Queue Template, Lab Assignment 7

Re-submit Assignment

Due Oct 18 by 11:59pm **Points** 100 **Submitting** a file upload **File Types** cpp and h

Part 1: Developing And Testing A Queue Template

Write a template, **Queue.h**, to implement a FIFO queue. Here is the specification for its public interface:

```
class Queue
{
  ...
  Queue( ); // may have a defaulted parameter
  void push(const V&);
  V& front( ); // return a mutable reference to the oldest node
  V& back( ); // return a mutable reference to the newest node
  void pop( ); // remove the oldest node
  int size( ) const;
  bool empty( ) const;
  void clear( );
};
```

You may implement your Queue as arrayed or as linked -- your choice. *But no public capacity functions!*

Fully test your template in a test driver CPP named **Queue.TestDriver.cpp**.

Part 2: Server Simulation

Using your queue template from part 1, write **Simulation.cpp** to perform a minute-by-minute simulation based on 6 inputs (to be read from a text file named **simulation.txt**, as numbers *only*, one per line, in this order):

1. the number of servers (1 or more, whole number)
2. the average arrival rate of customers, per minute (greater than 0.0, ***floating point***)
3. the maximum length of the **wait queue** (1 or more, whole number)
4. the minimum service time interval, in minutes (1 or more, whole number)
5. the maximum service time interval, in minutes (>=minimum service time interval, whole number)
6. the clock time at which new arrivals stop, in minutes (>0, whole number)

Echo the above 6 values in your output, well-labeled, as modeled in the sample output below.

Each minute's output should include the following:

- the **clock time** -- that is, the amount of time since the start of the simulation -- in minutes (whole number)
- a visual representation of the **wait queue**

The simulation should pause after each minute's summary is outputted, enabling the user to press ENTER to continue to the next minute. The simulation should end automatically after new arrivals stop arriving *and* the wait queue has been emptied *and* all servers become idle.

Here are the specs:

- Create a struct to represent a **customer** object. Include these data members: (1) an **ID tag** as explained below, (2) **arrival time**, and (3) **service end time**. "Arrival time" is the whole number **clock time** that the customer arrives to be placed in the **wait queue**. **Service end time** is the whole number **clock time** that the customer's service *is scheduled to* end -- it's calculated when their service *begins*, as explained later.
- The **ID tag** for the customer is a single letter of the alphabet, A-Z. Assign A to the first-created customer, B to the next, and so on. After the 26th customer is assigned Z, start the IDs over again -- that is, assign A to the 27th customer. *Use the Q&A section of the module to share ideas of how to manage this.*

- Customers arrive at the specified average arrival rate from the beginning of the simulation until the specified **clock time** at which new arrivals stop. After that time there are no new arrivals, but the simulation continues, allowing the **wait queue** to empty and the servers to become idle.
- Read 6 input values from a text file **simulation.txt** that you will write -- one value per line. *Do NOT submit this file.*
- Use a queue object to represent the **wait queue**. The queue should store customer objects.
- Create the **nowServing array** of customer objects to represent the customers being served. When a customer is removed from the **wait queue**, you'll copy that customer to the **nowServing array**. Include another corresponding array of **boolean values**, whose value is true if the server at that index position is busy serving a customer, false otherwise, indicating that the server is idle. (There's more than one way to accomplish this, so use a different way if you wish). Use your **StaticArray** or your **DynamicArray** -- your choice, and submit its H file with your solution *without modification*.
- As soon as a customer starts being helped by a server, the *service time interval* is determined as a random number in the range from the minimum service time interval to the maximum service time interval. Add the randomly-determined service time interval to the current **clock time** to compute the future **clock time** when service will end. The possible values for service time interval are whole numbers between the minimum service time and maximum service time, inclusive, all equally likely. If the minimum service time and maximum service time are the same, the service time interval is always the same. If the minimum service time is 1 and the maximum service time is 6, the possible service times are 1, 2, 3, 4, 5, and 6 -- all equally likely. HINT -- the last example is like rolling a 6-sided die (ref. **Burns COMSC 110 textbook, ch.8 (http://www.rdb3.com/cpp/exercises/Gaming.supplemental.pdf)**).

Use this algorithm:

```
// read the input values from a text file, one per line, in the order specified above.

// declare and create and assign arrays and queues to be used in the solution

// the clock time loop
for (int time = 0;; time++) // the clock time
{
  // handle all services scheduled to complete at this clock time
  for each server
    if the server is busy
      if the service end time of the customer that it's now serving equals the clock time
        set this server to idle

  // handle new arrivals -- can be turned away if wait queue is at maximum length!
  if clock time is less than the time at which new arrivals stop
    get the #of of arrivals from the "Poisson process" (a whole number >= 0)
    for each new arrival
      if the wait queue is NOT full
        create a new customer object
        set its arrival time equal to the current clock time
        assign it an ID tag (A-Z)
        push the new customer onto the wait queue

  // for idle servers, move customer from wait queue and begin service
  for each server
    if (server is idle AND the wait queue is not empty)
       remove top customer from wait queue
       copy it to the nowServing array at that server's index
       set service end time to current clock time PLUS "random service interval"
       mark that server as busy

  // output the summary
  output the current time
  output a visual prepresentation of the servers and the wait queue
  for each server
```

```
      output the server's index number (zero-based)
      show the ID of the customer being served by that server (blank if idle)
      for server 0 only, show the IDs of customers in the wait queue

    // if the end of the simulation has been reached, break

    // pause for the user to press ENTER
  }
```

NOTE: When you use **srand** in a program, make sure to call it *only once*. The best place to put the call to **srand** is as the first statement in main. Follow **srand** with a call to **rand**, to skip over the first number in the sequence, so it's like this:

```
srand(time(0)); rand( ); // requires cstdlib and ctime
```

NOTE: Use **cout.width(...)** or the manipulator **setw** to format your output table. Do *NOT* use **\t** for spacing.

Sample.

```
number of servers:     4
customer arrival rate: 2.5 per minute, for 50 minutes
maximum queue length:  8
minimum service time:  3 minutes
maximum service time:  10 minutes

Time: 0
---------------------------
server now serving wait queue
------ ----------- ----------
  0         A
  1         B
  2
  3
---------------------------
Press ENTER to continue...

...

Time: 49
---------------------------
server now serving wait queue
------ ----------- ----------
  0         H      KPQVYD
  1         L
  2         Z
  3         F
---------------------------
Press ENTER to continue...

Time: 50
---------------------------
```

```
server now serving wait queue
------ ----------- ----------
  0         H      PQVYDEFG
  1         L
  2         K
  3         F
---------------------------
Press ENTER to continue...

...

Time: 100
---------------------------
server now serving wait queue
------ ----------- ----------
  0
  1
  2         X
  3
---------------------------
Press ENTER to continue...

...

Time: 104
---------------------------
server now serving wait queue
------ ----------- ----------
 0
 1
 2
 3
---------------------------
Done!
```

[The Poisson Function](https://en.wikipedia.org/wiki/Poisson_distribution) (https://en.wikipedia.org/wiki/Poisson_distribution)

Input to this function is the average arrival rate in customers per minute. The output is the actual number of arriving customers for any given minute, randomly generated.

```
// requires cmath and cstdlib
int getRandomNumberOfArrivals(double averageArrivalRate)
{
  int arrivals = 0;
  double probOfnArrivals = exp(-averageArrivalRate);
  for (double randomValue = (double)rand( ) / RAND_MAX;
    (randomValue -= probOfnArrivals) > 0.0;
    probOfnArrivals *= averageArrivalRate / static_cast<double>(++arrivals));
  return arrivals;
}
```

To see how this works, make a test CPP with this loop -- see if the **20** numbers average very close to **2.5**:

```
int main( )
{
  for (int i = 0; i < 20; i++)
    cout << getRandomNumberOfArrivals(2.5) << ' ';
  cout << endl;
}
```

Lab Assignment Rubric

Criteria	Ratings							Pts
Fully accurate results, following all specifications view longer description	Works the first time. 70.0 pts	Works on the 2nd try 65.0 pts	Works on the 3rd try 60.0 pts	Works after 4 or more tries. 50.0 pts	Doesn't work after 2 weeks. Partial credit. 20.0 pts	Not submitted within two weeks of the due date. 0.0 pts	Work is not original -- appears to be a marked-up copy of the work of another or previous student. 0.0 pts	70.0 pts
Submits all work on time, fully complete if not fully correct. view longer description	Submitted on time 20.0 pts	Submitted on time, but one or more files are missing or not correctly named. 16.0 pts	Submitted on time, but with missing identification in one or more submitted CPP or H files. 15.0 pts	Submitted on time but not fully complete. 10.0 pts	Late or wholly incomplete! 0.0 pts			20.0 pts
Well-organized and professional quality code. view longer description	Fully meets expectations 10.0 pts	Mostly meets expectations, just needs to be a bit more careful. 8.0 pts	Many areas are well done, but there are a lot of areas that need work. 6.0 pts	Getting there, but needs to be a lot better. 3.0 pts	Needs a lot of work. See the instructor for guidance. 0.0 pts			10.0 pts
								Total Points: 100.0