder.

Using Euclid’s method, write a C++ function, gcd, that will take in two positive integers, determine the GCD of the integers, and return the result.

Then, modify the fraction calculator problem from problem 3 to include your gcd function and use it to produce reduced fraction results. That is, after you calculate the numerator and denominator of a fraction arithmetic operation, find their GCD and then divide each by the GCD to obtain the reduced fraction numerator and denominator.

A sample run:

Enter first fraction: **2/4**

Enter operation: **-**

Enter second fraction: **5/6**

Difference = 1/-3

Continue (y or n)? **y**

Enter first fraction: **2/4**

Enter operation: **+**

Enter second fraction: **5/6**

Sum = 4/3

Continue (y or n)? **n**

--------------------------------

Process exited after 105.3 seconds with return value 0

Press any key to continue . . .

**What you need to turn in:**

1. **Source code listing:** A printed copy of the source code for each problem. Remember to include the name of each group member in a comment at the top of your source code. Be sure to follow the “Code Style Guidelines” specified in class.
2. **Source code files:** E-mail me your source code as attachments.
3. **Working in Groups:** Every student will work with two other students in our class on this assignment; all members of the group must contribute to the solution. **Turn in only one copy of the solution** – **clearly identify the name of each member on everything that you turn in.**
4. **Late Assignments:** Assignments are due before class on the specified due date (both the paper copies and the e-mail copies). If you wish to turn in the paper copy of an assignment after class, place them under my office door. Assignments turned in late will be assessed a 20% penalty per class day late.