Homework 6, CSCE 240, Summer 2016

NOTE: We will spend a moderate amount of time in class discussing this program and the various good and bad ways in which it could be written. It is often said that choosing the right data structure makes programming (relatively) simple, and choosing the wrong data makes programming (relatively) hard.

Overview

This is an exercise in using the `multimap` to organize information as keys and values.

You are to write a basic discrete event simulation program.

The underlying motif is that of voters arriving at a polling place and then voting on one of "some number" of voting machines.

The input data will be:

1. the number of voters who actually vote in the precinct;
2. the minimum number of voting machines to simulate;
3. the maximum number of voting machines to simulate;
4. the number of time steps in an election day;
5. the mean time (average time) a voter takes to vote;
6. the standard deviation from the mean for a voter's time to vote.

The main program is given to you.

1. Read the data.
2. Initialize the simulation.
3. Run the simulation.

Discrete Event Simulation

In a "discrete event" simulation, which is what this is, events happen in discrete, separate, individual units. A voter's arrival is a discrete event for which we count "one" for the number of voters. The voter's being assigned to a voting machine is a discrete event, as is the voter's finishing voting and leaving. This program therefore is similar to what would be the program for tracking customers and hamburgers at a fast food restaurant, simulating shoppers in a checkout line, or messages arriving at an internet router.

In all such simulations, we have various parameters to consider.

1. The number of transactions to process. In this case, one voter is one transaction.

2. The distribution of the arrival times of the transactions. Usually, we assume a Poisson distribution of times between arrivals of transactions (you will learn a little about Poisson distribution in STAT 509). In this case, we will assume instead a uniform distribution of arrivals. That is, if we assume the Election Day is 12 hours long, then 1/12 of the voters will arrive, uniformly spaced, in each hour, 1/720 of the voters in each minute, and so forth.

3. The various steps involved in a single transaction. In this case, entering the line, being assigned a voting machine, and taking some time at the voting machine.

4. The distribution of the transaction times (the time it takes to vote). In this case, we assume a normal (or Gaussian) distribution (this is the famous Bell curve) about a mean of three minutes, because South Carolina state law says that a voter should only be permitted three minutes to vote.

5. The length of the queue that is permitted. In this case, we assume that the queue can be infinitely large. In the case of an internet router, the maximum queue size is the size of the buffer set aside for incoming messages, and the notion of a Denial of Service attack on a router is to send the router more messages than it can handle given its service time and the buffer size that is configured for the router.

6. The queueing model that is used. It can be mathematically proven that one queue leading to multiple servers results in shorter average

wait times than multiple queues for multiple servers. That is, the usual supermarket checkout process in which each person queues up to a specific server is NOT the most efficient way to get people checked out.

We will assume one queue and multiple voting machines.

For what it's worth, we are going to do this simulation with INTEGER units of time equal to ONE-TENTH of a minute. That is, the basic unit of time is six seconds. The reason for this is that integer numbers of minutes is too coarse, and integer numbers of seconds is too fine (it makes for too much output). Our election day is thus 12 hours = 720 minutes = 7200 time steps. And so forth.

Random Numbers

You have been given a class `MyRandom` that has three functions that generate random numbers. One function generates normally distributed random numbers when given the mean and standard deviation as parameters. This function returns a random value that is a `double`. Since we are going to be doing all the simulation with `int` values, you will have to cast the returned `double` to an `int`.

The other two functions return a `double` value, uniformly distributed, and an `int` value, uniformly distributed.

The sample data you have has a three-minute mean time to vote (30 time steps) with a deviation of 5 (half a minute).

I have provided you with the function to initialize the simulation.

Data Structures

The basic mantra of discrete event simulation is to sort the events into a queue by the arrival time. One can then write a loop:

```
for time t from 0 to some limit
    if there is an event at time t
        process that event
```

This requires iterating over each time step. One can save some time by peeking at the arrival time of the next event on the event queue and updating

the simulating time to that next arrival time, thus not having to walk through the times when nothing could happen.

That is, if current time is 20, and the next event on the queue is going to happen at time 30, there's really no point in looping through 21, 22, 23, ... only to find that nothing is happening. One jumps ahead to 30, when the next event is going to happen.

You don't have to do this. It's perfectly ok to run the full loop and then test to see if anything is scheduled to happen.

Now, as to data structures. I maintain that the best available data structure for keeping track of voters is the STL `multimap` container. We want to sort the events by time, but there could be multiple events happening at the same time. The `map` maps keys to values, but only allows one value for a given key. The `multimap` will keep things in sorted order by time but works like a `map` in that you won't have to do any of your own manipulation of the underlying storage structure. The `multimap` doesn't guarantee anything about the order of items with the same key (e.g., the same voter arrival time), but then again, you don't really care about that in this simulation.

Your Program

Your program, for which the application class will be named `Queue`, has these basic components.

1. `Queue::Init`: This function has been given to you. It generates the random numbers and creates the appropriate number of instances of `OneVoter`, storing each instance in a `multimap` called `voters_backup` with the voter arrival time as the key and the instance of `OneVoter` as the value.

2. `Queue::RunSimulation`: This is the outer loop of the simulation. The input data has a range of the numbers of voting machines. You are to run a different simulation for each possible number of voting machines. With too few machines, the lines to vote will get long. But since machines are expensive resources, the goal would be to allocated sufficient machines to guarantee a reasonable wait time but not so many as to have lots of expensive idle equipment.

   This function runs a loop on numbers of machines and calls an inner simulation function `RunSimulationInner` that does the simulation for a fixed number of machines.

The `RunSimulation`: function will have to set up and tear down the data structures so each call to `RunSimulationInner` starts fresh. That's the reason for the voter queue to be stored in a `voters_backup` data structure.

3. `Queue::RunSimulationInner`: This is the hard part of your code. You will need to run a loop, keeping track of time, until there are no more voters either waiting to vote or actually voting. (That is, you CANNOT run a `for` loop for a fixed range of times. You must run a `while` loop until you are "done".

4. `Queue::DoStatistics`: I have given you a simple function that histograms the wait times for the voters.

5. `Queue::ComputeMeanAndDev`: You will have to write a function to compute the mean and the standard deviation of the wait times.

6. `multimap<int, OneVoter> voters_pending`, `multimap<int, OneVoter> voters_voting`, `multimap<int, OneVoter> voters_done_voting`: These are the three maps that keep track of the progress of transactions (voters) through the system. Voters are initially put into the `voters_pending` map. When a machine becomes available they are taken from this map and put into the `voters_voting` map. When they are done voting they are taken from that map and put into the `voters_done_voting` map.

   (Note: these variables might need to be defined in the header, in which case the actual variable names will be different in order to comply with the style standard.)

7. `OneVoter`: This is the data payload class. You will need, in addition to the simple storage of information about one voter's time in the process, a function to "assign" a voter to a given voting machine and a function to "unassign" a voter when the voter is done voting.

   The sequence number is simply the number assigned at the initialization. This will allow you to keep track of individual voters as they progress through the process.

   Just to be clear:

   (a) `time_arrival`: the time when a voter is scheduled to arrive (uniform distribution over the entire day)

(b) `time_start_voting`: the time when a voter actually is allocated a machine

(c) `time_vote_duration`: the time when a voter actually is allocated a machine (normal distribution with given mean and deviation)

(d) `time_waiting = time_start_voting - time_arrival`

(e) `time_done_voting = time_start_voting + time_vote_duration`

(Note: these variables might need to be defined in the header, in which case the actual variable names will be different in order to comply with the style standard.)