COMSC-200 Lab 15
Using The STL

GOOD PROGRAMMING PRACTICES *show / hide*

ABOUT THIS ASSIGNMENT
In this lab session, you continue the ElevatorSimulation problem -- a problem that will be solved over labs 8-10 and 14-16.

As you complete this assignment, post the required files to the COMSC server. You may post them all at once, or one-at-a-time as you complete them, as you prefer. The individual labs are graded in order, starting with "a", which must be fully correct before "b" is graded, and so on. This assignment is worth 50-points, to be awarded after all labs are successfully completed. Use the "Submit your SP2015 work" link on the class website to post your file for this lab, using the **lab15** folder provided.

Lab 15 Using The STL [`Building.h and Building.cpp`]
Define the `Building` class in two new files: **Building.h** and **Building.cpp**. A single building object will contain multiple floors and elevators, in order to simulate the design of a proposed new building. The **Building** class will include a function to simulate all of the action that takes place in a single, one-second time-step. The `main` function, which we will write in lab 16, will call this function in a loop, to drive every second of the simulation.

Make sure that your floors are no closer than 100 inches from one another. Make sure that none of your elevators moves faster than 24 inches per second.

Here is what to expect:

1. Your program should place 10 riders into the building, one per second every second for the first ten seconds of the simulation, at randomly-determined floors.
2. Your elevators should run during the first ten seconds as the riders are placed, and continue for an additional 600 seconds (10 minutes) of simulation time.
3. At the end of the simulation (10 minutes and 10 seconds), all riders should have been picked up by an elevator and delivered to their destination.
4. At the end, having served the original 10 rides, the elevators should be idle, and the floors empty of waiting riders.

Include the following in the Building class' **class definition** in Building.h:

1. Prototypes for a default constructor, a destructor, and an overloaded `operator<<` friend function. Do *not* use friend relationships, except for overloaded stream insertion operators.

2. The simulation time in seconds, initialized to zero in the default constructor, and updated in the `step` function, explained below.

3. A vector of `Floor*`'s and a vector of `Elevator*`'s. (Do *not* include vectors of `Floor`'s and `Elevator`'s instead, because it would require overloaded `operator=`'s for each of these classes, due to their const data members.)

4. This prototype: `Building& step(int);`, which is the function that contains the instructions for moving elevators and riders during a one-second time-step. The `int` parameter indicates the number of randomly-placed riders to add in the time-step.

5. These inline functions, which you may use in your `step` function (or not). But if you do not use them, include them anyway -- they will be useful in lab 16.

- `int getFloorCount() const` // return #of floors in the vector of `Floor*`'s
- `int getElevatorCount() const` // return #of elevators in the vector of `Elevator*`'s
- `const Floor& getFloor(int index) const` // return a reference to the "indexth" floor
- `const Elevator& getElevator(int index) const` // return a reference to the "indexth" elevator

Include the following in the source file's **function definitions**:

1. In the default constructor, create *at least* 5 floors and 2 elevators, each with characteristics of your choosing. Use *dynamic memory allocation* to create these, and use the `push_back` function to add them to the `vectors`.

2. In the destructor, deallocate the floors and elevators that were created in the constructor. *HINT #1: Traverse the `vectors` with `for` loops.*

3. In the overloaded `operator<<` friend function for the Building class, first output the time with YOUR NAME and student ID, then output the elevators, and then output the floors. Output *each* elevator, but *only output the floors that have riders waiting*. Use Elevator's `operator<<` and Floor's `operator<<` in Building's `operator<<`. Be sure that your elevators "say" how many riders they carry, and their direction, and door status. Be sure the floors say their names, and numbers of up- and down-riders. Do NOT call getters of the Elevator and Floor classes in this friend function. *HINT #2: Traverse the `vectors` with `for` loops.*

4. In the `step` function, add the elevator simulation pseudo-code that was provided in previous labs and lecture notes (copy/paste is recommended). The function should return a reference to self, so that it could be used in chained calls (for example, `b.step(1).step(0);`) -- even though we will not be using it that way in this lab assignment. But we will make clever use of the self reference return in the `cout` statements in the for-loops in main (see below). Advice for writing this function will be presented in lecture.

HINT #3: Write step() in 3 phases -- the rider, floor, and elevator phases. After the rider phase, you should see stranded riders randomly placed on floors, and there should be no up-riders on the top floor and no down-riders on the bottom floor, and elevators should remain idle. After the floor phase, the elevators should choose a direction, but not move. After adding the elevator phase, the elevators should (a) stop to pickup riders on the floors on their way to another destination, and (b) go to floors with waiting riders after they reach an idle state. Test your simulation as you develop each phase, so that you can better find logic errors.

HINT #4: If yours does not work right, it's probably because your step function logic does not follow the pseudocode exactly. If you are not using all of the functions you wrote in the supporting classes in your step functions, you're logic is probably wrong. If you're adding logic to the if-statements in the pseudocode to try to fix your step function, the problem is probably someplace else in your step logic.

*Post **Building.h** and **Building.cpp** (and optionally **ElevatorSimulation.exe** -- see below) to the COMSC server for credit.* Do *not* submit the **ElevatorSimulation5.cpp** file or any other file with `int main()` in it. The ElevatorSimulation series is the subject of six (6) lab assignments, including the last one. *If you have to make corrections to already-submitted H or CPP files, do so into the folder for this lab -- do NOT overwrite your already-graded files from previous labs.*

To test your code, compile, build, and run with all previous ElevatorSimulation CPPs, and the following **ElevatorSimulation5.cpp**: Here's what it should do: (1) add one rider per second for 10 seconds, then (2) stop adding riders for the rest of the simulation, then (3) move those 10 riders to their destinations before the simulation ends. All this should be a 5 inches per second movement of the elevator, without sudden jumps that might injure or kill its passengers:

```
#include <iostream>
using std::cin;
using std::cout;
using std::endl;

#include <cstdlib>
#include <ctime>

#include "Building.h"

int main()
{
  srand(time(0)); rand(); // requires cstdlib and ctime
  Building building;

  // add one rider per second for 10 seconds
  for (int i = 0; i < 10; i++)
    cout << building.step(1) << endl;

  // continue the simulation for 600 more seconds (expect all riders to be gone by then)
  for (int i = 0; i < 600; i++)
  {
    if (!(i % 10)) // pause every 10 seconds
    {
       cout << "Press ENTER to continue, X-ENTER to quit...\n";
      if (cin.get() > 31) break;
    }
    cout << building.step(0) << endl; // time-step with no riders added
  }
  cout << << "DONE: All riders should be gone, and all elevators idle\n";
}
```

You might find this useful. The getDifferentInts function randomly selects 2 integers that are sure to be different from eachother:

```
void getDifferentInts(int max, int& a, int& b)
{
  do
  {
    a = rand() % max; // range is 0 to (max-1)
    b = rand() % max; // range is 0 to (max-1)
  } while (a == b); // try again if they are the same
}
```

Here's how to get a different set of random numbers every time you run your program -- otherwise you get the same sequence every time, and the game plays out the same every time:

```
// execute this ONCE in your program -- as the first statements in main
  srand(time(0)); rand(); // requires cstdlib and ctime
```

But add this only AFTER you are sure your step function works right. It's better to have the exact same random number stream every time you run

while you are developing and debugging. To test with a different sequence, just do one or more rand() calls without srand.

Command Line Compiling With Visual C++:
Using the Rider and Elevator classes from lab 10, and the Floor class from lab 14, here is the command for compiling this project in Visual C++ (assuming that main is in a file named **ElevatorSimulation5.cpp**):

```
cl ElevatorSimulation5.cpp Rider.cpp Floor.cpp Elevator.cpp Building.cpp -EHs
```

This creates an executable named **ElevatorSimulation5.exe**. To run, type the command **ElevatorSimulation5**.

Alternatively, to compile each code module separately:

```
cl ElevatorSimulation5.cpp -c -EHs
cl Rider.cpp -c -EHs
cl Floor.cpp -c -EHs
cl Elevator.cpp -c -EHs
cl Building.cpp -c -EHs
cl ElevatorSimulation5.obj Rider.obj Floor.obj Elevator.obj Building.obj
```

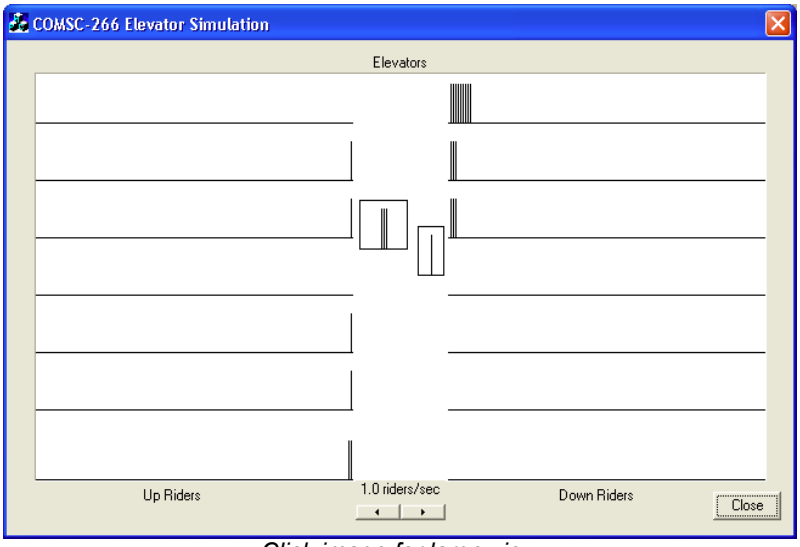
Command Line Compiling With g++:
Here is the command for compiling this project in g++ (assuming that main is in a file named **ElevatorSimulation5.cpp**):

```
g++ ElevatorSimulation5.cpp Rider.cpp Floor.cpp Elevator.cpp Building.cpp
```

This creates an executable named **a.exe** on a PC or **a.out** on a Mac. To run, type the command **a** on a PC or **./a.out** on a Mac.

Alternatively, to compile each code module separately:

```
g++ -c ElevatorSimulation5.cpp
g++ -c Rider.cpp
g++ -c Floor.cpp
g++ -c Elevator.cpp
g++ -c Building.cpp
g++ ElevatorSimulation5.o Rider.o Floor.o Elevator.o Building.o
```



Click image for large view

You can create a GUI version of your elevator sumulation, using Visual C++. Right-click here to download a ZIP file with the project files. Expand it to your desktop, where it creates a folder named **ElevatorSimulation**. Copy the CPPs and Hs for your 4 classes (for a total of 8 files) to this new ElevatorSimulation folder. Open the folder and double-click the **ElevatorSimulation.dsw** file, which should start Visual C++. (If it does not, start it yourself and open the project in this folder.) Use Build->Rebuild Solution menu option to create the program.

You do not have to submit this -- it's for your own enjoyment. But if you want to do so, submit the EXE only, as **ElevatorSimulation.exe**.

How to test and debug

First test only the RIDER loop. You should see 10 riders appear on floors at random locations. You should see no up-riders on the top floor, and no down-riders on the bottom floor.

Temporarily modify the RIDER loop in order to test the elevator operation in the following steps. In this modification, YOU CHOOSE the start and destination floors -- to NOT let this be random. Do NOT place riders on the floors where your elevators start. You should see 10 riders appear on floors chosen by you. The elevators should remain idle, at their starting locations.

Then test the FLOOR loop, with only the IF part of the "// if this elevator's idle..." if-else-if structure. This logic "calls" idle elevators to floors with waiting riders. You should see 10 riders waiting on floors as before, but the elevators shoould have closed doors, with directions and destinations. The elevators, however, will not be moving.

Before going any further with the FLOOR loop, start the ELEVATOR loop. But include only the "// if elevator door is closed (move up or down)" and "// if not near enough to destination to reach it in this time step, continue moving" parts of the if-else-if structures. This should cause the elevators to move towards their destinations, but pause just short of getting there.

In the ELEVATOR loop, write the "// otherwise it's near enough to destination to reach it in this time step..." code block, up to but not including the "// get a non-const pointer..." part. Now the elevators should reach their destinations, and open their doors, but hover there with no change of direction nor boarding of riders.

In the ELEVATOR loop's "// otherwise it's near enough to destination..." code block, implement the logic through and including the "// if elevator is empty..." if-structure. Now the elevators should set their directions, although remain hovering. You'll have to pay attention to the arrivals on the floors of your simulation so that you know if an up-rider or a down-rider arrived at the elevators' destination floors first. That way you can decide if the elevators are choosing their directions correctly, because the newly-set direction should be up if an up-rider appeared first, and down otherwise.

In the ELEVATOR loop's "// otherwise it's near enough to destination..." code block, implement the logic through and including the "// if there is space in the elevator..." if-structure. Now your elevators should board riders from their destination floors, while remaining hovering with doors still open.

In the ELEVATOR loop, implement the "// otherwise (then it already let off riders, or is in its initial state)" logic with the "// if elevator has any more riders...". Elevators with riders should now close their doors, although continue to hover. Elevators with no riders will remain with open doors.

In the ELEVATOR loop, implement "// reassess elevator destination based on its riders", and things should now begin to operator correctly. All riders should be picked up and delivered to their destinations. The elevators should hover at their last-reached locations, with no riders, but not getting set to idle.

In the ELEVATOR loop, implement "otherwise" and "// tell elevator to go idle" -- now the elevators should go idle.

You should notice that elevators going up are bypassing floors with up-riders (and similarly for down), coming back to get them later. This may be fine for the riders on the elevators, but it's exasperating to the riders on the floors as they watch elevators pass them by without stopping. So implement the two "// else if..." code blocks in the FLOOR loop, and see if this fixes that problem.

Elevator Simulation Index: where to find stuff...

class Rider, lab 8	class Elevator, lab 9	class Floor, lab 14
Rider::operator=, lecture 9	Elevator EXEs, lecture 9	Floor::addNewRider, lecture 14
Elevator::isNearDestination, lab 9	Elevator::moveToDestinationFloor, lab 9	Floor::isPreferredDirectionUp, lecture 14
class Building, lab 15	Elevator::addRiders, lecture 10	Floor::removeDownRiders, lecture 14
getDifferentInts(int, int&, int&), lab 14	Elevator::removeRidersForDestinationFloor, lecture 10	Floor::removeUpRiders, lecture 14
poissonHits(double), lab 15	Elevator::setDestinationBasedOnRiders, lecture 10	GUI simulation option, lab 15

How to pause a console program: *show / hide*

GRADING POLICY *show/hide*
