EECS402, Winter 2016, Project 4

Overview:

For this project, you will develop an event-driven simulation of an attraction at a theme park such as Disneyland. At some theme parks, you can pay an extra fee and get priority access to different attractions. We could use this simulation to determine the best way to set different aspects of our park, such as:

1. how much to charge for the priority access (charging more means less people will buy the priority access);
2. how often the attraction should accept new riders (more "runs" of the ride can result in higher operating costs and increased likelihood of breakdown);
3. how many non-priority riders should be allowed on a run even when priority members are waiting (too many means priority riders get upset and too few means non-priority riders may never get to ride);
4. etc…

Before implementing the simulation, though, you will need to develop a few data structures to support your simulation. Rather than write the data structures to only support this project, though, you will develop them in a template way so they can be used for other purposes as well.

As promised earlier in the semester, you will notice that the simulation aspect of this project is not as fully specified as some previous projects. This is completely intentional, and is intended to get you thinking about your own design, your own outputs, etc. Please take some time to make sure your program solves the problems posed in the specifications and acts as described, while providing quality and informative outputs so that an end-user could use your simulation to study effects of different changes to our attraction and/or priority pricing schemes.

Due Date and Submitting:

This project is due on **Friday, April 8 at 4:30pm**. Early and late submissions are allowed, with corresponding bonus points or penalties, according to the policy described in detail in the course syllabus.

For this project, your solution must be organized into multiple files, where each class definition is in its own header file (extension ".h"), and each class' member functions are defined in a "inline source" file (extension ".inl") for each templated class and a "regular source" file (extension ".cpp" for each non-templated class. Note: As discussed in lecture, do NOT put your template class function implementations in a ".cpp" file – instead, put them in a ".inl" file and remember to #include the corresponding ".inl" file at the bottom of each ".h" file.

Phase 1:

This phase will be completely specified – please be sure to follow these specs exactly, do not deviate from them in any way, including, but not limited to: changing the class name, method names, attribute names, etc; adding functionality; neglecting functionality; changing the purpose or functionality of the methods, etc…

In this phase, you will develop a doubly-linked list that is always maintained in a sorted order, along with a simple node class that will be able to be used by the sorted list. Keep in mind that, while the list will fully utilize the node class, the node class really shouldn't know anything about a list – it is just a node, and it might be used in other data structures (like the one in phase 2!).

LinkedNodeClass:

This templated class will be used to store individual nodes of a doubly-linked data structure. This class should end up being quite short and simple – no significant complexity is needed, desired, or allowed. The interface to the LinkedNodeClass will be ***exactly*** as follows:

```
//The list node class will be the data type for individual nodes of
//a doubly-linked data structure.
template < class T >
class LinkedNodeClass
{
  private:
    LinkedNodeClass *prevNode; //Will point to the node that comes before
                               //this node in the data structure. Will be
                               //NULL if this is the first node.
    T nodeVal;                 //The value contained within this node.
    LinkedNodeClass *nextNode; //Will point to the node that comes after
                               //this node in the data structure. Will be
                               //NULL if this is the last node.
  public:
    //The ONLY constructor for the linked node class - it takes in the
    //newly created node's previous pointer, value, and next pointer,
    //and assigns them.
    LinkedNodeClass(
         LinkedNodeClass *inPrev, //Address of node that comes before this one
         const T &inVal,          //Value to be contained in this node
         LinkedNodeClass *inNext  //Address of node that comes after this one
         );

    //Returns the value stored within this node.
    T getValue(
         ) const;

    //Returns the address of the node that follows this node.
    LinkedNodeClass* getNext(
         ) const;

    //Returns the address of the node that comes before this node.
    LinkedNodeClass* getPrev(
         ) const;

    //Sets the object’s next node pointer to NULL.
    void setNextPointerToNull(
         );

    //Sets the object's previous node pointer to NULL.
    void setPreviousPointerToNull(
         );

    //This function DOES NOT modify "this" node. Instead, it uses
    //the pointers contained within this node to change the previous
    //and next nodes so that they point to this node appropriately.
    //In other words, if "this" node is set up such that its prevNode
    //pointer points to a node (call it "A"), and "this" node's
    //nextNode pointer points to a node (call it "B"), then calling
    //setBeforeAndAfterPointers results in the node we're calling
    //"A" to be updated so its "nextNode" points to "this" node, and
    //the node we're calling "B" is updates so its "prevNode" points
    //to "this" node, but "this" node itself remains unchanged.
    void setBeforeAndAfterPointers(
```

```
        );
};
```

SortedListClass:

This templated class will be used to store a doubly-linked list in a sorted way, such that the user does not specify where in the list a value should be inserted, but rather the new value is inserted in the correct place to maintain a sorted order. The interface to the SortedListClass will be ***exactly*** as follows:

```
//The sorted list class does not store any data directly.  Instead,
//it contains a collection of LinkedNodeClass objects, each of which
//contains one element.
template < class T >
class SortedListClass
{
  private:
    LinkedNodeClass< T > *head; //Points to the first node in a list, or NULL
                                //if list is empty.
    LinkedNodeClass< T > *tail; //Points to the last node in a list, or NULL
                                //if list is empty.

  public:
    //Default Constructor.  Will properly initialize a list to
    //be an empty list, to which values can be added.
    SortedListClass(
         );

    //Copy constructor.  Will make a complete copy of the list, such
    //that one can be changed without affecting the other.
    SortedListClass(
         const SortedListClass< T > &rhs
         );

    //Clears the list to an empty state without resulting in any
    //memory leaks.
    void clear(
         );

    //Allows the user to insert a value into the list.  Since this
    //is a sorted list, there is no need to specify where in the list
    //to insert the element.  It will insert it in the appropriate
    //location based on the value being inserted.  If the node value
    //being inserted is found to be "equal to" one or more node values
    //already in the list, the newly inserted node will be placed AFTER
    //the previously inserted nodes.
    void insertValue(
         const T &valToInsert  //The value to insert into the list
         );

    //Prints the contents of the list from head to tail to the screen.
    //Begins with a line reading "Forward List Contents Follow:", then
    //prints one list element per line, indented two spaces, then prints
    //the line "End Of List Contents" to indicate the end of the list.
```

```
    void printForward(
         ) const;

    //Prints the contents of the list from tail to head to the screen.
    //Begins with a line reading "Backward List Contents Follow:", then
    //prints one list element per line, indented two spaces, then prints
    //the line "End Of List Contents" to indicate the end of the list.
    void printBackward(
         ) const;

    //Removes the front item from the list and returns the value that
    //was contained in it via the reference parameter.  If the list
    //was empty, the function returns false to indicate failure, and
    //the contents of the reference parameter upon return is undefined.
    //If the list was not empty and the first item was successfully
    //removed, true is returned, and the reference parameter will
    //be set to the item that was removed.
    bool removeFront(
         T &theVal
         );

    //Removes the last item from the list and returns the value that
    //was contained in it via the reference parameter.  If the list
    //was empty, the function returns false to indicate failure, and
    //the contents of the reference parameter upon return is undefined.
    //If the list was not empty and the last item was successfully
    //removed, true is returned, and the reference parameter will
    //be set to the item that was removed.
    bool removeLast(
         T &theVal
         );

    //Returns the number of nodes contained in the list.
    int getNumElems(
         ) const;

    //Provides the value stored in the node at index provided in the
    //"index" parameter.  If the index is out of range, then outVal
    //remains unchanged and false is returned.  Otherwise, the function
    //returns true, and the reference parameter outVal will contain
    //a copy of the value at that location.
    bool getElemAtIndex(
         const int index,
         T &outVal
         );
};
```

Phase 2:

After you've completed phase 1, move on to phase 2, which should be able to be completed very quickly so you can move on to the more complex phase 3.

FIFOQueueClass:

One additional data structure will also be required – a "first-in-first-out queue" data structure will be used to maintain a line of riders that are waiting to ride the attraction. You should be able to use the LinkedNodeClass you developed in phase 1 to very quickly develop and test this data structure. If written correctly, this class should be *very short* and simple, and should not use a significant chunk of the time required to implement this project. Since the FIFO queue has such restricted functionality, it is quite straight forward to develop, especially since the bidirectional LinkedNodeClass is already available and can be used to make this a very short and simple data structure to implement.

This templated class will be used to store a simple first-in-first-out queue data structure. It's full and complete specification is as follows:

```
template < class T >
class FIFOQueueClass
{
  private:
    LinkedNodeClass< T > *head; //Points to the first node in a queue, or NULL
                                //if queue is empty.
    LinkedNodeClass< T > *tail; //Points to the last node in a queue, or NULL
                                //if queue is empty.

  public:
    //Default Constructor.  Will properly initialize a queue to
    //be an empty queue, to which values can be added.
    FIFOQueueClass(
         );

    //Inserts the value provided (newItem) into the queue.
    void enqueue(
         const T &newItem
         );

    //Attempts to take the next item out of the queue.  If the
    //queue is empty, the function returns false and the state
    //of the reference parameter (outItem) is undefined.  If the
    //queue is not empty, the function returns true and outItem
    //becomes a copy of the next item in the queue, which is
    //removed from the data structure.
    bool dequeue(
         T &outItem
         );

    //Prints out the contents of the queue.  All printing is done
    //on one line, using a single space to separate values, and a
    //single newline character is printed at the end.
    void print(
         ) const;
};
```

Phase 3:

This is the phase where you will develop your actual simulation. Unlike the earlier phases, this phase is not nearly as specified, giving you the freedom to design your own solution.

Simulation Specification:

The simulation must be an event-driven simulation. Rider arrivals at the attraction are to be pseudo-randomly determined using a given distribution. Your event list cannot have more than one rider arrival at any time – in other words: do not pre-populate all rider arrivals at the beginning of your simulation – instead, generate the "next" rider arrival event each time a rider arrives at the attraction.

The attraction you are simulating will have a name and a pre-set number of seats available in the train of cars. For this project, you should set your attraction's name to "Space Mountain" and set the number of seats available to 20. Note: these should be maintained as attributes so they could be easily initialized for different attractions at the park.

Our theme park currently has three levels of priority: "Standard", "Fast Pass", and "Super Fast Pass". We've discussed the possibility of additional priority levels ("Ultra Fast Pass", "Platinum", etc.) but we have not put those in place yet. It's important to realize that we may want to introduce additional priority levels in the future, and your simulation should allow this change to be relatively straightforward.

Attractions at our park have a separate line that riders must stand in for each priority. For example, all "Standard" riders (STD) wait in one line, while "Fast Pass" riders (FP) wait in a second line, and "Super Fast Pass" riders (SFP) wait in a third line. Attractions are operated by a train of cars, each of which holds some number of riders. When this train of cars arrives at the attraction's station, some number of riders from each priority level are allowed on to enjoy the ride. To make the extra cost worth it for the priority riders, more riders from the SFP line will be admitted than riders from the FP line (and even fewer from STD). For example, if there are 10 seats available in the cars, we might allow 6 SFP riders, 3 FP riders and 1 STD rider. For this example, we have:

idealNumSFP = 6, idealNumFP=3, idealNumSTD = 1

When there are lots of riders in each line, this is straightforward, but occasionally there are times when some of the lines are short or even empty. At these times, we still want to fill up the cars with riders as much as possible, so additional logic is needed. To keep things as simple as possible, our park's policy will be as follows: If the ideal number of each priority level rider is not able to fill up the car, we take as many SFP riders as available to fill up the car. If there aren't enough SFP riders to fill it, we take as many FP riders are needed. Finally, if there are not enough SFP or FP riders, the car is filled up with STD riders.

To make this simple example sound even more complicated, I'll provide an example. Say we have the ideal number of riders as described above. Currently, though, there are 3 SFP riders in line, 20 FP riders, and 12 STD riders. In this case, we would admit the 3 SFP riders, then the first 3 FP riders (the ideal number of FP riders), then 1 STD rider (the ideal number of STD riders). At this point, our car has 7 riders, but 10 seats, so 3 seats are empty. If there were SFP riders waiting, we would admit them, but since there are none in this example, we admit the next 3 FP riders to fill up the car.

The ride takes some amount of time, during which, additional riders may arrive and get in the appropriate line. When a train of cars arrives at the station, the next set of riders is admitted and the process continues until the park closes. Because we are nice theme park-owners, we don't want to kick people out as soon as the park closes – instead, we will stop allowing new rider arrivals, but those in line will get a chance to ride before the ride shuts down for the evening, meaning that the ride could continue accepting waiting riders well after the official closing time.

Input:

Your simulation will be controlled by a set of parameters, which you must read in from a text file. These parameters include:

- Park closing time: No additional rider arrivals will be allowed after this time
- Rider arrival rate normal distribution mean: Riders have been determined to arrive at the attraction in a normally distributed way. This value is the mean of that normal distribution.
- Rider arrival rate normal distribution standard deviation: See the previous description
- Car arrival rate uniform distribution minimum value: Car arrivals at the attraction station are not perfectly timed. Sometimes there are delays for the amount of time it takes riders to get off the car at the end of the ride, there are minor mechanical problems, etc. We have determined that cars arrive at the station to pick up riders in a uniformly distributed way, and this is the minimum value of that distribution.
- Car arrival rate uniform distribution maximum value: See the previous description
- What percentage of the riders have purchased super fast pass access
- What percentage of the riders have purchased fast pass access
- The ideal number of riders from the super fast pass access line that are admitted for a run of the ride
- The ideal number of riders from the fast pass access line that are admitted for a run of the ride

Riders should randomly be assigned priority level using a uniform distribution and the specified percentages. For example, when a rider is instantiated to be introduced into the simulation, draw a uniform random number and convert to one of three possible priorities according to the specified percentages.

These values will be specified in a text file, whose name is input to the simulation via the command line at execution time. For example you will execute your program via the following command line:

```
./parkSimulation simParams.txt
```

Where "./parkSimulation" is your compiled/linked executable, and "simParams.txt" is the name of the text file containing the simulation parameters for this run. Do not assume the name "simParams.txt" as this will change from one run of your program to the next.

The text file of parameters is being kept simple for this project and will assume/expect parameters to be specified in a very set order as follows:

```
<closing time>
<rider arrival mean>
<rider arrival stddev>
<car arrival min>
<car arrival max>
<percentage of riders that are super fast pass>
<percentage of riders that are fast pass>
<number of super fast pass riders admitted>
<number of fast pass riders admitted>
```

Where each item shown in <> is replaced with a single value. For example, a parameter file that looks like this:

```
1000
12
2.5
18
24
20
40
6
3
```

Would mean the park closes at time 1000, riders arrive at the attraction via a normal distribution with a mean of 12 and a standard deviation of 2.5, and cars arrive at the station via a uniform distribution between 18 and 24. It also specifies that 20% of the riders have super fast pass access, and 40% of the riders have fast pass access (it is implied, then that the remaining 40% are "standard" access riders), and for each run of the ride, 6 riders from the super fast pass line will be admitted, 3 from the fast pass line, and the remaining number from the standard line.

Input Error Checking:

In order to keep the complexity of phase 3 a little bit lower, you are not required to perform extensive error checking on the input file – when we test your program, we will use files having the format described here. This is generally a terribly assumption to make, but I want you to focus on the data structures and simulation for this project, as opposed to worrying about file format errors and such.

Output:

Your simulation must produce output that allows a user to see a step-by-step run of the simulation. That is, each time an event is handled, produce output that shows the entire event list, the number of riders of each priority level waiting to ride, etc. Make it clear what the simulation is doing – don't just say "filling up car!", instead, indicate how many of each priority level rider is being admitted to the car, etc. While it will be somewhat subjective, part of your grade will be based on whether we can easily determine exactly what your simulation is doing, and more importantly, that it is doing the *right* thing. If we can't tell if the simulation is acting properly or not, we will *assume it is not* and you will be graded accordingly! I will not provide any sample outputs and am leaving it up to you to make it clear.

Finally, when the simulation is complete, you must output some overall simulation statistics. Things like "what was the longest SFP line" and "what was the average amount of time a FP rider had to wait" etc. are excellent statistics. Think about the purpose of this simulation and what kind of conclusions we might want to come up with as a result of running your simulation, and provide a suite of statistics that could help make those decisions.

Randomness

The code to generate pseudo-random values will be posted on the ctools site along with these specs.

"Specific Specifications":

These "specific specifications" are meant to state whether or not something is allowed. A "no" means you definitely may NOT use that item. In general, you can assume that you should not be using anything that has not yet been covered in lecture (as of the first posting of the project).

- Use of Goto: No
- Global Variables / Objects: No
- Global Functions: Yes
- Use of Friend Functions / Classes: No
- Use of Structs: No
- Use of Classes: Yes
- Public Data In Classes: **No** (all data members must be private)
- Use of Inheritance / Polymorphism: No
- Use of Arrays: Yes
- Use of C++ "string" Type: Yes
- Use of C-Strings: No
- Use of Pointers: Yes
- Use of STL Containers: **No** (use the data structures from phases 1 and 2!)
- Use of Makefile / User-Defined Header Files / Multiple Source Code Files: Yes – required!
- Use of exit(): No
- Use of overloaded operators: Yes
- Use of float type: **No** (That is, all floating point values should be type double, not float)