CISC 3130
Project 2

Simulated Shopping

Introduction

The online bookstore has failed. The basement of Boylan is being gutted to make room for ℬℂ**barter**, a store selling artisanal textbooks; quirky, one-of-a-kind office and school supplies, fair trade hoodies, and a wide range of organic junk food. Like many retail stores, they are interested in optimizing the number of points of sale in the store: customers should spend as little time waiting as possible, but their cashiers also should not spend too much time idle.

The store management has some ideas about how the points of sale should be organized, and they also have data from other franchises, along with an analysis of local demographics, that gives them some starting expectations about how their customers will behave. Rather than experiment with different configurations in the store itself (which will take a lot of time and extra energy), they've asked us to run some simulations to help them decide how to design the store initially.

Background

What determines how long you wait at a cash register? Mainly, it's the number of people in line ahead of you, as well as the number of items they're purchasing, the speed of the cashier, and the time required to complete payment. What determines how many people are in line ahead of you? Fundamentally, it's a question of whether people are getting in line faster than they can be rung up—if it takes, say, 2 minutes to ring someone up, but someone gets in line every 1 minute, after a few minutes there's going to be a bit of a line.

Of course, we can't predict exactly when each customer is going to enter the store, or how many things they're going to want to buy, and so on. We may know something about the *average* behavior of customers, though. So in our simulation, we're going use a combination of simplifying assumptions and random numbers.

In particular, we're to make the following assumptions (based on data that ℬℂ**barter** has collected):

- Each item being purchased requires 9 seconds of handling (to take it out of the basket, scan it, and bag it). That's an average time, of course, but there's not a lot of variance, so we'll just use 9 seconds. Similarly, we'll assume it always takes 60 seconds to process payment.
- On average, each shopper buys 6 items. But obviously there's going to be a *lot* of variance in that—a lot of people will be getting 4-8 items, a few 1 (or even 0—realizing in line that they forgot their wallet at home), and a few 15 or more. Probabilistically, the number of items follows a *Poisson distribution* with mean 6. (Here is a graph of the Poisson distribution with mean 5.) You don't need to worry about the technical details, but later we'll be using methods that produce random numbers distributed in this way.
- At busy times, a shopper gets in line on average every 30 seconds. Put another way, the odds that someone will arrive during any particular second are 1/30. Again, there's going to be wide variance; the time between shoppers follows an *exponential distribution* with rate 1/30. (Here is a graph of a few exponential distributions with different rates—note that as the rate increases, shorter inter-arrival times become more likely.) During slow times, shoppers get in

line every 120 seconds on average.

Based on this initial data, the store management plans to set up three cash registers, though they're not sure how to designate them or staff them. They have 4 basic ideas in mind that they want to run simulations on:

1. The "express" model. One cash register will be reserved for customers with 5 items or less. During busy times, they will add either 1 or 2 registers for people with more than 5 items. During slow times, they will either just have one register open for everyone, or one express register and one "regular" register. (We'll assume that no-one lines up in the wrong line, and when they have a choice, they'll pick the shorter of two lines.) We should simulate all 4 variants of the express model.
2. The "loyalty" model. One cash register will be reserved for customers who have joined the ℬℂ**barter** loyalty program. As with the "express" model, during busy times, they'll add 1 or 2 registers for non-loyalty customers; during slow times they might have one general register or one of each.
3. The "superpriority" model. One line forms, and when a register becomes available, the person in line with the fewest items goes next. If there's a tie, customers in the loyalty program have preference; if there's still a tie, the customer closest to the front of the line goes. During slow times, there will be 1 or 2 registers; during busy times, there will be 2 or 3.
4. The "uniform" model. All registers are open to everyone; customers will pick the one with the shortest line. Slow times will have 1 or 2 registers; busy times 2 or 3.

Implementation

You will need to write several classes to implement this set of simulations!

The objects that actually arrive in line are `Carts`. A `Cart` has a number of items, an arrival time (when the cart enters a line), and a departure time (when they leave the store after paying). If the `Cart` belongs to a loyal customer, it also has a `Customer` field. The `Cart` class will also need methods to calculate/report certain values needed for the simulation. For the purposes of our study, we will assume that half of the visitors to the store are loyal customers. (Note: in general, we expect some loyal customers to visit the store multiple times during each simulation.)

A `Customer` is similar to a `Shopper` from Project 0. Each `Customer` object has a loyalty number (a value between 0 and 9999), a count of many items the customer has purchased at the store, a count of how many times the customer has been to a cash register, and the total time the customer has spent waiting in line. Like `Cart`, `Customer` will need methods to calculate/report certain values needed for the simulation.

A cash register will be represented by a `Register` object. Each `Register` object will include a reference to the line that "feeds" it, and will track the number of `Carts` it has processed, and the minimum, maximum, and average wait times (i.e. the time between entering the line and beginning the checkout process. The `Register` class will have a `processNextCart()` method (which will cause it to take the next `Cart` from its line) and a `getNextAvailableTime()` method, which will return the time it will be available to process the next cart.

A line in the store will be represented by a `Line` object. This is basically a queue of `Cart` objects. Most `Lines` can use the `queue` class from the STL, but the "superpriority" `Line` will need to use `priority_queue`.

For customers in the loyalty program, we can track their experience over several simulations (this is part of the value of having a loyalty program!) The `CustomerHistory` class should allow you to record and access information about your loyal customers. In particular, use the `unordered_map` class template to track your loyal customers. The key values are the loyalty program membership number; the mapped values are `Customer` objects.

Finally, a `Store` object represents one configuration of a store (a collection of Lines, as well as the way those lines are used to serve customers). These objects will essentially manage the simulations. The design of the Store class is up to you—you'll need to be able to start a simulation, collect the data from a simulation, and set necessary parameters for the simulation.

What To Do

For each variant of the four store models (but see note about extra credit below), you will run 4-hour simulations of slow and busy periods. That is, run a simulation of a 4-hour slow period, and a simulation of a 4-hour busy period. Do this 1000 times for each kind of period—each simulation will start with empty lines and will be "seeded" with a different random number. For each variant, and for the slow/busy times your program will report:

1. Average customer wait time
2. Maximum customer wait time
3. Maximum line length
4. Total idle time (no customers waiting or paying) for all lines

In addition, for your loyal customers, report the average and maximum wait time per item purchased. (That is, calculate each customer's wait time per item purchased, and report the average and maximum across all customers.) This report should include *both* the busy and slow simulations for the store configuration.

How To Do It

There are several things you need to do this assignment. First, you need to get (pseudo)random values with the appropriate characteristics. Also, you must be able to run several simulations with *different* sequences of random values, which means you need to *seed* your random number generator with different values. To do that, we'll use the current time.

To get numbers from a Poisson distribution with (say) mean 6, declare:

```
unsigned seed =
      std::chrono::system_clock::now().time_since_epoch().count();
std::default_random_engine generator(seed);
std::poisson_distribution<int> distribution(6);
```

and then call `distribution(generator)` to get a random `int` value.

Similarly for an exponential distribution with (say) rate 1/30,

```
unsigned seed =
      std::chrono::system_clock::now().time_since_epoch().count();
std::default_random_engine generator (seed);
```

```
std::exponential_distribution<double> distribution (1/30.0);
```

Note that you can "create" your customers *before* the simulation starts, which will make managing the simulation pretty easy.

Another challenge is keeping track of time during a simulation. How to do this? Obviously you don't want to do "in real time"—that would take quite a while. You might be tempted to have a "second counter" that you increment, then see if anything happens (a customer arrives, or departs), then increment, etc. But that isn't very efficient—*most* seconds are not going to have any activity. So instead, start your simulation clock at 0, then figure out what the next thing to happen will be—at the beginning, the next thing will be a customer getting in line; later on, it could also be a customer completing payment. So advance your clock to that time (say, 37 seconds), process that activity, then identify what's going to happen next. That way, your "clock" will only take on "interesting" values. (Note that you'll need a mechanism for transforming (for example) the fact that customer *n* arrives 43 seconds after customer *n-1* into the fact that customer n arrives at second 853.)

What to Turn In

Email me a zipfile named familyname_givenname.zip. (So I would send `dexter_scott.zip`). You may also send a `.tgz` file if you're using Unix/Linux. Do not send a RAR or other kind of compressed file; those will get a grade of 0 points. Make sure the subject of your email is *CISC 3130 Project 2*. The zipfile should contain:

- Your makefile, which produces an executable file called `projec2`
- All `.h` and `.cpp` files in your project.
- The output from a run of your program, named familyname_givenname.out.

I should be able to extract files from your zip, run `make`, and run `project2`. If any part of that fails, you will not earn full points.

On Style

As usual, your code should follow best-practice style guidelines—probably Google's C++ Style Guide is the most comprehensive. Pay particular attention to the sections on Naming, Comments, and Formatting. If I have to work hard to figure out what you're trying to do, you will not earn full points.

On Full and Extra Credit

For *full credit*, run the simulations for all variants of store model 4 ("uniform). For *some extra credit*, also simulate models 1 and 2. (You may need to update your object-oriented design to accommodate different kinds of store configurations.) For *more extra credit*, also simulate model 3. Note that this will involve function pointers/objects along the lines of project 1.

In your project-submission email, indicate whether you are submitting for full credit, some extra, or more extra. If you don't indicate you are submitting for extra credit, you will not receive any, regardless of the status of your code.