```cpp
#pragma once

#include <chrono>

#include "main.hpp"

/// THe only thing accessing the clock is the memory
// no need for multi-thread protection

class Clock {
    typename clk::time_point mStart; // this stores when the simulation
    // started, it is absolute

    unsigned mHour; // this stores the start time, it wont change
    // unless the simulation changes.
    unsigned mMinute;

    // Forexample if the simulation states that it started at 14:59 then the
    // mAbsTime will store the start of the system clock, and mHour will store
    // 14 and mMinute stores 59. To figure out the current time

public:
    Clock();

    void reset();
    void reset(bool hour, unsigned val);

    unsigned getHour();
    unsigned getMin();
};
```

```cpp
#pragma once

#include <thread>

enum class ControllerState {
    CTR_ASK = 0b00,
    CTR_UP,
    CTR_DOWN,
    CTR_WAIT_LATCH,
    CTR_WAIT_REPLY,
    CTR_WAIT_DCLOSE,

    CTR_DC1 = 0b110,
    CTR_DC2
};

struct Controller {
    std::thread mThread;
    ControllerState mState;

    Controller();
    ~Controller();

    void start();

};
```

```cpp
#pragma once

#include <atomic>

#include "ir_sensor.hpp"

enum DoorState {
    DOOR_CLOSED = 0b000,
    DOOR_LAT_SND,
    DOOR_OPEN,
    DOOR_IR,
    DOOR_CLOSING,
    DOOR_DC1 = 0b101, // not needed, just so we can remember
    DOOR_DC2,
    DOOR_DC3
};

inline std::string pretty(DoorState s) {
    switch (s) {
        case DOOR_CLOSED:
            return "Closed";
        case DOOR_LAT_SND:
            return "Latch and Sound";
        case DOOR_OPEN:
            return "Open";
        case DOOR_IR:
            return "Waiting for IR";
        case DOOR_CLOSING:
            return "Door Closing";

        default:
            return "Broken";
    }
}

struct Door {
    DoorState mState;
    IR_Sensor mIRSen;

    std::atomic_bool mConDoorOpen;
    std::atomic_bool mConDoorClose;

    void start();

};
```

```cpp
#pragma once

#include <iostream>
#include <chrono>

#include "door.hpp"

/// CHECKED FOR THREAD SAFTEY //

enum FloorNum {
    FG = 0,
    F1,
    F2,
    F3
};

class invalid_floor_reached : public std::exception {
public:
    std::string what() {
        return "Umm, ya you killed people. Invalid floor reached";
    }
};

/// I know I know What am I doing here, Do I even know C++.
/// Ill fix it later
inline void inc(std::atomic<FloorNum>& a) {
    switch (a) {
        case FG: a = F1; return;
        case F1: a = F2; return;
        case F2: a = F3; return;
        default:         throw invalid_floor_reached();
    }
}

inline void dec(std::atomic<FloorNum>& a) {
    switch (a) {
        case F1: a = FG; return;
        case F2: a = F1; return;
        case F3: a = F2; return;
        default:         throw invalid_floor_reached();
    }
}

// inline void operator++ (FloorNum& a) {
    // switch (a) {
        // case FG: a = F1; return;
        // case F1: a = F2; return;
        // case F2: a = F3; return;
        // default:         throw invalid_floor_reached();
    // }
// }

// inline void operator++ (FloorNum& a, int) {
    // switch (a) {
        // case FG: a = F1; return;
        // case F1: a = F2; return;
        // case F2: a = F3; return;
        // default:         throw invalid_floor_reached();
    // }
// }

// inline void operator-- (FloorNum& a) {
    // switch (a) {
        // case F1: a = FG; return;
        // case F2: a = F1; return;
        // case F3: a = F2; return;
        // default:         throw invalid_floor_reached();
    // }
// }
```

```
70
71  // inline void operator-- (FloorNum& a, int) {
72      // switch (a) {
73          // case F1: a = FG; return;
74          // case F2: a = F1; return;
75          // case F3: a = F2; return;
76          // default:         throw invalid_floor_reached();
77      // }
78  // }
79
80  enum ElevState {
81      ES_WAIT = 0b00,
82      ES_DOWN = 0b01,
83      ES_UP = 0b10,
84      ES_DC = 0b11 // wont be used (dont care)
85  };
86
87  inline std::string pretty(ElevState s) {
88      switch (s) {
89          case ES_WAIT:
90              return "Waiting";
91          case ES_DOWN:
92              return "Down";
93          case ES_UP:
94              return "Up";
95          default:
96              return "Broken";
97      }
98  }
99
100 struct Elevator {
101     std::atomic<FloorNum> mFloor;
102     std::atomic<ElevState> mState;
103     Door mDoor;
104     std::thread mThread;
105
106     std::atomic_bool mStop;
107
108     Elevator();
109     ~Elevator();
110
111     void start();
112     void reset(FloorNum flr);
113
114 };
```

```cpp
#pragma once

#include <atomic>
#include <cstddef>
#include <thread>

class FloorLights {
    std::thread mThread;
    std::atomic_uchar X20;

public:
    FloorLights();
    ~FloorLights();

    unsigned char getLights();

    void start();

};
```

```cpp
#pragma once

#include <string>

enum IRState {
    IR_OFF = 0b00,
    IR_BUSY,
    IR_CLEAR,
    IR_DONE

};

inline std::string pretty(IRState s) {
    switch (s) {
        case IR_OFF:
            return "Sensor Off";
        case IR_BUSY:
            return "Person Walking";
        case IR_CLEAR:
            return "No Person Walking";
        case IR_DONE:
            return "Sensor Done";
        default:
            return "Laser weapons kill people";
    }
}

struct IR_Sensor {
    IRState mState;

    void start();
};
```

```cpp
#pragma once

#include <chrono>
#include <iostream>

using clk = std::chrono::system_clock; // I am not typeing all that over and over
using namespace std::chrono_literals;

#include "controller.hpp"
#include "elevator.hpp"
#include "clock.hpp"
#include "memory.hpp"
#include "door.hpp"
#include "floor_lights.hpp"
#include "ir_sensor.hpp"

extern std::atomic_bool FireKey;
extern std::atomic_bool IRon;
extern std::atomic_bool Sound;

extern std::atomic_bool gStop;
extern std::atomic_bool gStart;

extern Elevator gLift;
extern Controller gControl;
extern Memory gMem;
extern Clock gClk;
extern FloorLights gFL;
```

```cpp
#pragma once

#include <iostream>
#include <queue>
#include <mutex>

///// NOT THREAD SAFE /////
// actually doesnt need to be, only thing accessing the memory is the
// controller and the inputs
// f*ck it Im using a mutex, to lazy to think this through

class Memory {
    std::queue<FloorNum> mFloors;
    std::mutex mGuard;

    FloorNum mFireFloor;

public:
    FloorNum getFloor();
    FloorNum getDefaultFloor();
    void clearMem();
    void setFloor(FloorNum flr);
    bool isEmpty();

};
```