Elevator Simulation

Consider an elevator system, similar to the one on McBryde Hall. At any given time, there may be zero or more elevators in operation. Each operating elevator will be on a particular floor, numbered 1 through n. An elevator may be empty or contain one or more passengers. For the purposes of this assignment, there is no upper bound on the number of passengers an elevator may contain.

An elevator *call* may be issued from any floor; once a call has been issued, it is *outstanding* until an elevator arrives at that floor.

There is a single controller, which is responsible for managing the simulation. The controller will:

- Maintain a simulated clock storing the official simulation time. This "clock" will simply be a counter; do not attempt to synchronize your program to the hardware clock or an OS timer.
- Keep track of which floors have outstanding calls for an elevator.
- Notify an elevator whether it is "broken" or is "fixed".
- Cause other components of the simulation to update themselves on each "tick" of the simulated clock. Each "tick" corresponds to a simulated second of elapsed time and is simulated by the incrementing of the counter for the simulated clock. Elevators will update their state in the order they are numbered.

The controller will manage the interface with the input file parser; no other component of the simulation should read input. The controller will also handle `display` and `stats` commands (described below).

There will be one or more elevators, numbered sequentially starting with zero. Each elevator will:

- Keep track of a list of its current passengers.
- Keep track of destinations requested by current passengers.
- Keep track of the current destination it (the elevator) is attempting to reach. Note this may not be the next floor on which it will actually stop.
- Keep track of whether it is broken or operational.
- Keep track of the time delay when it stops to load/unload passengers (specified below).
- Keep track of the time delay when moving between adjacent floors (specified below).
- Keep track of its current location. This may be more complex than a floor number.
- Keep track of its current direction of movement (if any).
- Update its state when the simulated clock "ticks".

Some of these will require data members and associated operations; others may be computed when needed from other data values that are maintained continuously. In addition, the operation of each elevator must be governed by the set of Simulation Rules given below. Failure to properly implement these rules will almost certainly cause results that are inconsistent with this specification, and such results will probably not receive full credit.

The simulation is inherently time-driven. It is acceptable to manage this by simulating the passage of time via a loop counter and updating relevant objects on each "tick" of the simulated clock. You will not make the simulation operate in real-time. All references to "time" in this specification are to the value of the simulated clock.

The specification above suggests a number of objects that should be included in your design and implementation. There are additional objects that are implied and should also be included.

Input

Input to the simulation will be provided in an input file, which will be named on the command-line when the simulation is invoked. The input file will begin with a header section that specifies the initial state of the system

```
StartTime:         <time value in hh:mm:ss format>
NumberOfFloors:    <number of floors>
NumberOfElevators: <number of elevators>
```

All time values specified in the input file will be in hh:mm:ss format, where hh, mm and ss are integer values. Leading zeros will be shown (e.g., 14:05:00) and a 24-hour clock will be used. Any time values printed in your output must also be in this format.

Next there will be a section that specifies the initial location (floor) for each elevator in the system. Each line of this section will have the format:

```
init  <elevator number> <floor number>
```

Elevators will be numbered from 0 to m-1, where m is the number of elevators in the system.

This section will be followed by an arbitrary number of lines, each containing a single command to be processed by the simulation. Commands are as follows:

```
call  <name>      <floor>     <time>      <dest>
```

meaning that, at the given time, the person named has issued a call for an elevator to come to the specified floor. For simplicity, the destination of the caller is also given as part of this command.

```
fail  <elevatornumber>  <time>
```

meaning that the specified elevator experienced a failure at the given time. If an elevator fails, it remains inoperative until a "fix" command is received for it.

```
fix   <elevatornumber>  <time>
```

meaning that the specified elevator (which is presumably broken) was fixed and became operative at the given time.

```
display     <time>
```

meaning that information about the state of the simulation should be written to standard output, at the specified time. The content and formatting requirements for this output are specified in the Output section below.

```
stats <time>
```

meaning that statistical information about the simulation up to the specified time should be written to standard output. The content and formatting requirements for this output are specified in the Output section below.

The elements of a command will be tab-separated; elements such as names may contain embedded spaces.

The commands in the input file will be listed in order of ascending timestamps. Commands will be syntactically correct, although logical errors may occur; the simulation should recognize logically invalid commands and handle them appropriately. Each such error should result in a detailed entry being appended to a log file named "`error.log`".

Each command should be recorded in some fashion when the simulation time reaches the timestamp of the command; this will change the state of the system, and that change of state will eventually cause some action to be taken. For example,

```
call  Homer Simpson     08:43:27    1
```

will eventually result in an elevator stopping on floor 3 and Homer Simpson entering that elevator (but this will almost certainly not happen at system time 8:43:27). The point is the system must "remember" that Homer Simpson has called an elevator to the third floor, until an elevator stops there; when that elevator arrives will depend on the initial state of the system and the preceding commands that have been encountered. Note: commands are really events.

Sample input files will be posted on the course website by Friday, October 15.

Output

When a `display` command is simulated, the following information should be printed. For each elevator, print the elevator number, its current location, its current direction of travel (if any), and its current status (broken, stopped, moving, etc.), and a list of the names of all current passengers. Output should be clearly labeled and neatly arranged, with some reasonable care given to making the output compact.

When a `stats` command is simulated, the following information should be printed. For each elevator, print the total number of passengers that elevator has delivered to their destination, the number of current passengers, and the total distance the elevator has traveled since the beginning of the simulation.

Simulation Rules:

The following rules are a crucial part of this specification. Hopefully, they are complete. Ideally, your design and implementation will make it easy to adapt to changes in the rules.

1. Each elevator is initially operational, stopped on floor 1, empty and has no destination. All elevators remain in that state until a call occurs.
2. The initial destination of each elevator will be the floor on which the first call occurs. Subsequent destinations are selected according to Rule 12.
3. If an elevator fails, it will retain its previous state (passengers, direction, location) until it is later fixed or the simulation ends.
4. If an elevator reaches the floor requested by one or more current passengers, the elevator will stop on that floor, and those passengers will leave the elevator (even if this floor is not the elevator's current destination).
5. If an elevator reaches a floor from which a call is pending, and no other elevator is currently stopped on that floor, it will stop on that floor. Once an elevator has stopped on a floor from which a call is outstanding, the call is no longer outstanding.
6. If an elevator stops on a floor where passengers are waiting (after making a call), all those passengers will enter that elevator. If two or more elevators reach that floor at the same time, the elevator with the lowest number will stop and the others will not stop unless they are carrying passenger who are to get off on that floor. In any case, if two or more elevators stop on a floor at the same time, any waiting passengers will board the elevator with the lowest number, whether it is going in the direction they wish or not.
7. If an elevator stops on a floor and its last passenger disembarks, the elevator will remain stopped on that floor until a call occurs. The elevator will then take as its destination the floor from which the call was issued.
8. It takes an elevator 5 seconds to move from one floor to an adjacent floor.
9. If an elevator stops on a floor, it will remain on that floor for 10 seconds. There is no "Door Close" button or "Stop" button in these elevators. After that, it will proceed towards its destination if it has one, selecting a new destination if necessary. Otherwise, the elevator will remain on that floor.

10. On each tick of the clock, an elevator will:

    (a) Check if it is moving, and update its location if necessary.
    (b) Check if it is stopped for passengers to enter/exit, update the countdown until it can move again.
    (c) Check its current destination. If the destination is now invalid (e.g., another elevator has serviced the call this one was responding to), the elevator must choose a new destination. If the destination is valid, the elevator must determine if it has reached that destination.

11. If an elevator reaches its destination, it must stop there for passengers to enter/exit and then choose a new destination.

12. An elevator chooses its next destination by considering the following possibilities in the given order:
    (a) First, if there's an outstanding call from a floor in the same direction the elevator last moved, that floor will become the elevator's next destination.
    (b) Second, if the elevator is carrying passengers, the destination of the senior passenger (one longest aboard) will become the next destination of the elevator.
    (c) Third, if there is an outstanding call in the opposite direction the elevator last moved, that floor will become the elevator's next destination.
    (d) Fourth, then there is nothing for the elevator to respond to at this time and the elevator will have no destination. In this case, the elevator will remain stopped in its current location (which may, under certain conditions, be between floors).

Programming Standards:

You'll be expected to observe good programming/documentation standards, as described in the *Elements of Programming Style*. Some specifics:

- You must include a header comment, preceding `main()`, specifying the compiler and operating system used and the date completed.
- Each .cpp and .h file must begin with a header comment block containing the name and e-mail address of programmer and the date of last modification of file
- You must include a comment explaining the purpose of every variable or named constant you use in your program.
- You must use meaningful identifier names that suggest the meaning or purpose of the constant, variable, function, etc.
- Always use named constants or enumerated types instead of literal constants in the code.
- Precede every major block of your code with a comment explaining its purpose. You don't have to describe how it works unless you do something so sneaky it deserves special recognition.
- You must use indentation and blank lines to make control structures more readable.
- Precede each function and/or class method with a header comment describing what the function does, the logical significance of each parameter (if any), and pre- and post-conditions.
- Decompose your design logically, identifying which components should be objects and what operations should be encapsulated for each.
- The organization of your implementation into `.h` and `.cpp` files should follow your design. In particular, classes should be declared and implemented in `.h/.cpp` pairs, named with the corresponding class name.

Neither the GTAs nor the instructors will help any student debug an implementation, unless it is properly documented and exhibits good programming style. Be sure to begin your internal documentation right from the start.

You may only use code you have written, either specifically for this project or for earlier programs, or code taken from the textbook, the notes, and the CS 2704 website. Note that code examples from these sources are not designed for the specific purpose of this assignment, and are therefore likely to require modification. Such code may, however, provide a useful starting point. You may **not** use code from STL, MFC, or a similar library for your linked list implementation.

Evaluation

Your score on this project will be based on several factors:

- Runtime testing of your program, and correctness and completeness of output.
- Quality of external documentation (class diagram, class/operation forms) and amount of difference between the interim design and the final design.
- Quality of design, and adherence to good software engineering practice.
- Quality of internal documentation.

The relative weighting of these factors will be decided later. We may require you to demonstrate your program. To receive partial credit for programs that are non-working, or are not fully functional, a brief one or two paragraph description of the problem(s) must be included in the assignment submission, in an ASCII text file named `problems.txt`. The location of the problem, minimally identified to a specific function, must also be specified along with possible corrections that need to be made.

Deliverables

You must submit (electronically) an interim design document, containing a class diagram and class/operation forms, no later than the midnight on Friday October 15. Your submission must be either readable by MS Word or a PDF file. **Do not zip the file and do not submit multi-file designs.**

Your final project submission must include:

- All source code (`*.cpp` and `*.h` files) comprising your project.
- MS Visual C++ project files (`dsp` and `dsw`) or Unix `makefile`, as appropriate. (VC++ users: do not submit unnecessary files, such as `ncb`, `opt`, `ilk`, `obj`, `pch`, or `pdb` files.
- One set of input/output files; this should include redirected output.
- Revised design documentation reflecting the final design of your project.
- A brief ASCII text readme file, named `readme.txt`, containing:
  - Your name and e-mail address.
  - The section of the course you're enrolled in (example : MWF 11:00-11:50).
  - Your instructor's name.
  - The project name.
  - The platform and compiler you're using (example: MSVC++ 6.0 under NT).
  - Any special execution instructions. (If your program is not fully functional, be sure to mention that here and to follow the directions in the Evaluation section above.)
- An ASCII text file, named `pledge.txt`, containing the Honor Code Pledge listed below:

```
On my honor:
   - I have not discussed the C++ language code in my program
      with anyone other than my instructor or the teaching
      assistants assigned to this course.
   - I have not used C++ language code obtained from another
      student, or any other unauthorized source, either
      modified or unmodified.
   - If any C++ language code or documentation used in my
      program was obtained from another source, such as a
      textbook or course notes, that has been clearly noted
      with a proper citation in the comments of my program.

      student's name
```

Your project submission will consist of a zipped archive file containing all of the items specified above.

You are allowed to submit your solution up to five times, in case you detect (or solve) problems after your first submission. Your last submission will be the only one tested and graded. Note that, due to late penalties, it is possible that a fixed but late submission may receive a lower score than a faulty but on-time submission.