Homework 2: A Histogram Class

CSCI 60
Krehbiel

Due 11:59pm Wednesday, October 8, 2025

Overview. This program will have you implement in `histogram.cpp` a class whose (complete) interface is given in `histogram.h`, and you will write a driver program for it in `main.cpp`. The `Histogram` class stores the data associated with a frequency histogram, and this data is accessible via overloaded operators and other member functions. This handout overviews what is in `histogram.h`, which you should not change, gives detailed instructions for implementing each of the required functions in `histogram.cpp`; and it gives general guidance for how to test as you go in `main.cpp`. You should employ object-oriented development and do your own unit testing in the `main.cpp` driver program as you write each function – don't wait to test until the very end! At no point should change anything in `histogram.h` or any of the boilerplate code at the top of each of the other files. Compile your code as follows:

```
g++ main.cpp histogram.cpp -std=c++11
```

You will turn in a fully implemented `histogram.cpp` and a `main.cpp` program that calls each of the `Histogram` functions you implemented. Look over the submission checklist from HW1 for usual submission instructions and style guidelines.

The interface. The interface file `histogram.h` is fully written; the autograder will run your submitted cpp files with the provided header file, so you should neither upload nor modify this file as you work on your implementation and test program. You will lose substantial points if your implementation file relies on changes you made to the interface file. The next paragraph helps orient you about what is in this `histogram.h` implementation file, which you'll have to understand in order to implement the class in `histogram.cpp`.

The interface shows that the `Histogram` class definition declares a single private member variable, an array called `counts`. Array size is determined by `public static constant size_t MAX`, which specifies the largest value to be expected in any data file[1]. After construction, `counts[v]` should contain a non-negative integer representing the number of occurrences of the value `v` in a specified data file for every `v` from `0` to `MAX`, or it should be an all zeros array if no arguments are provided for construction. *Draw some examples of arrays and their contents for different datasets and smaller values of* `MAX`.

The public interface allows a class client to query the histogram's size, min, max, mode, median, mean, and variance. Additionally, it declares overloaded operators so users can add the counts of one histogram to another, access the count of a specific value using array-style syntax, and insert a text representation of the histogram into an output stream. Class this week will provide examples of each of these overloaded operators.

The implementation. Your primary task is to implement all the histogram functions declared in the interface. You can tackle these tasks in the following stages:

1. The 0-argument constructor should simply initialize all the values of the counts array to zero. The 1-argument constructor should read in a stream of whitespace-separated integers between `0` and `MAX`, keeping track of the number of occurrences of each value.

   The sample code in `main.cpp` provides some (highly documented) syntax you can repurpose for this file-reading task. Recall that the counts array is not guaranteed to be initialized to zero, so you should initialize these values before you start reading values in from the file.

   When you are done, you can test your constructor implementation by temporarily adding `cout` statements to the body of the constructor to report the counts after reading the file. A small file `data.txt` is provided for you if you wish to use it.

[1] We will briefly discuss the `const` keyword, `static` variables, and the `size_t` type in class this week, but all you need to know for now is that `const` prevents variables from changing, `static` variables are the same across objects, and `size_t` works like a non-negative `int`.

2. Implement the member functions size, min, max, mode, median, mean, and variance, testing each one before moving on to the next.

   - The `size` of the histogram is the sum of the counts across all values.
   - The smallest and largest *possible* values anticipated in a data file are `0` and `MAX`, respectively, but the functions `min` and `max` should respectively return the smallest and largest values *with positive count*. If the histogram is empty, these functions should return `MAX+1`.
   - The `mode` is the value that occurred with highest frequency; if there is a tie, you may return any of the values with highest frequency.
   - The `median` is the 50th percentile value. For example, if there are 5 entries, then the third largest is also the third smallest, and that value should be returned. If there are 4 entries, then you may return either the second largest or the second smallest, which may or may not be the same value.
   - The `mean` is the sum of all the values (with repetition) divided by the total number of values, and the `variance` is the mean squared difference between the values and their mean.

   For example, the sample `data.txt` file contains one 1, four 2s, two 3s, and one 9. The corresponding histogram size is 8, the min is 1, the max is 9, the mode and median are both 2, the mean is 3.0, and the variance is $(1 \cdot (1-3.0)^2 + 4 \cdot (2-3.0)^2 + 2 \cdot (3-3.0)^2 + 1 \cdot (9-3.0)^2)/8 = 5.5$.

3. Overload the += operator as a member function. The command `hist1 += hist2;` should not change `hist2` (which is why the parameter is declared const) but it should add each of the counts in `hist2` to the corresponding count in `hist1` (which is why the function is not declared const).

4. Overload the `[]` operator as a member function. This command `hist[val]` should return the count of the value `val`.

5. Overload the << operator as a non-member function. This should insert several lines into the specified ostream, with each line labeled with its value, a colon, and a string of stars representing the count of that value as done in your first programming assignment. The first line should be the smallest value with positive count and the last line should be the largest value with positive count.[2] The commented out code in your starter `main.cpp` should produce the following output[3] when you are done:

```
1: *
2: ****
3: **
4:
5:
6:
7:
8:
9: *
```

The test program. Use the `main.cpp` file to test your implementation as you go. The test program is worth a minority of the points, and the only firm requirements are that it must compile and that it must call each of the functions defined in your implementation file at least once. For full credit, your testing should be thorough, well-organized, and easy to read. Skim the Homework 1 style guidelines before you submit to make sure you don't lose any style points, including for adding appropriate information to file comments!

[2] You may but don't have to ensure perfect spacing to get the colons to line up if some values have more digits than others, but you *do* need whitespace between the colons and the stars for full credit from the autograder.

[3] Note that lines for values < 1 and > 9 are intentionally *not* output!