Lab 1 – Introduction to C++

Computer Science 2413 – Data Structures – Fall 2019
Due by 11:59 pm CST on Thursday, 29 August 2019

This lab is individual work. Each student must complete this assignment independently.

User Request:

"Create a simple program to read and write drilling data."

Objectives:

1	Create and correctly submit a C++ source code file (named "`Driller.cpp`").	*5 points*
2	Use the standard C++ input and output streams (`cin` and `cout`) to read and write data.	*5 points*
3	Use C++ to do minor processing of input.	*5 points*
4	Use proper coding style, documentation, and formatting.	*5 points*

Description:

Modern drilling of wells involves substantial collection of data from embedded sensors. This data may be used by operators of the drilling equipment to control various aspects of the drilling process, such as the rotational speed at which the drill turns. In the future, this data might be used in real time by automatic control systems to optimize the drilling process for more efficient drilling than is possible for human operators. For this lab, you will put together several techniques and concepts you have learned in CS 2334 (or from a similar background) and some new techniques to make *Driller*, a program that reads, processes, and outputs data from these files of this sensor data. Driller version 0.1 (which satisfies the requirements of this lab) will read through a file of drilling data (using redirected `cin`), perform simple processing on the data, and send the resulting output to `cout`.

Operational Issues:

Your program will read the drilling data file—a text file in which data fields are separated by commas. The data file will be organized as follows:

The first row consists of column headers and should be ignored. The remaining lines of the file contain the data. Each data row contains data on a single second of drilling. Driller will not know in advance the number of data rows present in the file and there may be data missing for some seconds, minutes, or even hours. However, for this lab, you only need to read, process, and output one line at a time, so it should not be necessary for Driller to know the total number of rows in advance. The data in these rows will be in 18 columns (that is, separated by 17 commas) per row. Most of the columns contain data that should be internally represented as floating-point data, but the first two columns contain data that should be internally represented as strings.

As Driller reads the data file, there are three correctness checks that it must make on the data.

First, Driller should ensure that all data in the file comes from the same date by ensuring that the date stamps (found in the first column) are the same. If, while reading the file, Driller encounters a date stamp that does not match the date stamp found on the first data line of the file, it must provide an error message indicating that it encountered a non-matching date stamp and the line number at which the non-matching date stamp was found, then move on to reading, processing, and outputting the next line in

data file. The exact format of the error must be as follows:

`Non-matching date stamp` *`d`* `at line` *`n`*`.`

Naturally, *`d`* should be replaced by the non-matching date stamp and *`n`* should be replaced by the line at which the non-matching date was found, counting the first data line (not the header line) as line 1. Lines at which non-matching date stamps are encountered *do* increment the line count.

Second, if the date stamp is valid (matches the date stamp of the first data line) Driller must ensure that the time stamp (found in the second column) of each data line is unique (not repeated). If Driller encounters a duplicate time stamp while reading the data file, it must provide an error message indicating that it encountered a duplicate and the line number at which the duplicate was found, then move on to reading, processing, and outputting the next line in data file. The exact format of the error must be as follows:

`Duplicate time stamp` *`t`* `at line` *`n`*`.`

Naturally, *`t`* should be replaced by the duplicated time stamp and *`n`* should be replaced by the line at which the duplicate was found, counting the first data line (not the header line) as line 1. Lines at which duplicates are encountered *do* increment the line count. Note that, since Driller checks first for non-matching date stamps, then for duplicated time stamps, this second possible error should be checked for only in the case that the first error was *not* encountered. Moreover, time stamps from data lines with non-matching date stamps should *not* be added to the list of date stamps encountered, since those data lines are invalid and should be further processed.

Third, if the data stamp is valid (matches) and the time stamp is valid (not a duplicate), Driller should verify that all floating-point values in the data are greater than zero. If any of the floating-point values are zero or negative, Driller must provide an error message indicating that it encountered an invalid data value and the line number at which the invalid data was found, then move on to reading, processing, and outputting the next line in data file. The exact format of the error must be as follows:

`Invalid floating-point data at line` *`n`*`.`

Naturally, *`n`* should be replaced by the line at which the invalid data was found, counting the first data line (not the header line) as line 1. Lines at which invalid floating-point data is encountered *do* increment the line count and *do* have their time stamp added to the list of unique time stamps already read in.

As the data is read in, line by line, and processed to check for duplicates and invalid values, Driller will send it back out again, albeit in a slightly altered form. Rather than comma separated, the output data will be separated by semicolons. Lines containing duplicate time stamps or invalid data values will not be output, only errors messages will be output for those lines.

Implementation Issues:

You must create and submit C++ source code to complete this lab (as well as all other labs and projects in this course).

You will use the standard C++ input and output streams (`cin` and `cout`, respectively) for all input and output for this assignment. To test your code on the data file, you will want to tell Driller that you wish to redirect `cin` to read from your data file. This can be done from the configurations within your Driller project. Note that you do not need to consider the standard C++ error stream (`cerr`) for this assignment.

Note that this lab does not require you to create any classes or to use object-oriented design at all. In fact, you don't even need to create any functions except the `main` function. Nonetheless, you must use

good programming style and documentation, including making your code modular, using explanatory comments for each section of code, using meaningful variable and method names, using consistent indentation, etc.

You may write your program from scratch or may start from programs for which the source code is freely available on the web or through other sources (such as friends or student organizations). If you do not start from scratch, you must give a complete and accurate accounting of where all of your code came from and indicate which parts are original or changed, and which you got from which other source. Failure to give credit where credit is due is academic fraud and will be dealt with accordingly.

Due Date:

You must submit an electronic copy of your Driller project to the appropriate lab in your zyBook by **11:59 pm CST on Thursday, 29 August 2019**.