Homework 8

CS 173: Intermediate Computer Science — Spring 2014
Instructor: Thomas Bressoud — Part I Due: 2014-04-8
Part II Due: 2014-04-14

- In this assignment we will build on our discrete event simulation foundation from HW7.
- We will use C++ inheritance to derive both a `CustomerArrival` class and a `Server` class from our `Event` class.
- This project will again be done in two-member teams, randomly generated. You are allowed to use either team members' HW7 submissions for the basis of the classes repeated in this assignment.
- The earlier guidelines for documenting and testing your code hold.

Part I: Modifications to existing classes: The Event class, the OrderedSet class, and the two unit tests for those classes.
Part II: The new classes: CustomerArrival, Server, Simulator, and a main driver.

For Part II, you are expected to thoroughly test your implementation, but I will not impose the use of GoogleTest for the new classes (although you are welcome to use its infrastructure if you desire). However, I do expect an essay (of at least a page) describing how you tested and why the testing you chose both exercised as many paths of execution as possible, and also validated the correct operation of the simulation.

The `Event` Class

We need to make a minor modification to the `Event` class to prepare it for use as a base class and having inherited derived classes. For each member function that we foresee being overridden by a derived class, we annotate the function prototype in the class definition with the `virtual` keyword. So we have:

```
virtual string str() const;
virtual void execute();
virtual ~Event();
```

Note that, for inheritance, we should have a destructor for the base class, even though, for `Event`s, that method will be empty, since there is no dynamic allocation in the class. For access by derived classes, the data members should be changed to `protected`. Note also that this virtual keyword only occurs in the `.h`, not in the `.cpp`. Once you re-make, you should be able to successfully execute `testEvent` without modification.

The `OrderedSet` Class

Our first real order of business is, now that we better understand pointers, to make our OrderedSet contain an array of *pointers* to Events, instead of copies of Event objects. The motivation for this is that we need to avoid the copies of Event objects for the two classes that will derive from `Event`, namely `CustomerArrival` and `Server`. To do this, in the header file for `OrderedSet`, we define:

```
typedef Event * EPointer;
```

This, then, becomes the data type for the parameter to `insert` and for the return type from `removeFirst`:

```
int insert(EPointer e);
EPointer removeFirst();
```

The other member function prototypes in the class definition should not change.
If our `private` data member for the array storage were named `A`, then a pointer for a dynamically allocated array of `EPointer`s would be:

```
EPointer * A;
```

and would be allocated in a constructor by:

```
A = new EPointer[maxsize];
```

The rest of the changes in `OrderedSet` involve reflecting the decision of have an array of pointers, and passing and returning pointers to Events throughout the existing code. The compiler will help a lot here, detecting places where the types are no longer correct. As an example of the kind of change required, anytime you used to have something of the form `A[i].id()`, you would now have `A[i]->id()`. However, manipulating array elements and copying what are now pointers instead of Events does not require change. For example, `A[i+1] = A[i]` or `A[i] = e` (where `e` is of data type `EPointer`).

The `CustomerArrival` Class

The `CustomerArrival` class derives from Event and has the following interface:

```
#include <string>
#include <random>
#include "Event.h"
#include "Queue.h"
#include "Server.h"
#include "Simulator.h"

class CustomerArrival: public Event {
public:
```

```
  CustomerArrival(double mean = 10.0, Queue * queue = NULL,
      Server * server = NULL, Simulator * sim = NULL,
      int count = 10, double time=0.0);
  std::string str() const;
  void execute();
  virtual ~CustomerArrival();
private:
  int num_;          // Count of customers generated
  double mean_;      // Customer inter arrival time
  Queue * Q;         // Access to the shared queue
  Server * S;        // Access to the server
  Simulator * sim_;  // Access to the simulator
  int count_;        // When to stop generating customers

  std::default_random_engine gen;       // Generator of random numbers
  std::exponential_distribution<> exp;  // Object for getting values
                                        // from a particular distribution
};
```

Constructor: The constructor has parameters for all the other objects that the CustomerArrival Event must interact with. It must be able to place customers on the Queue shared with the server. If no customers are waiting on the queue, it must be able to invoke the server to begin servicing a customer, and it needs the Simulator itself to add the CustomerArrival Event into the simulator's OrderedSet. For each of these interacting objects, the constructor is given a pointer to the corresponding object, which it will store in private data members for the use of its member functions. The constructor is also given a count of how many total times to generate a Customer as well as the time for the first arrival.

`execute()` Customers "arrive" by being created each time this Event runs its `execute` method. A newly arrived customer should generate its own string label, based on the data member keeping track of how many customers have been generated. The arrival time for the newly created customer should be the current time of the simulation. The execute() method should check the status of the server and, if available, initiate service on the generated customer. Otherwise, the customer should be placed in the shared queue.

The CustomerArrival Event should then determine it the next customer arrival time, and insert itself back into the simulator to execute at that time. The next arrival time should be the current time plus an inter arrival time given by an exponential distribution with mean `mean_`.

`str()`: Yield a string of the form "<Arrival *id*: *time*>".

Desctructor: The Destructor may be empty, unless some dynamic allocation is occurring.

The `Server` Class

The `Server` class also derives from Event and has the following interface:

```
#include <random>
#include "Event.h"
#include "Queue.h"
#include "Simulator.h"

class Server: public Event {
public:
  Server(double mean = 7.0, Queue * queue = NULL, Simulator * sim = NULL);
  bool available();
  void startService(Customer & c);
  void execute();
  virtual ~Server();
private:
  double mean_;            // Service time mean
  Queue * Q;               // Access to shared queue
  Simulator * sim_;        // Access to "parent" simulator
  Customer current;        // Copy of current customer
  bool busy;               // Boolean maintaining service-in-progress
  int count;               // Number of customers served
  double lastStart;        // Time of last start-of-service
  double totalServiceTime; // Aggregate of all completed service time

  std::default_random_engine gen;       // Generator of random numbers
  std::exponential_distribution<> exp;  // Object for getting values
                                        // from a particular distribution
};
```

Constructor: The constructor is given pointers to the queue and the simulator along with the mean for the service time. It should use the mean to generate the exponential distribution maintained by the object.

`available()`: This method simply returns a boolean indicating whether or not the Server object is busy servicing a customer.

`startService()`: This method is invoked either because of a customer arrival, or, internally, based on completion of some previous servicing of a customer. It is responsible for setting variables used in the gathering of statistics of the simulation (for instance, this point in time marks the end of the waiting time for a customer, and begins the service time for that customer), as well is internally recording the customer being serviced and that the server is now busy. Finally, it should

determine the customer service completion time, and insert itself back into the simulator to `execute()` at that time. The next event time should be the current time plus an service time given by an exponential distribution with mean `mean_`.

`execute()`: This method is invoked (by the Simulator `doAllEvents()`) to mark the completion of service for the "current" customer. On service completion, the Server should record/update any statistics for the simulation, and should mark itself as now available. If the queue has any waiting customers, it should dequeue the first customer and invoke its own `startService()` method.

Destructor: This may be empty unless deallocation of dynamic resources is necessary.

The `Simulator` Class

The `Simulator` class controls the overall simulation and is a close cousin to the simulators we defined in Python. Its interface is given below:

```
#include "OrderedSet.h"
#include "Event.h"

class Simulator {
public:
  Simulator();           // Constructor
  virtual ~Simulator();  // Destructor
  double now() const;    // Return value of current virtual time
  int insert(Event * e); // Given a pointer to an event, insert in Future events list
  void doAllEvents();    // Main engine to drive simulation

  // Setup will change from simulation to simulation.  The following is specific
  // to the MM1 Queue simulation.

  virtual void setup(int custCount, double arrivalMean = 10.0,
                                    double serviceMean = 5.0);

private:
  double vtime_;        // Current virtual time of the simulation
  OrderedSet events;    // Future events list in time order
};
```

The Simulator class above is specific to the M/M/1 Queueing simulation of this assignment. One could imagine a more general design, in which the Simulator had a virtual, and argument-free setup and this Simulator class was then used as a base class. In this alternate design, different classes derived from Simulator would setup for the particular system being modeled.

Constructor: The constructor should initialize the data members for the object, but will defer to the `setup` method for the creation of the top level objects of the simulated system.

`setup()`: This method takes an integer specifying the total count of customers to introduce into the system, and the simulation parameters of the mean inter arrival time and the mean service time. The method should create the shared queue, the server, and the customer arrival objects with appropriate arguments. Finally, it should insert the customer arrival into to simulator to "prime" the future events OrderedSet of the simulation.

`now()`: Return the current virtual time of the simulation.

`insert()`: Takes a pointer to an event and inserts it into the OrderedSet maintained by the simulation, returning the OrderedSet's return value indicating success or failure of the insertion.

`doAllEvents()`: Main driver of the simulation object, wherein it repeatedly removes the first event from the OrderedSet, updates virtual time to the time of the event, and executes the event. This continues as long as there are events to process.

Statistics and Testing

The point of a simulation is to gather information during the execution of a system and to report that information out at the end of a simulation run. The experimenter then runs the simulation over a set of input parameters (in our case, varying the arrival mean and the service mean, and perhaps the number of customers to simulate). The sets of outputs then give the experimenter a quantitative view of the behavior of the system.

The more creative part of this assignment is for your team to figure out how to add data members to the various objects to keep track of the statistics needed for the simulation. I am not going to mandate a set of outputs, but will list here some of the information often reported for systems of this type. In designing what information to keep where, and how to report the information, you can add data members and new methods to any of the top level objects, but you should not *change* any of the methods I have already specified.

Part of the grade for this project will be based on how much information you are able to report and on how well you design and implement these facilities.

As noted at the beginning of this writeup, you are also responsible for deciding how to test your simulator. This can tie in with the statistics you gather during the simulation, as there are closed form solutions for some of the simulation outputs given below, based on just the arrival mean and the service mean of an M/M/1 queueing system. This can allow you to validate your results against theoretical results from these closed forms.

Possible Information:

- Total Arrivals
- Mean time spent waiting in queue
- Mean time spent in system
- Server utilization
- Max/Mean length of queue
- Mean number of customers in system

Submission

All source files along with a Makefile should be submitted through submitbox. The Makefile should be operational when placed with the given set of source files in a directory of my choosing. So make sure there are no dependencies on the relative location of other files outside the current directory, and the only absolute location may be the directory root for Google Test source and includes.
You should also submit your essay on test methodology and describe your results. If your program requires anything more than simply executing:

```
$ ./testSimulator
```

then you must submit explicit instructions for me to follow to execute your program and put it through its paces.