CSC 330 Object-oriented Software Design   Hw1 – Classes in C++

Design a program that simulates an elevator. The elevator serves floors from zero (the basement) to the top floor. It is an old elevator and it's not automatic. When people get in the elevator, they enter their desired floor number. Several numbers can be requested at a time. After all numbers have been entered, the door is closed by pressing the close door number (the return key).

Each time the door closes, the elevator checks to see if there are any floors in the current direction (up or down). If there are, then it services these floors first, starting with the closest one to the current floor. If there are no floors requiring stops in the current directions, it checks the opposite direction, again servicing the floor closest to the current floor. If the elevator is not moving (direction **STOP**), then it services up requests before down requests.

Each time, the elevator arrives at a floor, new passengers can get on and request a floor. The new requests are added to the ones still pending, and the elevator again evaluates which floor will be processed first.

The structure for this program is shown in ***Fig.1***. The elevator is represented as a structure with three fields: the current floor, a pointer to an array of buttons, and the current direction of the elevator. The button values are IN, meaning the floor has been requested, and OUT, meaning the floor has not been requested. After a floor has been serviced, the button is reset. The direction values are **UP**, **DOWN**, and **STOP**.

Fig. 2 shows a structure chart for the program, while ***Fig. 3*** presents a state diagram for the elevator. An elevator can be in one of three states: moving up, moving down, or stopped. To move from one state to another, a change must occur in the elevator environment. For example, to change from the stop state to the up state, a button must be pressed. This is reflected on the line between stop and up as *anyUp*.

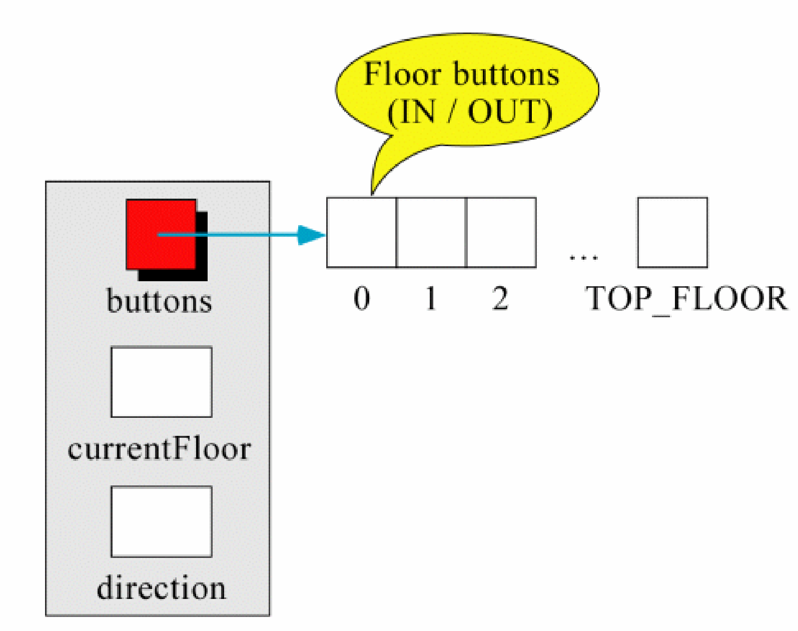


Fig. 1. Elevator structure

,

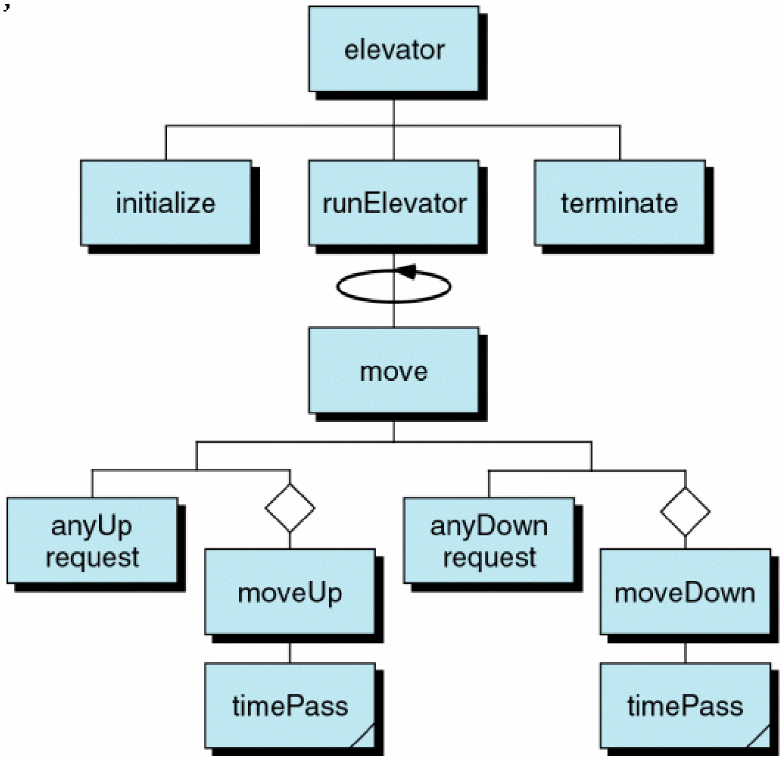


Fig. 2. Elevator structure chart

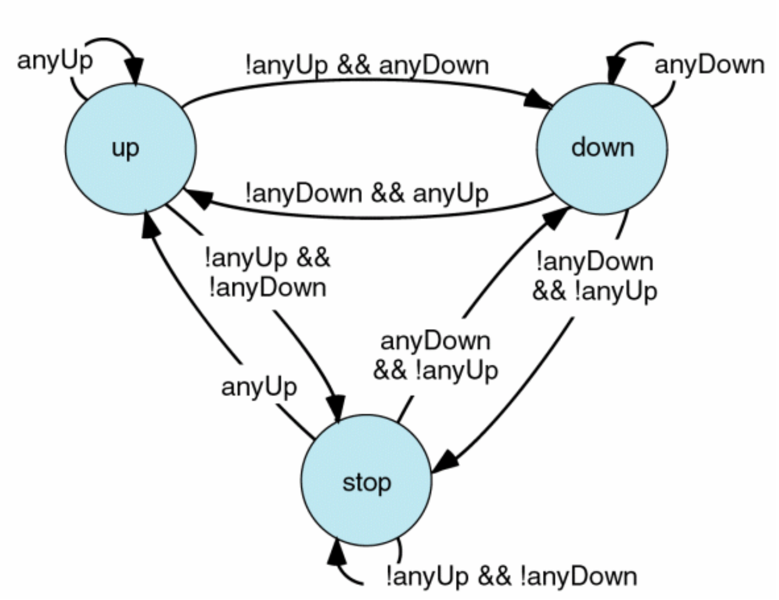


Fig. 3. Elevator states

<u>Class definition</u>

```
//       Elevator class
//       See elevClas.h for implementation

const int TOP_FLOOR = 8;
const int DELAY_FACTOR = 10000;

class Elevator
{
         private:
           enum BUTTONS   {OUT, IN};
           enum DIRECTION {DOWN, STOP, UP};

           BUTTONS*  buttons;
           int       currentFloor;
           DIRECTION direction;

           void move           (void);
           void moveUp         (void);
           void moveDown       (void);
           bool anyUpRequest   (void);
           bool anyDownRequest (void);
           void timePass       (int time);

         public:
               Elevator    (void);
              ~Elevator    (void);
           void  runElevator (void);
} ; // Class Elevator
```

Elevator constructor

```
//      Elevator constructor
//      See elevClas.h for implementation

/*      =================== Constructor =====================
        This function dynamically allocates memory locations for
        the buttons and initializes the current floor to 1 to show
        that the elevator is parked in the first floor.
        Pre:  Nothing
        Post: Elevator created, all buttons are reset, and
              elevator is parked at first floor (not basement).
*/
Elevator::Elevator (void)
{
//      Local Declarations
        int       i;

//      Statements
        buttons = (BUTTONS*) new(BUTTONS[TOP_FLOOR + 1]);

        for (i = 0; i <= TOP_FLOOR; i++)
          buttons[i]  = OUT;
        currentFloor   = 1;
        direction      = STOP;
}       // initialize
```

Run elevator

```
//      runElevator
//      See evelClas.h for implementation

/*      =================== runElevator =====================
        This function simulates the operation of the elevator.
        Pre:  The elevator structure has been initialized.
        Post: The simulation is complete.
*/
void Elevator::runElevator (void)
{
//      Local Declarations
        char  aCh;
        int   floor;

//      Message Constants
        const char* INSTRUCTION1 =
          "\n\nThis elevator goes from basement (0) to floor ";
        const char* INSTRUCTION2 =
          "\nType floors & press return key to start";
        const char* INSTRUCTION3 =
          "\nIf no new floors, just press return key.";
        const char* INSTRUCTION4 =
          "\nTo quit, enter <Q> \n\nPlease enter floors: ";
        const char* INVALID_FLOOR = " not a valid floor.\n";
        const char* CURRENT_FLOOR = "\aAlready on Floor ";
```

```
          const char* INVALID_IP =
                  "\n\aInvalid Input. Please re-enter: ";
          const char* NEXT_FLOOR =
                  "\n\nPlease enter next floors or <Q>: ";

//        Statements
          cout << INSTRUCTION1 << TOP_FLOOR
             << INSTRUCTION2 << INSTRUCTION3
             << INSTRUCTION4;
          aCh = toupper( cin.get () );
          while ( aCh != 'Q')
            {
            do
              {
               if (isdigit(aCh))
                 {
                  // Convert digit to decimal
                  floor = (aCh - '0');
                  if (floor < 0 || floor > TOP_FLOOR)
                     cout << "\n\a" << floor << INVALID_FLOOR;
                  else
                     if (floor == currentFloor)
                        cout << CURRENT_FLOOR << floor;
                     else
                        buttons[floor] = IN;
                 } // if digit
               else
                  if (!isspace(aCh) && aCh != 'Q')
                     cout << INVALID_IP;
               if (aCh != '\n')
                  // Read next floor
                  aCh = toupper( cin.get () );
              }  while ( aCh != '\n' && aCh != 'Q');
            if (aCh != 'Q')
               move ( );
            cout << NEXT_FLOOR;
            aCh = toupper( cin.get () );
            } // while
          return;
}         // runElevator
```

<u>Elevator: anyDownRequest</u>

```
//        Elevator: anyDownRequest
//        See elevClas.h for implementation

/*        ================== anyDownRequest ==================
          This function checks to see if any request is for a floor
          below the current floor.
          Pre:  The elevator.
          Post: returns true if button below current floor pushed.
                returns false otherwise.
*/
bool Elevator::anyDownRequest (void)
```

```
{
//          Local Declarations
            int  check;
            bool isAny = false;

//          Statements
            for (check = currentFloor;
                 check >=  0 && !isAny;
                 check--)
                   isAny = (buttons[check] == IN);

            return isAny;
}           // anyDownRequest
```

Elevator: timePass

```
//          Elevator: timePass
//          See elevClas.h for implementation

/*          =================== timePass =====================
            This function simulates the concept of passing time by
            executing an empty for-loop.
            Pre:        The time to be passed (number of moments).
            Post:       Time has passed.
*/
void Elevator::timePass  (int time)
 {
//          Local Declarations
            long i;

//          Statements
            for (i = 0; i < (time * DELAY_FACTOR); i++)
                ;
            return;
}           // timePass
```

Design the remaining member functions:

1.

```
/*          =============== Elevator Destructor =================
            Release the memory occupied by buttons.
            Pre    The elevator.
            Post   The memory is released.
*/
Elevator::~Elevator (void)
```

2.

```
//          Elevator: move
//          See elevClas.h for implementation
```

```
/*       ==================== move =====================
         Moves the elevator to a requested floor. It stops
         the elevator after responding to one request.
         Pre    The elevator.
         Post   The elevator has been moved. While it is
                moving, the floors are called out.
*/
void Elevator::move (void)
```

3.

```
//       Elevator: moveUp
//       See elevClass.h for implementation

/*       ==================== moveUp ======================
         This function simulates the movement of the elevator
         when it is going up.
         Pre:  The elevator.
         Post: The up simulation is displayed on the screen.
*/
void  Elevator::moveUp (void)
```

4.

```
//       Elevator: moveDown
//       See elevClas.h for implementation

/*       ==================== moveDown ======================
         This function simulates the movement of the elevator when
         it is going down.
         Pre    The elevator.
         Post   The down simulation displayed on screen.
*/
void Elevator::moveDown (void)
```

5.

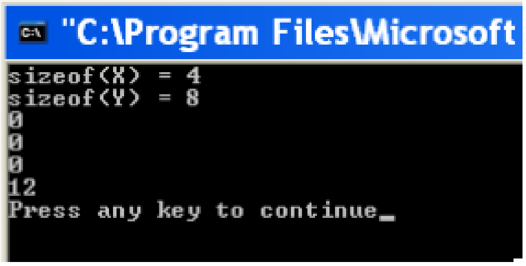
```
//       Program 11-24 Elevator: anyUpRequest
//       See elevClas.h for implementation

/*       =================== anyUpRequest =====================
         This function checks to see if any request is for a floor
         above the current floor.
         Pre:  The elevator.
         Post: returns true if button above current floor pushed.
               returns false otherwise.
*/
bool Elevator::anyUpRequest (void)
```

6. Function **main** to test your functions and display respective messages.

Requirements to submission :

PRINT:

1. Draw the detailed UML class diagrams.
2. Complete source code (all necessary .h and .cpp files) with comments.
3. Testing snapshots in the shown on the right format.



```
sizeof(X) = 4
sizeof(Y) = 8
0
0
0
12
Press any key to continue_
```

Electronic submission (on my pen drive):

All of the above plus the exe file.

Don't forget to include as comments:

1. **Your name**
2. **CSC330 HW1**