CSCI 1300 Introduction to Computer Programming
Instructors: Knox
Assignment 3
Due Friday, Feb 10, by 12:30 pm

For this assignment, the solution to **each problem should be in a separate .cpp file**. Create a new program for each problem within CodeBlocks. Name the file *problem1.cpp* for the first problem and *problem2.cpp* for the second one. Once you have your code running on your virtual machine (VM), you must *submit it to the autograder by zipping all the files* into a single file to be submitted. You **must also submit your code (.zip) to Moodle** to get full credit for the assignment.

Important Note Regarding Zipping Your File: Ensure the .zip is a "flat zip." That is, multi-select the files you wish to zip together and compress them; do not zip the folder they are contained in - this is called a hierarchical zip and COG will not be capable of locating your files.

Submitting Your Code to the Autograder:
Before you submit your code to COG, make sure it runs on your computer. If it doesn't run on the VM, it won't run on COG. The computer science autograder, known as COG, can be found here: https://web-cog-csci1300.cs.colorado.edu

- Login to COG using your identikey and password.
- Select **Assignment 3** from the dropdown.
- Upload your .zip file and click Submit.

Your files within the .zip **must** be named *problem1.cpp and problem2.cpp* for the grading script to run. COG will run its tests and display the results in the window below the *Submit* button. If your code doesn't run correctly on COG, read the error messages carefully, correct the mistakes in your code, and upload a new file. You can modify your code and resubmit as many times as you need to, up until the assignment due date.

You must submit your code on COG and Moodle. **For this assignment you must schedule an interview grading appointment with your TA after you submit your code.** We'll talk more about scheduling the interview in lecture and recitation.

The grade distribution for this assignment is 60% grade from cog and 40% from interview grading.

Submitting Your Code to Moodle:
You must submit your code to Moodle to get full credit for the assignment, even if the computer science autograder gives you a perfect score.

Please also include comments in your code submission to describe what your code is doing. Comments at the header of your files should include your name, recitation TA, and the assignment and problem number. TAs will be checking that you code has comments.

Problem Set

For each of the following problems, create a program to solve the problem. Divide the program into separate, but cooperating, functions. Do not write the complete program as one big ``chunk'' of statements in main().

Problem 1: The Story Generator

In the game Mad-libs, players are asked for parts of speech, such a noun, adjective, or adverb, and those words are plugged into a template sentence to generate a sometimes- funny story.

For this problem, write a function named "**madLibs**" that plays a simple game of Mad-libs. Your program needs to store the story template in a string variable and ask the user for parts of speech to fill in the template. Use the following template:

"In the book War of the <PLURAL NOUN>, the main character is an anonymous <OCCUPATION> who records the arrival of the <ANIMAL>s in <PLACE> -- Needless to say, havoc reigns as the <ANIMAL>s continue to <VERB> everything in sight, until they are killed by the common <SINGULAR NOUN>."

When the programmer starts, you should first ask the user if they want to play a game. Your question should look like:

"Do you want to play a game? (y) or (n)"

The user types *y* if they want to play and *n* if they don't. If the user answers *y*, then your program should ask for entries for each of the missing word types in your template, such as:

"Enter a plural noun: "
"Enter a singular noun: "
"Enter an occupation: "
"Enter an animal name: "
"Enter a place: "
"Enter a verb: "

Build a new string variable that includes the user entries in place of the word-type placeholders and print the variable to show the user the new sentence.

If the user types "n" when asked if they want to play, your program should print "good bye" and exit. The user should be allowed to play the game as many times as they like. After printing the new sentence, your program should ask if they would like to play again, and if they answer *y*, your program should repeat the game, beginning at the spot where you ask the user to enter word types.

IMPORTANT NOTE: User-input to play further must be obtained inside the main function, ***not*** within your user-defined function. That is, the loop that enables the game to be played indefinitely must not be within your user-defined function - COG will not accept it.

Problem 2: Solar Energy

Solar Energy(measured in KWh) generated by a solar panel or array of panels depends on its area, efficiency, solar radiation of the location and performance ratio.
Equations to calculate solar energy output is shown here:

$$E = A * r * H * PR$$

where E is the average energy generated annually by the solar array in kilowatt-hours (kWh).

There are several other noteworthy parameters in these equations:

A	is area of the solar array in meter square
r	is the solar panel efficiency in percentage
H	is the average solar radiation in kWh/m^2/year
PR	is the performance ratio (coefficient of loss)

- usually between 0.5 and 0.9, with a default of 0.75

Part A

Write a function "**energyCalculator**" to calculate average energy output. Your main function should take in user inputs for A, r, and H, it should declare *a constant value for PR*, pass these values as parameters to your function. After your function calculates the average energy, it must **return** this value to main. To test your code, if you use *A = 20, r = 0.15 , H= 1250 as inputs and assume PR= 0.75*, you should get *2812.5 KWh*.

Within your main function, print the following cout statement:

```
cout << "The average annual solar energy production is " << average_energy <<  "kWh." << endl;
```

Part B

Once you have the average energy output calculation working, create another function **"printEnergy"** that prints all the peak energy values for efficiency (r) from 0.10 to 0.35 by 0.05 step. The function should receive A, r, H, and PR as input parameters; you may utilize the same user input for A and H as received in *Part A* above.

Use the following cout statement to output the values from *within* the function:

```
    cout << "The average annual solar energy for an efficiency of " << efficiency_value
         << " is " << annual_energy_ave << " kWh." << endl;
```

IMPORTANT NOTE: The loop that enables the iteration over different efficiency values must not be within your user-defined function - COG will not accept it.

Part C

Solar panel arrays can be installed on each house to support that individual household, or we can build solar farms with much larger solar arrays to support communities. Get user input for the average number of kWh used by an average household and calculate the number of households the specified solar array can support for a year. (Note: average usage in Colorado is ~8500 KWh per year)

Create a function **"calculateNumberHousesSupported"** that takes parameters of average energy of solar array and energy required by one household and **returns** the number of houses it will support. The function call from the main function of your program should use the average energy output as calculated in part a above and the user supplied average energy usage to calculate a floating point value.

Use the following cout statement to output the value in main function:

```
cout << number_houses_supported << " houses can be supported." << endl;
```

***Note**: The COG grading script will test with different inputs for the same Mad LIb template. Once you have your code running on COG, you are welcome to explore other approaches to generating the template, such as reading it in from a file, or having multiple templates that can be selected randomly. Doing so will make it possible to build a much more complex, and interesting, game.*

Note: Redundant code is discouraged.
See the DRY (Do not Repeat Yourself) coding principle: https://en.wikipedia.org/wiki/Don't_repeat_yourself

Points to remember when running your code in COG:

These are some things in which COG is very rigid:

1. Students *must* include *void* for the input parameter of functions that **don't** take input, rather than leaving it blank.

   For example,

   ```
   *correct*        int some_function (void) {}
   *incorrect*             int some_function () {}
   ```

2. Whenever you are prompting the user for input from ***within*** a **user-defined function**, they must end the prompt with a colon (:).

   For example,

   ```
   // Inside some_function()
   *correct*        cout << "Please enter a value for x: ";
   *incorrect*            cout << "Please enter a value for x ";
   ```

3. Answers printed from ***within*** a **user-defined function** must end with a period.

```
// Inside some_function()
        *correct*              cout << "The answer is 42.";
        *incorrect*            cout << "The answer is 42!";
        *incorrect*            cout << "The answer is 42";
```

4. In addition to #3, answers printed from *within* the user-defined function must exactly follow the cout string template given in the writeup.