Moving Ahead – Odometer Calculations

CSE 1325 – Fall 2016 – Homework #2
Due Thursday, September 8 at 8:00 am

A common form of program is the “filter” - data flows into the program, and a transformed version of the data flows out of the program. This is analogous to, though implemented differently from, C++ streams (good 'ole cout and cin). A good example is the Unix command line

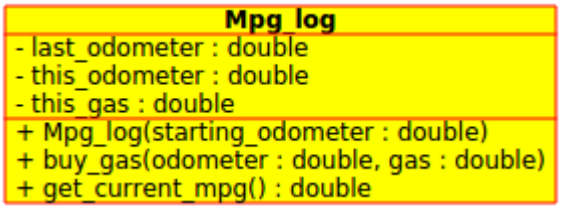
```
cat unsorted_names.txt | sort | uniq > unique_sorted_names.txt
```

The text file “unsorted_names.txt” is sent into the “sort “program, where it is (well) sorted by line; that output flows in to the “uniq” program, when any duplicate lines are removed, leaving only lines that are (well) unique; and then *that* output is written to the text file “unique_sorted_names.txt”. Filters can be chained together like this in ad hoc fashion to solve a lot of problems quickly – one of the advantages of the command line interface.[1]

In this assignment, you'll begin writing a filter in which gallons of gas purchased and the odometer reading at the time of purchase flows in, and out will flow miles per gallon statistics. This is useful for detecting when your engine needs a tune up, or to convince your family, financial advisor, or significant other that you should buy a more efficient (or better, an electric :-)) car.[2]

The Math

This program must include a class named Mpg_log and a main() function with no parameters. You may include “std_lib_facilities.h” or not, as you choose. The main function should instance Mpg_log and use the resulting object for maintaining odometer and gas data and for calculating mpg. A suggested class design is shown in UML.



Miles per gallon is calculated by dividing the number of gallons pumped into the tank by the number of miles traveled. So if your odometer when you filled up the tank last week was 100,000 miles, and today you filled the tank with 10.0 gallons and the odometer reads 100,300 miles, your gas mileage was (100,300 – 100,000) / 10.0 = 30.0 miles per gallon (MPG). (Notice that we don't care how much gas we bought last week – it doesn't matter, since that was gas we burned *prior* to our 100,000 odometer reading.)

If next week, you buy 5.0 gallons of gas and the odometer reads 100,400, your gas mileage for the week was (100,400 – 100,300) / 5.0 = 20.0 MPG.

Incidentally, your gas mileage *for both weeks* is (100,400 – 100,000) / (10.0 + 5.0) = 26.7 MPG. This is

1 Yes, graphical interfaces have many advantages, too. We'll get to them after the first exam...
2 Of course, mathematically you can rarely justify replacing a car on the basis of “saving money”. I didn't say this would provide a *good* argument; just an argument.

NOT the same as the average gas mileage, which would be (30.0 + 20.0) / 2 = 25 MPG – that's not mathematically valid or particularly useful. MPG is always “miles traveled / gallons used”.

Requirements

The program will first show a prompt (think cout <<) and then read an initial odometer value from the user (think cin >>).

Next, the program enters a loop in which it prompts for and collects subsequent odometer readings and associated gallons of gas purchased, then prints the MPG value since the last time you bought gas.

No exit command is required, but feel free to add one if you like.

```
ricegf@pluto:~/dev/cpp/P2$ g++ -w mileage.cpp
ricegf@pluto:~/dev/cpp/P2$ ./a.out
Initial odometer: 36381.1
Odometer: 36545.2
Gallons: 4.5
This mpg: 36.4667
Odometer: 36712.9
Gallons: 8.2
This mpg: 20.4512
Odometer: 36845.7
Gallons: 4.4
This mpg: 30.1818
Odometer:
```

Given

You will be provided a spreadsheet called Mileage_Log.csv with a sample log of odometer readings and gasoline purchases, including additional data that you don't need. You may use the spreadsheet to copy and paste or to rekey test data into your program (the expected values are in column E to help you with testing), or you may create your own data.

Deliverables

You will provide a file called mileage.cpp, and a screenshot in PNG, JPG, or GIF format of your interactive testing session (it need not be comprehensive).

Grading

- **Full Credit** – If you deliver a working mileage.cpp and a screenshot of the test session.
- **Bonus**[3] – In addition to calculating the MPG since the last fill up, also calculate and print to the console the MPG *since the first odometer reading entered*. Thus, you are printing out two calculated values: This MPG and Total MPG. Column F of Mileage_Log.csv provides expected values for testing this version of the program. Include mileage1.cpp, a screenshot of testing, and a screenshot of your class design in UML, including both private and public data and methods.
- **Advanced Bonus** – In addition to calculating This MPG and Total MPG, calculate a Rolling Average MPG for just the most recent 5 gasoline purchases. This value will be the same as Total MPG for the first 5 purchases, but will then deviate. You may use vectors or discrete variables (or a data structure that we have not yet covered) to hold the most recent 5 odometer and gasoline purchase values. Column G of Mileage_Log.csv provides expected values. Include mileage2.cpp, a screenshot of testing, and a screenshot of your class design in UML.

3 You will need additional private data and public method(s) for the Bonus level, even more private data and public method(s) for the Advanced Bonus level, etc. Designing the class for the bonus levels is part of the requirements.

- **Extreme Bonus** – Create a true filter program that accepts a stream of data (in CSV format) from standard input, and prints a stream of data (in CSV format) to standard output that includes the original unmodified data but with columns added for This MPG, Total MPG, and Rolling Average MPG[4]. You may omit headers for the columns. It should be capable of handling the sample log file provided with column headers and the rightmost 3 spreadsheet-calculated columns removed. Give some thought to handling exceptions that may be caused by bad input, perhaps by ignoring the bad line of data and printing a warning to standard error. Include mileage3.cpp, a screenshot of testing (which has now been semi-automated), and a screenshot of your class design in UML.

4 Hey, look – it's a *filter*!