CSE 3150 — C++ Essentials — Yufeng Wu — Fall 2024
Project Part 1. Due: 11:59 pm @ 11/15/2024.

You will have two weeks for this part of project.
Warning: this assignment is perhaps more difficult than the previous ones. So start early.

In this part of the project, you will implement a simulator for elevator. I am sure you have used elevator before. Now you are to implement a class for elevator simulation. Please note: make sure you follow the taught object orientation approaches. We will perform manual code inspection (see below).

1 Problem specification

We assume the following.

1. There is a single elevator cabin. For simplicity, we don't model door open/close or signals for floor arrivals.

2. Floors are numbered from 1 (lowest) to K (highest). The cabin position is at one of these floors (at floor one at start). We don't model positions between floors.

3. A user is modeled by the floor he/she is currently at, and the floor going to. A user requests service by pushing two buttons. (i) Inside the elevator cabin, there is a button for each of K floors. (ii) Outside the cabin, there is a button "UP" for each floor except the topmost floor, and a button "DOWN" for each floor except the lowest floor. Note that this is only conceptual; we won't implement these buttons in our program.

4. Once a request (initiated by pushing a button) is made, the request is queued; we don't allow cancellation of request.

5. We assume all events are ordered. There is no delay in responding to a request. That is, if the elevator is stopped and there is a request coming, the elevator would move immediately. You may assume the events are ordered in the time of the event/request is made.

6. Once the cabin stops at a floor, all users waiting at this floor automatically enter the cabin (regardless which direction this user is going; for example, a user going downwards may first go up and then down) and push the wanted floor button (no action from the user needed).

7. The elevator should stop at current position if there is no outstanding requests. If there is at least one outstanding request, the elevator will move towards one requested floor.

8. The order of request servicing is decided by which floor a request is made (either inside the cabin or outside) and the moving direction.

   (a) If the cabin is not moving and there is a request from floor, the cabin moves towards that requested floor. If there are multiple requests at the same time, go to the nearest floor (if there is a tie, i.e. two requests from the same distance to the current floor but on different side, prefer going up).

   (b) When the cabin is moving (say upwards), the cabin will stop at each requested floor sequentially (whether it is made inside the cabin or at a floor) if that floor is along the moving direction. This is regardless the order these requested were made.

9. We model the timing of elevator in the following simple way. Time starts at zero. It takes 1 time unit the cabin to move from one floor to the next. There is no delay in start: if there is a request while the elevator is not moving, it starts moving immediately. The stopping time for loading/unloading passengers is also 1 unit. That is, it take 1 time unit to load/unload any number of users at a floor during a stop. Note: if a passenger arrives at the floor where the elevator is parked at, there is no delay and the elevator would leave immediately. All requests are made at some whole time units (i.e., we ignore the fraction of time). All requests by user are at some specific time point.

10. The following are the key numbers to keep track during simulation.

    (a) The arrival time of each passenger. The arrival time is the time that the elevator first arrives at the destination floor. That is, a passenger is considered to be "arrived" right at the time the elevator arrives at a floor (doesn't consider the loading/unloading time).

    (b) The current floor of the elevator at a specific time during simulation.

The above should fully specify the behavior of the elevator. That is, your code should produce the same results as my code if it works. If you think there is still ambiguity, ask in Piazza.

2 Grading

We will have autograder for testing functionalities. Moreover, we will also perform automated tests for code quality to ensure your code follows the taught object orientation techniques. Here is what we will check during code inspection.

1. No long functions. You will lose points if your code has a function that is longer than **100** lines of code (when formatted in the common way, e.g., one statement per line; properly commented; etc).

2. No complex loops (e.g., nested loops and/or complex loop breaking conditions). Complex loops are difficult to debug and maintain. Try to avoid such complex code by re-factoring your code. We will use cyclomatic complexity tests to evaluate your code.

3. No long chained conditional statements. Long chained conditional statements usually indicate a poor design from object orientation perspective. If you find yourself writing such complex code, it is time to consider re-factor your code. I would suggest using virtual functions: when used properly, virtual functions can simplify code structure. We will use cyclomatic complexity to measure the complexity of your code.

4. You should use *multiple* classes which include base classes and sub-classes. That is, you must use inheritance in some way.

5. You must use "virtual" functions.

6. You need to use at least one pure virtual function. This is to let you practice this key C++ feature.