CPSC 221, January-April 2017
Programming Project 1:
Queues, Simulation of an Airport Runway

History of Non-Trivial Changes:

- Jan. 19 @ 18:00: Posted
- Jan. 21 @ 17:50: Clarification about the random numbers—search for "mod"
- Jan. 23 @ 19:40: Recommended function call: `q1.merge_two_queues(q2)`

Due Date and Time

Due: **Thursday, February 2, 2017 at 9:00 PM** via electronic `handin` submission. Be sure to hand in the project using the online `handin` program that's described at the end of this document.

The **late penalty** is 3% per hour (or portion thereof), with no late assignments being accepted after 12 hours. For example, if you hand in your program at 02:10 AM on the morning after the due date, then this is 6 hours late; so, you would lose 6(3%) = 18% of the maximum possible mark.

You are allowed to work in pairs. If you work in a pair, then each partner is expected to contribute meaningfully to the project. Both partners will get the same grade. Please review Academic Conduct rules and keep in mind that violation of any of these rules constitutes academic misconduct and is subject to substantial penalties. If you are uncertain as to what is, or is not, reasonable collaboration, please contact your TAs or instructor.

Part marks will be awarded, so even if you don't finish everything, make sure that your code compiles on our undergrad machines (these machines tend to have the same software releases, so as long as it works on one, you're OK; our TAs will mark your code based on the undergrad environment). Use your judgment to determine whether it's worth turning the project in late, when considering the late penalties.

Objectives

- Gain more experience with C++
- Understand the notion of a C++ class, including member functions
- Work with linked lists and pointers
- Construct a two-queue simulation, using an airport runway as the application

C++ References

Notes on basic C++ programming can be found in numerous sources, including:

- Our textbook
- Your labs
- Lecture Notes (some)
- Online sources mentioned in the course outline on our course Web pages

Don't be alarmed by the word "project" on our Web site, since the code shouldn't require too many lines, and we're giving you the LQueue class. In other words, this "project" is quite do-able in the timeframe given.

Part 1: Add Two New Functions to the LQueue Class

In Part 1, you will use the LQueue class, header file, and driver (main function). You are provided with a set of sample code and a Makefile, and you should verify that you can compile, link, and execute the code successfully. You will be adding two new member functions to the LQueue class:

(1) A function to search through a queue for a particular key value, and if found, move the node to the front of the same queue. Call the void function `move_to_front` with an appropriate input parameter.

- This function might be useful in an airport situation where an emergency requires a plane to get landing priority (while other planes circle/wait); or, where a plane moves to the front of the queue for prioritized take-off.

(2) A function that takes two queues that are each ordered according to a given field (so be sure to include a sequence number, time, or some other measure in the data portion of a node), and merges the two queues into one queue whose absolute order of the merged elements is kept. For example, if queue 1 has nodes with sequence numbers of 7, 10, and 15; and if queue 2 has nodes with sequence numbers 1, 2, 7, 16, 17—then the result should be a single queue with order 1, 2, 7, 7 (arbitrarily break ties), 10, 15, 16, and 17. Call the function `merge_two_queues` with appropriate parameter(s). A recommended function call is: `q1.merge_two_queues(q2)` where queues 1 and 2 already exist. After the merge operation, queue 1 should have all the elements, and queue 2 should be empty. Note that the nodes were originally dynamically allocated, so it is OK to move them around without fearing that they'll be lost—providing you don't `delete` their storage or lose their addresses.

- This function might be useful in an airport with multiple runways, where one of the runways is temporarily shut down due to an emergency, bad weather, maintenance, etc.

Adequately test your new functions. You'll need to change the driver to convince yourself and the marker that your two functions work.

Part 2: Simulating Airport Runway Activity using Landing and Takeoff Queues

This problem is based on a question in Larry Nyhoff's book, "ADTs, Data Structures, and Problem Solving Using C++", and in particular Question 27 on Page 444. Here's the basic idea of the simulation. Consider an airport with one runway. At any given time during the day, there is a queue of zero or more airplanes waiting to takeoff (because only one plane is allowed to use the runway at a time). For example, another plane may be taking off, or another plane may be landing. Thus, there is also a landing queue.

This is a simulation; so, it will involve a random number generator. This is meant to reflect reality since we cannot predict exactly when a plane will arrive or depart due to numerous factors such as weather, winds, the departure time at the previous airport, the spacing of planes in the sky (by air traffic controllers), delays at the gate, the number of planes waiting to takeoff or land, contention during taxiing (e.g., when two planes in the same proximity are both in the process of leaving their respective gates to taxi to the runway), baggage handling backlog, security issues, plane de-icing, etc.

The takeoff rate and landing rate (in number of planes per hour) are user-supplied inputs, with priority given to landings. Write a program to simulate this airport's operation. Assume a simulated clock that advances in one-minute intervals. For each minute, generate two random numbers: If the first number *mod* 60 is less than the `landingRate` (e.g., 10, which means 10 planes per hour) then we will assume that a "landing" request has been generated, and that the plane is enqueued or added to the landing queue; and if the second is less than `takeoffRate`, a "takeoff" request has been generated, and the plane is added to the takeoff queue. Next, check whether the runway is free. If it is, then first check whether the landing queue is non-empty, and if so, allow the first airplane to land; otherwise, consider the takeoff queue. Have the program calculate the maximum queue length and the average time that an airplane spends in a queue. For these statistics, you don't need to add an element (private member variable) to the LQueue class; you can simply increment local counters using any reasonable interpretation for "average time".

Note that even if the runway is free, and the queue is empty, a plane still has to enter the appropriate queue, even if only for an instant. In such a case, if a plane is landing, the queue length is 1, and the average wait time is zero (since it entered the queue at time *t* and was permitted to start landing at time *t* in our simple simulation). A plane submitting a request to land at the 5-minute mark, and actually being permitted to start landing at the 12-minute mark, has spent 7 minutes in the landing queue (we don't count the time it takes for a plane to actually perform the landing on the runway, and we'll assume that the plane only enters the landing queue when it is ready to land; otherwise, it circles the airport while waiting).

Once a plane begins to takeoff, it finishes, even if a new plane requests to land before that plane finishes its takeoff. If there are multiple planes in the landing queue, then they all get serviced before a plane is allowed to takeoff. If a plane is taking off, and there is another plane in the takeoff queue, you must make sure that the landing queue is still empty before releasing that next plane for takeoff.

Your program might start as follows (the numbers are examples only):

```
bowen<221> runway
Enter:
  Time for a plane to land (in minutes): 3
  Time for a plane to takeOff (in minutes): 2
  Landing rate (planes per hour): 10
  Takeoff rate (planes per hour): 12
  How long to run the simulation (in minutes): 120
```

You can make up sequential identifiers (flight numbers) for planes, starting at 1000 (for example). These numbers can be your data element in the `LQueue` class (see the bottom of the `LQueue.h` file). Provide some running commentary in your output, so that the marker can understand what's going on. Here is an example of such comments (and the output from the program):

```
...
Time = 51
Plane 1007 wants to takeoff; added to takeoff queue; 1 in queue
Taking off:  Plane 1007
Time = 52
Plane 1008 wants to land; added to landing queue; 1 in queue
Time = 53
Takeoff complete; 0 in queue
Plane 1009 wants to land; added to landing queue; 2 in queue
Plane 1010 wants to takeoff; added to takeoff queue; 1 in queue
Landing:  Plane 1008
Time = 54
Time = 55
Plane 1011 wants to land; added to landing queue; 3 in queue
Time = 56
```

```
Landing complete; 2 in queue
Landing:  Plane 1009
...
Time = 120
No new takeoffs or landings will be generated.
...
Time = 131
Takeoff complete; 0 in queue
End of program.


STATISTICS

Maximum number of planes in landing queue was: 5
Average minutes spent waiting to land:  1.45
Maximum number of planes in takeoff queue was: 7
Average minutes spent waiting to takeoff:  6.333
```

In summary, loop through your program once per (simulated) "minute", and check the queues. Be sure to enqueue and dequeue any requests appropriately. You can use the remaining functions of the `LQueue` class as you see fit.

Finally, be creative and create one additional, non-trivial feature to your application. For example, you might want to incorporate one of the two functions from Part 1. The `move_to_front` function isn't hard to incorporate, so that might be worth 3-5% rather than 10%. However, the second is more involved, assuming you manage two runways. Projects that go well beyond expectations can earn a 5% bonus mark.

- Feel free to make any **reasonable assumptions** in your implementation, since there probably are some details that are missing from the above specifications. In a `README.txt` file, be sure to list any reasonable assumptions that you make, as well as any instructions to the marker.

The C++ libraries you probably need are `iostream`, `cstdlib`, and `ctime` (for randomization). The `srand()` and `time()` functions will be useful—the latter to seed the random number generator—although you may wish to use a constant seed (e.g., "5" or some other number for repeated random sequences during debugging).

Deliverables (to be submitted online, not on paper):

- All source code that you wrote (either `.cpp` or `.C` or `.h` files)
- A working `Makefile` so that the marker (and you) can compile the program.
- A `README.txt` file that gives any special instructions or comments to the marker:
    - Your name or, if you have a programming partner, both your names
    - Important: Only one partner should submit the files, but be sure to list both students' CS userids, names, and student numbers.
        - Partners: Please make sure you communicate with each other as to who is turning in the files—and when. You can always resubmit your files before the due time. After the deadline, you'll only get one submission attempt.
    - Acknowledgment of any assistance you received from anyone but your team members, the 221 staff, or the 221 textbooks, but please cite code quoted or

adapted directly from the texts (per the course's Collaboration policy)
    - A list of the files in your submission with a brief description of each file
    - Any special instructions for the marker
- Two sample output files consisting of output from your testing of Part 1 and Part 2.
- Do reasonable error checking, but you don't have to go overboard by checking for all kinds of bad input conditions. *You can assume that the user enters legitimate data*.
- You must comment your code adequately (i.e., "reasonably").
- And here is what you should *not* hand in: `.o` files, executables, core dumps, irrelevant stuff.

How to Use the Online "handin" Program

Please pay attention to the messages displayed on your screen during `handin`, to verify success. All submissions—on-time or not—are timestamped.

1. Create a directory called `~/cs221/proj1` (i.e., create directory `cs221` in your home directory, and then create a subdirectory within `cs221` called `proj1`).

2. To prepare to submit, first move or copy all of the files that you wish to hand in, to the `proj1` directory that you created in Step 1.

3. Before the due deadline, hand in your directory electronically, as follows:

```
handin cs221 proj1
```

Note that you will receive a set of confirmation messages. If you don't get any kind of an acknowledgement, then you probably did something wrong. Please re-read the instructions and try again.

4. You can overwrite an earlier submission—unless it is now past the deadline—by including the `-o` flag, and re-submitting, as follows:

```
handin -o cs221 proj1
```

You can hand in your files electronically as many times as you want. The latest timestamp will determine your official `handin` time.

5. Additional instructions about `handin`, if you need them, are listed in the `man` pages (i.e., by typing: `man handin`). At any time, you can see what files you have already handed in (and their sizes) by typing the command:

```
handin -c cs221 proj1
```

Note: If your files have zero bytes, then something went wrong and you should run the original `handin` command (without the `-c`) again.

6. To see the due dates and times for our course, type:

```
handin –l cs221
```

The first pair of dates and times is the actual deadline; the second pair is the last date and time after which late assignments will no longer be accepted.