ECE 150 Lab 8: States, Structs, Stats and Strings

Due: Section 001: Monday, November 20th at 10:00 PM
Sections 002/003: Thursday, November 23rd at 10:00 PM

This lab consists of 3 questions worth 3 marks.

If you are in Section 001 you submit your work to http://marmoset01.shoshin.uwaterloo.ca

If you are in either Section 002 or Section 003 you submit your work to http://marmoset03.shoshin.uwaterloo.ca

Pithy Latin Quote #16

1 Objectives

This lab is intended to have students:

- Implement a priority queue as a linked list
- Implement a Dataset Class
- Implement a discrete-time simulation

Learning Outcomes

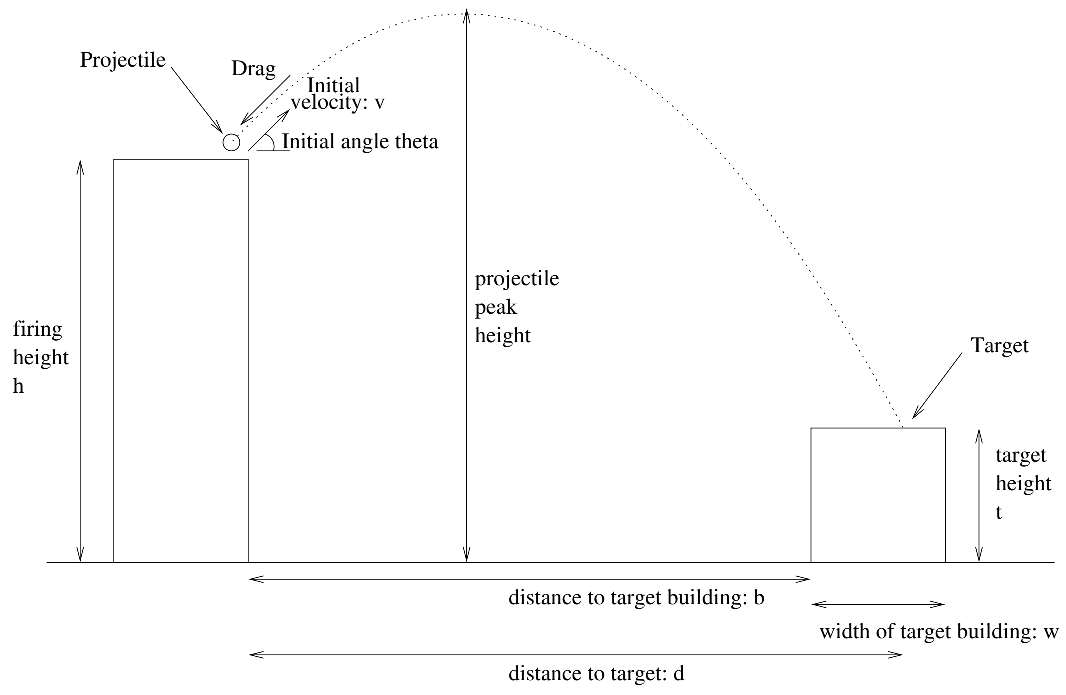
After completing this assignment, students should be familiar with the following:

1. Priority queues using struct
2. Class implementation
3. Discrete-time simulation

1. Projectile Aiming with Air Resistance [1 marks]
In Lab 7, Question 1 you were required to compute either the initial velocity or the firing angle necessary to hit a target. In doing so you assumed that air resistance, also known as drag, was not a factor in where the projectile landed. For this question you will need to include the effect of drag in your computation. However, we will drop the requirement to compute the angle for a given velocity and focus solely on computing the velocity for a given angle, since that is what you will be doing for your Physics lab. Per the previous two labs, you will be given:

1. Initial position: (0,h)
2. Mass of projectile: m
3. Location of target: (d,t)
4. Distance to target building: b
5. Width of target building: w
6. g is 9.8 m/s^2.

In addition, you will be given the firing angle θ as well as a drag constant, k. An illustration of the situation is as follows:



The drag force F_D of a projectile in a fluid[1] is proportional to the square of the velocity of the projectile, but in the exact opposite direction. This is illustrated in the above diagram with the arrow labeled "Drag." The drag equation in fluid dynamics is:

$$F_D = \frac{1}{2}\rho\, v^2\, C_D\, A$$

where ρ is the fluid density, v is the projectile velocity, A is the cross-sectional area of the projectile, and C_D is a dimensionless drag coefficient, which varies according to the projectile shape.[2] For

[1] In physics, air is a fluid.
[2] See https://en.wikipedia.org/wiki/Drag_(physics), https://en.wikipedia.org/wiki/Drag_equation and https://en.wikipedia.org/wiki/Drag_coefficient for further details.

our problem, we combine ρ, A, and C_D into a single constant k:

$$k = \tfrac{1}{2}\rho C_D A$$

which becomes a paramter we pass to our function along with other parameters, described above. The drag force is then:

$$F_D = k v^2$$

Given that this force varies with velocity, and is combined with the gravitational force, calculating an algebraic solution to determine where the projectile lands is non-trivial and possibly not possible.[3] An alternative approach is to perform a *discrete-time simulation*. Consider the projectile at time t. It has a particular position and velocity (which can be decomposed into a horizontal and vertical velocity). At that position and velocity, it has two forces acting on it: gravity and drag. Gravity can be simply seen as g acting downward. The drag force can be calculated, which then enables the drag acceleration[4] to be calculated and decomposed into horizontal and vertical acceleration amounts. The vertical drag acceleration component can be combined with the acceleration caused by gravity to determine a net vertical acceleration. Thus we now have at time t a position (x_t, y_t), velocity (v_{x_t}, v_{y_t}), and acceleration (a_{x_t}, a_{y_t}) for the projectile. For a very small time period Δt we can assume that the velocity and acceleration are constant, allowing us to calculate the new position:

$$x_{t+\Delta t} = x_t + v_{x_t}\,\Delta t$$

and velocity:

$$v_{x_{\Delta t}} = v_{x_t} + a_{x_t}\,\Delta t$$

(These equations just show the equations for the x-axis; you would need to do likewise for the y-axis.)

Given that we can calculate the new position and velocity, we can repeat this process in a loop, allowing us to calculate where the projectile will land. In doing such a calculation, we refer to Δt as the "step size."

For this assignment, you will implement a function that performs a discrete-time simulation to compute the necessary initial velocity to hit the target given the above parameters, including the drag constant k and the step size `step`:

```
bool hitTargetGivenAngle(const float h, const float m,
                         const float theta, const float d,
                         const float step, const float k,
                         const float t, const float b, const float w
                         float& v);
```

The function should return `true` if it is able to calculate the initial velocity, v, of the projectile that will enable it to land on the target, and in that case the velocity should be set in `v`. If for

[3]At a minimum, it requires expressing the problem in differential equations and solving those equations, which is well beyond the 1A level.

[4]Technically deceleration, since drag can never speed something up.

whatever reason the function cannot calculate an initial velocity necessary to hit the target for the given initial angle θ, it should return `false`.

Given that you are implementing functions, we will provide a template `main()` function in the file `Drag.cpp` that you may edit as you see fit to test your functions.

All code will be subject to style checking.

Submission: You should submit the result of this exercise to the project A8-Drag. Your submission file must be named **Drag.cpp**. Note that the filename is *case sensitive*.

2. Homework Management Priority Queue [1 marks]
Imagine having homework tasks to complete. You receive each assignment on a sheet of paper and simply stick it on your desk on top of the previous homework handouts. When you have time to do homework, you grab the first piece of paper at the top of the pile. The problem with this strategy is that you are doing your homework in the reverse order than it is handed out. The chances are very good that you will not get to the bottom of the pile before the item there has become overdue. A better strategy is to take items from the bottom of the pile first, since you will then at least be doing the work in the order in which it is handed to you. While this strategy is better, it is not ideal, since some homework may be handed out early with a late due date, while other homework is assigned not long before it is due. The best strategy is to do the assignments in the order of the due date of the assignment.[5]

For this question, we will imagine that assignments are handed out in the form of a simple structure:

```
enum COURSE {CHE102, MATH115, MATH117, ECE105, ECE150, ECE190, NULL};

struct Assignment {
    COURSE course;        // CHE102, MATH117, etc.
    int    assnNum;       // Assignment Number
    int    dueMonth;      // Month due (1-12)
    int    dueDay;        // Day due (1-31)
    char*  description;   // Assignment description
};
```

Your task is to implement the following functions, which will maintain a linked list of assignments, ordered by the due date of the assignments, and only allow the removal of the assignment with the earliest due date.

```
bool enqueue(HomeworkQueue*& queue, const Assignment& assignment);
const Assignment* dequeue(HomeworkQueue*& queue);
int daysTillDue(const HomeworkQueue* q, const COURSE course);
const Assignment* dueIn(const HomeworkQueue* q, const int numDays);
```

The `enqueue()` function takes an Assignment reference and adds it to the queue, ordered by the due date of the assignment. The `enqueue()` function returns true if the operation is successful and false if it fails for any reason. The function must also take a reference pointer to the `HomeworkQueue` because it is possible that the assignment being added will affect that pointer. If two assignments have the same due date, they should be ordered according to when they were added to the queue.

The `dequeue()` function takes a reference pointer to the `HomeworkQueue`, removes the `Assignment` from the front of the queue, adjusting the `HomeworkQueue` reference pointer accordingly, and

[5]This scheduling strategy is known as "Earliest Deadline First" and is provably optimal provided the system is not overloaded.

returns a pointer to that assignment. It should then delete the `HomeworkQueue` node that is no longer used. If the queue is empty, the `dequeue()` function returns the NULL pointer (*i.e.*, it returns 0).

Some courses are more difficult than others, and so we would like to know how many days before the next assignment is due in some particular course. The `daysTillDue()` function serves this purpose. It takes the `HomeworkQueue` pointer and a course name (*i.e.*, the enum type for that course), and finds the number of days from today until the next assignment is due for that course. To implement this function you will need to find out details about the `gettimeofday()` and `localtime()` functions. If no matching assignment is found, the function should return `INT_MAX`.

Just as some courses are more difficult than others, some weeks are busier than others, and so we would like to know what assignments we have to complete over the next N days. The `dueIn()` function serves this purpose. It takes the `HomeworkQueue` pointer and a number of days, N and returns a pointer to an array which should contain the `Assignments` due over the next N days. As with the previous function, you will need to find out details about the `gettimeofday()` and `localtime()` functions. Since we want to just return a pointer to an array of `Assignments` we need some way to indicate the end of this array. We will do so by creating an artificial last assignment, at the end of the array, which has the artificial course designation `NULL`.

You may find it useful to create a helper function as follows:

```
bool isEarlier(const Assignment& a1, const Assignment& a2);
```

which returns true if a1 is due earlier than a2, and false otherwise.

Given that you are implementing functions, we will provide a template `main()` function in the file `Homework.cpp` that you may edit as you see fit to test your functions.

All code will be subject to style checking.

Submission: You should submit the result of this exercise to the project A8-Homework. Your submission file must be named **Homework.cpp**. Note that the filename is *case sensitive*.

3. StudentGrades dataset as a class [1 marks]

In Lab 7, Question 3 you created several functions that read a CSV containing grade data, computed statistatics over that data, and wrote those statistics to a file. While this is an improvement over an unstructured data approach, there are still several problems with it. First, the "contract" between the code that *uses* these functions and structures and the code that *implements* them is quite unclear. Second, the code that *uses* the structures requires that they not change; more precisely, if the structures change, then the code that *uses* the structures must also change. This is highly undesirable. We would like to be able to change the *implementation* of the functions (and thus of the structures supporting those functions) without having to change the code that wishes to *use* those functions. To do so, we move from *structured programming* to *object-oriented programming* and define the notion of a *class* which specifies an *interface*. The class can be *used* by code that knows only the *interface*, without knowing the *implementation*.

For this question, we define the class *StudentGrades* which is capable of reading a file of grades, computing particular statistics over those grades, and writing reports on those statistics. The class definition is as follows:

```
class StudentGrades {
public:
    // Supported statistics
    int   minimum();
    float average();
    int   maximum();
    float popStdDev();
    float smplStdDev();
    int   numModes();
    int   mode(const int modeNumber);
    int   histogram(const int bucketNumber);

    // Populating the data and writing out results
    int  readCSV(const char filename[]);
    int  writeStats(const char filename[]);

    // Some supporting functions
    bool validStudentIDs(const int minAcceptableID, const int maxAcceptableID);

    int  numRejects();
    int  reject(const int rejectNumber);

    // Constructor and Destructor
    StudentGrades();
   ~StudentGrades();

private:
    // It is up to you what you store here
};
```

The statistics methods are what you would expect them to be, doing the same as was required of them in Lab 7, with two caveats:

1. The mode calculation is split into two separate functions: `numModes()` and `mode(int)`. The first of these returns the number of modes in the student data set; the second returns the particular mode requested, from 1 to N, where N is the number of modes. If there are no modes (*e.g.*, because there is no data because `readCSV()` has not yet been invoked), then `numModes()` should return 0. If there are N modes, then invoking `mode(i)` with $i > N$ is undefined. You cannot assume that `numModes()` will be invoked before `mode()` and it is expected that `mode()` will return a correct value if it is invoked over a mode that exists (*e.g.*, if there is at least one mode and `mode(1)` is invoked, it should return the correct result. The modes should be in sorted order.
2. The `histogram()` function takes a bucket number as input (from 1 to 10) and outputs the number of grades in that bucket. If an invalid bucket number is passed to the function, it should return -1.

The `readCSV()` and `writeStats()` functions read and write the file, repsectively, per the requirements from Lab 7. While writing is essentially as was the case in Lab 7, there are some changes in the case of reading the file. First, the acceptable studentID range is not passed in the `readCSV()` function. Instead, all studentIDs and grades in the valid range of 0 to 100, inclusive, should be kept in the object instance. Rejected data should be identified by line number in the CSV file and that information kept by the object instance. The `numRejects()` and `reject(int)` methods allow querying of this data.

The method `validStudentIDs()` is used to set the acceptable studentIDs, and it may be used to change that acceptable set any time it is invoked. All statistics calculated (including those that would need to be calculated if the "`.stat`" file is to be written) must take into account the acceptable studentID range. If this method has not been invoked prior to any statistics being calculated, then the statistics should be calculated over all of the currently stored data, if any.[6]

Given that you are implementing a class, we will provide a template `main()` function in the file `StudentGrades.cpp` that you may edit as you see fit to test your functions.

All code will be subject to style checking.

Submission: You should submit the result of this exercise to the project A8-StudentGrades. Your submission file must be named **StudentGrades.cpp**. Note that the filename is *case sensitive*.

[6]There may be no data stored since the `readCSV()` function may not yet have been invoked.