CSci 1113: Introduction to C/C++ Programming for Scientists and Engineers
HW 4

Due Date: Monday, Mar. 11 before 5:30pm. (Note this is a two-week assignment.)

Purpose: In this homework you will write C++ programs that use functions and/or file I/O.

Instructions: This is an *individual* homework assignment. There are *three* problems worth 20 points each. The rules and reminders are the same as for past homework, so see the previous homework write-ups if you need a reminder.

Problem A: π Day (March 14) is Coming (20 points)

Constants such as π and e play an important role in science and engineering. Nowadays, we can find (approximate) values of important constants such as π by using a calculator or a math library. But how did people calculate these constants prior to electronic calculators and computers? This problem asks you to implement one formula for calculating an approximation to π.

Specifically, you will need to use the formula

$$\pi = 16 \arctan(1/5) - 4 \arctan(1/239)$$

along with the arctangent series

$$\arctan x = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \dots.$$

This problem asks you to approximate π using these formulas. To begin, first write a function `double arctanApprox(double x, double tol)` that uses the arctangent series above to calculate an estimate for `arctan(x)`. This function should first estimate arctan(x) using just one term from the series, then using two terms. If these two estimates are strictly within the tolerance, then return the two-term estimate. If not, find an estimate using three terms, and if that is strictly within the tolerance of the two-term estimate, return the three-term estimate. Otherwise continue in this fashion, generating a new estimate containing one more term in the series than the previous estimate, stopping when the new and previous estimates are strictly within the tolerance, and then returning the new estimate.

Once you have your arctangent function working, notice that you can get an approximation for π by using the arctangent series to give approximations for both 1.0/5.0 and 1.0/239.0, and then combining them as in the π formula above. So write a main program that does the following:

1. Your program should first ask the user to input a tolerance. Validate that the input value is positive, if not, ask the user to input another value. Do this repeatedly until the user inputs a valid value.

2. Your program should then approximate π using the two equations above and your arctangent function. Specifically, it should call the arctangent function once for each arctangent term in the π equation, *and with tolerance each time being one-twentieth of the user-input tolerance.* So notice that the user-input tolerance and the tolerance value sent to the arctangent function are slightly different values.

3. Your program should then output the π approximation with 13 digits to the right of the decimal point.

4. The program should then ask if the user wishes to obtain another estimate. If the user answers 'y' or 'Y' then then the program should repeat the steps here. Otherwise it should terminate.

Note that in doing this problem you should not use the arctangent functions from the `cmath` library; rather you must use the series above to approximate the arctangent.

Here are a couple examples:
(Formatting note, only one space is used or expected in all places in this example)

Example 1:

```
Input tolerance:  .1
Pi approximation:  3.1405970293261
Do you wish to approximate pi again?  y
Input tolerance:  .001
Pi approximation:  3.1415917721832
Do you wish to approximate pi again?  n
```

Example 2:

```
Input tolerance:  -42
Tolerance must be positive.  Input tolerance:  -4.2
Tolerance must be positive.  Input tolerance:  .00042
Pi approximation:  3.1415917721832
Do you wish to approximate pi again?  Y
Input tolerance:  1E-8
Pi approximation:  3.1415926536236
Do you wish to approximate pi again?  n
```

Depending on how you order your calculations, you might get slightly different values for the output. However, if your values are different they should not be much different.

When you are done, name the source code file <username>_4A.cpp (where you replace <username> with your U of M email address) and submit it using the HW 4 Problem A submission link on the class website. Remember to follow the naming convention diligently.

Problem B: Vectors, Vectors, Vectors (20 points)

This problem will give you practice in writing a problem involving multiple functions, void functions, and pass-by-reference. The vector operations in this problem are often parts of the calculation often done to determine the color of an object for a particular pixel in a graphics scene.

Write a program consisting of four functions:

- `double vectorLength(double u, double v, double w)`. This function computes and returns the length $\sqrt{u^2 + v^2 + w^2}$ of the vector (u, v, w).
- `void vectorNormalize(double& u, double& v, double& w)`. This function takes a nonzero vector (u, v, w) and normalizes it — i.e., it replaces it by a vector in the same direction but with unit length. It should do this by dividing each vector component by the vector length obtained by calling the `vectorLength` function.
- `void interpolateVectors(double a, double u1, double v1, double w1, double u2, double v2, double w2, double& u, double& v, double& w)`. This function first computes a vector that is a weighted average of (u_1, v_1, w_1) and (u_2, v_2, w_2). Specifically, it should compute the vector

  $$((1-a)u_1 + au_2,\ (1-a)v_1 + av_2,\ (1-a)w_1 + aw_2).$$

  It should then normalize it using the `vectorNormalize` function, and return this normalized vector through the pass-by-reference parameters `u`, `v`, and `w`.
- `int main()`. Your `main()` program will not be extremely long, but will need to do a number of tasks:
  - Have a user input two vectors.
  - If all components of either vector are zero, print "Vectors must be non-zero" and terminate with `exit(1)`. If the vectors are both non-zero, then your program should do the remaining steps below.
  - Normalize each of the two vectors using the normalize function.
  - Ask the user to input a number n. This will be the number of "interpolation vectors" to compute, as explained below. The value n should be a positive integer that is at least 2, but you do not need to validate this input.
  - Then call `interpolateVectors` n times with (u_1, v_1, w_1) and (u_2, v_2, w_2) always being the normalized user-input vectors, and a taking on successive, equally spaced values between 0.0 and 1.0, inclusive. For example, if n is 5, then `interpolateVectors` should be called 5 times with a being 0.0, .25, .5, .75, and 1.0. Each call to `interpolateVectors` will pass back a vector through the pass-by-reference variables. Your program should print these vectors as shown in the example below: with each value having four digits to the right of the decimal point (use `cout.setf(ios::fixed);` etc.), and in a field of width 8 (use `setw(8)`).

Here is an example showing the input and output text and formatting. As usual, follow it exactly, and all spaces are single-spaces.

```
Input the first vector u1 v1 w1:  1.0 1.0 0.0
Input the second vector u2 v2 w2:  2.0 -1.0 2.0
Enter the number of interpolation vectors to compute:  5
Interpolation vectors:
```

```
Input the first vector u1 v1 w1: 1.0 1.0 0.0
Input the second vector u2 v2 w2: 2.0 -1.0 2.0
Enter the number of interpolation vectors to compute: 5
Interpolation vectors:
  0.7071  0.7071  0.0000
  0.8252  0.5292  0.1973
  0.8739  0.2378  0.4241
  0.8013 -0.0867  0.5920
  0.6667 -0.3333  0.6667
```

Here's an example of the output when the input validation fails:

```
Input the first vector u1 v1 w1:  1.0 1.0 1.0
Input the second vector u2 v2 w2:  0.0 0.0 0.0
Vectors must be non-zero.
```

Comments

- None of the functions should be especially long. If you find yourself writing large amounts of code, think about whether you are writing the functions efficiently.
- Implement your solution incrementally, for example, make sure you get the length function working before beginning to write the normalize function. Then get the normalize function working before writing the interpolate function. (This program has a large number of parts, but should not be extremely difficult *if you break the problem up into these subparts, and implement the subparts incrementally.*)
- Some functions (in particular `vectorLength`) are very simple. However, make sure you follow the problem instructions here; for example, do *not* skip writing the length function by doing the length calculation directly in the normalize function.

When done, name the source code file <username>_4B.cpp (where you replace <username> with your U of M email address) and submit it using the HW 4 Problem B submission link on the class website. As usual, remember to follow the naming convention diligently.

Problem C: Tournament of Pets (20 points)

One of the most popular use for computer programming is analyzing large data files. A simple program can often read and summarize large amounts of information much faster than any human. Once there's enough data to work with it can sometimes be impossible to understand except through computer summarization. One popular example of this is using programs to better understand the large volume of data generated by various sporting leagues.

For this problem you will need the `sports.dat` file from Canvas. This file contains information from the BPL (Beloved Pet League). In the beloved pet league seven teams (Cats, Dogs, Bunnies, Rats, Birds, Fish, and Horses) compete to be the most beloved pets. `sports.dat` contains data about 330 matches from 10 seasons of the BPL. Each line of `sports.dat` is formatted as follows:

```
<season number> <team1 name> <team 2 name> <team 1 score> <team 2 score>
```

An excerpt of the file is below:

```
0 Horses Fish 1 4
0 Birds Cats 0 1
0 Horses Rats 9 6
0 Fish Horses 2 0
0 Rats Bunnies 1 5
```

This example shows a few properties of the file:

- Teams can play each-other more than once per season (Fish and Horses)
- A team may be either team 1 or team 2 (Note how Fish and Horses appear in a different order between their two matches)
- The BPL does not allow ties, so matches continue until one team or the other has a higher score.

Your task in this problem is to write a program to help us summarize the BPL data and settle questions such as "which pet is better, Cats or Dogs?" To do this your program will first need to ask the user which two teams they wish to compare. Then it will need to read the BPL file match-by-match. For each match you will need to determine whether the match was between the target teams, and determine who won. At the end it will need to print how many matches occurred between the two teams, how many times each team won, and which team won the most.

Examples are below.
(Formatting note, only one space is used or expected in all places in this example)

Example 1:

```
Please input the two teams you are interested in:  Cats Dogs
The Cats and the Dogs played 15 times.
The Cats won 4 times.
The Dogs won 11 times.
The Dogs are the better team!
```

Example 2:

```
Please input the two teams you are interested in:  Cats Rats
The Cats and the Rats played 16 times.
The Cats won 11 times.
The Rats won 5 times.
The Cats are the better team!
```

Example 3:

```
Please input the two teams you are interested in:  Dogs Rats
The Dogs and the Rats played 12 times.
The Dogs won 6 times.
The Rats won 6 times.
The teams are equally matched!
```

Example 4:

```
Please input the two teams you are interested in:  Rocks Horses
The Rocks and the Horses played 0 times.
The Rocks won 0 times.
The Horses won 0 times.
The teams are equally matched!
```

Notes:

- The examples above are based one the same data file you were given.
- You can assume the data file will always be named `sports.dat`.
- You cannot assume that your code will be graded on the same `sports.dat` file as the one provided on Canvas.
- You should not assume that `sports.dat` has exactly 330 matches in it.
- You do not have to validate team names, if the user enters a team that is not in the BPL, it should show no matches. (see example 4).
- Team names are case sensitive. While the "Cats" are a team in the BPL, the "cats" are not.
- You may find it useful to write functions to simplify parts of this program. For example a `printResults` function will certainly make the main method easier to read and more elegant.

When you are done, name the source code file <username>_4C.cpp (where you replace <username> with your U of M email address) and submit it using the HW 4 Problem C submission link on the class website. Remember to follow the naming convention diligently.