CMSC 341 - Project 0: Using GL, Memory Management, Testing, and Efficiency - Fall 2021

Due: Tuesday Sep 14, before 9:00 pm

There are no late submission folders for Project 0. You must turn it in on-time to receive credit.

Addenda

- 09/07/2021 - Modification to source files: compiling the files with -Wall option generates warning. These warnings will not affect your work and you can continue. However, the files `hist.h` and `driver.cpp` are modified to remove the warnings. You can download the latest from the links on this page.
- 09/02/2021 - Clarification: a table added to the introduction section to clarify the grouping of grades by buckets, and the related description is modifed.

Objectives

- Review the procedures to access the GL servers and to compile programs on GL.
- Review C++ memory management (allocating and deallocating memory dynamically), including copy constructors, destructors, and assignment operators.
- Use Valgrind to check for memory leaks.
- Learn how to test a project.
- Learn how to measure the efficiency of algorithms.
- To ensure that you are able to submit project files on GL.

Introduction

In this project, you will complete a C++ class by writing a copy constructor, destructor, and assginment operator. Furthermore, you will write a tester class and a test program and use Valgrind to check that your program is free of memory leaks. Finally, you will submit your project files on GL. If you have submitted programs on GL using shared directories (instead of the `submit` command), then the submission steps should be familiar.

In this project we develop an application which takes student grades as input and produces histograms. Since we need a large amount of data, we use a random number generator to produce random grades instead of input data. There are three classes in this project.

- The `Random` class generates data. The implementation of this class is provided. It generates grades with normal distribution. To create an object of this class we need to specify min and max values for the constructor. The grades fall between 0 and 100 inclusively. The class only generates integer numbers.
- The `Data` class stores and outputs the histogram for a grading item. You implement the Data class. In this context, a grading item could be a quiz, a test, an exam, etc. The following figure represents the sample output of this class. In this figure, there are 3 stars next to 30, it means there are 3 grades between 20 and 29 for this grading item. There are 6 stars next to 80 which means there are 6 grades between 70 and 79. The grades are classified in 10 groups. The following table shows the grade range to be counted in every group.
- The `Hist` class stores and outputs the histogram for multiple grading items. For example, a professor can use this class to draw the histograms for all grading items in a course. You implement the Hist class.

```
Histogram for item 1:
 10
 20
 30 ***
 40 *****
 50 ****
 60 ******
 70 **********
 80 ******
 90 **
100 **
```

grade	bucket name
0-9	10
10-19	20
20-29	30
30-39	40
40-49	50
50-59	60
60-69	70
70-79	80
80-89	90
90-100	100

Assignment

Step 1: Create your working directory

Create a directory in your GL account to contain your project files. For example, `cs341/proj0`.

Step 2: Copy the project files

You can right-click on the file linkes on this page and save-as in your computer. Then, transfer the files to your GL account using SFTP.

For this project, you are provided with the skeleton .h and .cpp files, a sample driver, and the output of driver program:

- hist.h - The interface for all three classes, i.e Random, Data, and Hist.
- hist.cpp - The skeleton for the Data class and the Hist class implementation.
- driver.cpp - The sample driver program.
- driver.txt - The output of sample driver program.

Step 3: Complete the `Data` and `Hist` classes

The `Data` class

The Data class has a member variable which stores the counts of grades for a grading item. The member variable m_data is a pointer to an array of integers and requires memory allocation. We split the grades into 10 categories. The total count of grades between 0 and 9 will be stored in array cell with index 0. The count of grades between 10 and 19 will be stored in array cell with index 1 and so on. The count of grades between 90 and 100 will be stored in the array cell with index 9.

Data::Data()	This is the constructor. It allocates memory to m_data with the length of DATAARRAYSIZE.
Data::~Data()	This function deallocates the memory.
void Data::simulateData(int students)	The students parameter indicates the number of grades this function generates, counts, and stores the counts in the m_data cells. Every cell of m_data array stores the count of grades in a specific category. For example, if the grade is between 10 and 19, the function increments the count in m_data[1]. The function generates random grades using a

	Random object. The grades are between MINGRADE and MAXGRADE values.
void Data::dump(int item)	This function outputs the histogram similar to the above figure. For every cell in m_data the function prints out stars using the number stored in the cell. The item parameter indicates the item number. This function will be called in Hist::dump() to print out a histogram for all grading items.

The `Hist` class

The Hist class stores histogram data for multiple grading items. The member variable m_table is a pointer to an array of Data objects and requires memory allocation.

Hist::Hist(int items)	This is the constructor. It allocates memory to m_table with the length of items. If user passes a value less than 1, the number of items should be set to zero. In such a case we do not need to allocate memory.
Hist::~Hist()	This function deallocates the memory.
void Hist::simulateData(int students)	This function calls Data::simulateData(int students) to populate every cell of m_table. Every cell of m_table represents a grading item. The parameter students indicate the number of grades. If the number of items is less than 1, this function throws a std::out_of_range exception.
Hist::Hist(const Hist & rhs)	This is the copy constructor. It creates a deep copy of the rhs object. Deep copy means the new object has its own memory allocated.
const Hist & Hist::operator=(const Hist & rhs)	This is the assignment operator. It creates a deep copy of rhs. **Reminder:** an assignment operator needs protection against self-assignment.

Step 4: Test your code

You must write and submit a test program along with the implementation of the Data and Hist classes. Since the Hist class depends on the Data class, you need to make sure your implementation of the Data class is working correctly. You may achieve this by testing the Data class properly. You do not need to submit your tests for the Data class. You only submit the tests for the Hist class. To test your Hist class, you implement your test functions in the `Tester` class. The `Tester` class resides in your test file. You name your test file `mytest.cpp`. It is strongly recommended that you read the testing guidelines before writing test cases. A sample test program including `Tester` class, and sample test functions are provided in driver.cpp. You add your Tester class, test functions and test cases to `mytest.cpp`. The following two test functions are provided to you in driver.cpp in the Tester class. We recommend that you study the implementation of these functions.

There is no need to test a program exhaustively, however, any program should be adequately tested. Adequate testing includes normal cases, edge cases and error cases. The following list presents some examples.

- Trying to create a Hist object with -1 items is an error case.
- Trying to create a Hist object with 0 items is an edge case.
- Trying to create a Hist object with 1 item is an edge case.
- Trying to create a Hist object with 100 items is a normal case.

bool Tester::testCopyConstructor(const Hist& histogram)	This function tests the correctness of copy constructor. If the copy is performed correctly, the function returns true, otherwise it returns false. There are multiple cases to check for the correctness of copy operations. For example, a copy constructor should make a deep copy, then the test function should check for that. Or the corresponding values of the two copies should be equal, then the test function needs to check for it. The test function also should account for the edge cases. For example, the object being copied can be an empty object.
void Tester::measureSimulationTime(int numTrials, int items, int students)	An important matter in data structures is the efficiency or running time of algorithms. This test function is an example for analyzing the running time of the data simulation in a Hist object (the call to the Hist::simulateData(int students) function). The running time grows/scales with the amount of data. In this algorithm we expect that the running time grows linearly corresponding to the growth of the total number of grades. In

	the Hist class the total number of processed grades is defined by the number of grading items multiplied by the number of students. In multiple runs of the function If we increase the total number of grades by a factor of 2, we expect that the running time increases by a factor of 2.

Your test program must test the correctness of assignment operator, and proves that the running time of your copy constructor grows linearly;

- Check that a copy is made. The new Hist object should contain exactly the same data as the source object.
- Check that the copy is *deep*.
- Check *edge cases*. For example, do they work correctly if the source object is empty?
- For the assignment operator, check that you have guarded against self-assignment.
- Write a test function that measures the running time of the copy constructor. Run the copy constructor with at least 3 trials, and compare the measured times.

Step 5: Check for memory leaks

Run your test programs using Valgrind. For example, assuming you have compiled `mytest.cpp`, producting the executable `mytest.out`, run the command

```
valgrind mytest.out
```

If there are no memory leaks, the end of the output should be similar to the following:

```
==8613==
    ==8613== HEAP SUMMARY:
    ==8613==     in use at exit: 0 bytes in 0 blocks
    ==8613==   total heap usage: 14 allocs, 14 frees, 73,888 bytes allocated
    ==8613==
    ==8613== All heap blocks were freed -- no leaks are possible
    ==8613==
    ==8613== For lists of detected and suppressed errors, rerun with: -s
    ==8613== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

The important parts are “in use at exit: 0 bytes” and “no leaks are possible.” The last line is also important as memory errors can lead to leaks.

Step 6: Link your shared directory

Follow the instructions on the Project Submission page to make a symbolic link to the shared directory in your home directory.

Step 7: Submit your files

See the “What to Submit” section, below.

Implementation Notes

Each project has a section on implementation notes. These point out some issues that you might encounter while developing your code. You should look through the Implementation Notes before you start coding.

For Project 0, there are only a few notes:

- The class declaration (`Hist`) and provided function implementations in `hist.cpp` may not be modified in any way. No additional libraries may be used, but additional `using` statements are permitted.
- The locations for the function implementations are clearly marked in `hist.cpp`. They must be written at the specified locations; in particular, they must not be written “in-line.”
- Private helper functions may be added, but must be declared in the private section of the `Hist` and/or `Data` class. There is comment indicating where private helper fuction declarations should be written. **Note:** in this project, the private functions which are already declared, provide you the required functionality.
- You should read through the coding standards for this class.

What to Submit

You must submit the following files to the `proj0` submit directory:

- `hist.h`
- `hist.cpp`
- `mytest.cpp` (**Note:** This file contains the declaration and implementation of your Tester class as well as all your test cases.)

If you followed the instructions in the Project Submission page to set up your directories, you can submit your code using the following command:

```
cp hist.h hist.cpp mytest.cpp ~/cs341proj/proj0/
```

Grading Rubric

The following presents a course rubric. It shows how a submitted project might lose points.

- Conforming to coding standards make about 10% of the grade.
- Correctness and completeness of your test cases (mytest.cpp) make about 15% of the grade.
- Passing tests make about 30% of the grade.

If the submitted project is in a state that receives the deduction for all above items, it will be graded for efforts. The grade will depend on the required efforts to complete such a work.
