

1. **实验目的**

本实验要求模拟对于一个40层楼、有十部电梯的建筑，电梯的运行情况。乘客在运行期间随机产生在一楼，并有乘梯请求，10部电梯独立运行。设计实现显示仿真信息的界面。整个实验分为五个部分，要求分步逐次完成要求。

1. **实验环境**
2. **软件环境**

IDE:Clion 2022.1.3

编译工具：Cmake 3.23.2

编译器版本: MinGW w64 9.0

GDB version: 11.1

1. **硬件环境**

处理器 Intel(R) Core(TM) i7-10875H CPU @ 2.30GHz 2.30 GHz

机带 RAM 16.0 GB

系统类型 64 位操作系统, 基于 x64 的处理器

1. **实验内容**
2. 实验1要求

设计出电梯类、乘客类，设计显示仿真信息的界面，显示必要的信息。随机产生乘客要到达的楼层，乘客只产生一次乘梯请求。所有的乘客都结束到达请求的楼层后仿真结束。

1. 实验2要求

每位乘客产生L次请求，每到达一次目的楼层停留一段时间，最后一次请求下降到一楼并结束乘梯行为，所有乘客结束乘梯后仿真结束。

1. 实验3要求

修改乘梯规则，不同电梯所能够到达的楼层不同。

1. 实验4要求

仿真结束后，对电梯在仿真过程中的运行情况进行统计分析。

1. 实验5要求

根据对电梯仿真运行的统计分析结果对乘客进行提前预警或推荐电梯较为空闲的时间段。

1. **实验步骤**
2. 设计乘客Passenger类

乘客随着模拟过程的进行状态会变化，我给乘客设计了5种状态：

// 表示乘客状态的枚举类型  
typedef enum {  
 *BeforeSimulation* = 0, // 表示该乘客还未进入大楼，仿真还未开始  
 *StayForRandomTime* = 1, // 表示乘客在某层随机停留  
 *WaitForElevatorArrive* = 2, // 表示乘客在等待电梯到达  
 *RunWithElevator* = 3, // 表示跟随电梯运行  
 *AfterSimulation* = 5, // 仿真结束  
}PassengerState;

针对不同的状态，乘客在每次模拟过程中会更新自身的信息。这里给出三种模拟的接口与实现，根据当前状态选择模拟过程的接口为：

void Passenger::p\_Simulation(){}

// 乘客在某层停留时的模拟

// 如果结束了停留，则将状态改变为等待电梯，并为乘客重新分配目的地  
void Passenger::staySimulation() {}  
// 乘客等待电梯时的模拟，乘客在当前楼层的等待时间自增  
void Passenger::waitSimulation() {}  
// 乘客跟随电梯运动时的模拟，实时更新乘客的楼层信息

// 保持乘客与当前所乘坐的电梯的楼层相同。  
void Passenger::runSimulation() {}

此外，为了模拟乘客的运行，Passenger类中还有一些别的成员变量：

int destination; // 乘客的目的楼层

int currFloor; // 乘客的当前所在的楼层

int id; // 乘客的ID

int evtID; // 为乘客分配的电梯的ID

乘客在模拟过程中应该有自己的编号，有自己的目的地，有当前所在的楼层，有载乘客的电梯的编号。

由于乘客在在模拟过程中会产生多次乘梯请求，所以passenger类还需要有成员变量表示乘客总的模拟次数与当当前的模拟次数。当乘客的当前模拟次数达到总模拟次数后，乘客产生一个到达一楼的请求，到达一楼后乘客结束模拟过程。

int currSmuTimes; // 乘客当前的仿真次数

int maxSmuTimes; // 乘客的总仿真次数

1. 设计电梯Elevator类

电梯的功用为接送乘客，所以电梯会有自己的状态、运行的方向，在两层之间运行所需要的时间、当前所处的楼层、编号、目的楼层、乘客、对电梯发起请求的楼层。因此Elevator类中会有以下的变量：

ElevatorState state; // 电梯的状态  
Direction direction; // 电梯的方向  
int passTime; // 电梯在两层之间已经运行的时间  
int currFloor; // 电梯当前的楼层  
int id; // 电梯的编号  
int destination; // 电梯的目的地  
vector<Passenger\*> passengers; // 电梯中的乘客数

vector<int> requests; // 电梯接收到的请求的楼层

与乘客的状态相似，电梯的状态也是枚举类型：

// 电梯状态

typedef enum {  
 Stop = 0, // 停止  
 GoUp = 1, // 向上  
 GoDown = 2, // 向下  
 Stay = 3, // 停留  
}ElevatorState;

当电梯中没有乘客且没有楼层对电梯发起请求时，电梯处于停止状态、乘客运行时会有向上、向下、停留在某层三种状态、与对乘客的模拟类似，在时钟的激励下，电梯调用接口e\_simulation()；根据自己的状态选择要模拟的过程。模拟的接口如下：

void goUpSimulation(); //电梯向上运行时的模拟

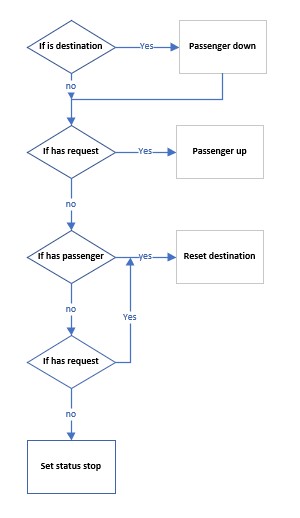
void goDownSimulation(); // 电梯向下运行时的模拟

图示

描述已自动生成具体的算法思路为如果电梯当前位于两层之间之间，则电梯继续运行。如果电梯到达了新的楼层，更新电梯中所有乘客的位置，判断该楼层是否是搭载的某位乘客的目的地，若有乘客到达了目的地或有乘客对电梯发起请求，则将电梯的状态置为停留。向上与向下运行时的模拟类似，算法流程图如下：

void StaySimulation(); // 电梯停留在某层时的模拟

当电梯停留在某层时，要判断是否有乘客在该层下电梯，若有乘客要下电梯，则将该乘客从电梯中转移到该楼层中；判断是否有乘客要上电梯，若有乘客要上电梯，则将该乘客从楼层转移到电梯中。处理完乘客后，检查是否仍有乘客，若有乘客，则根据乘客的目的地重新选择目的地、若无乘客，判断是否有请求，有请求则前往请求地点、无乘客无请求则将电梯的状态置为停止。算法流程图如下：



由于不同的电梯能够到达的楼层不同，电梯类中还有一个静态成员变量：

static bool arrivedTable[10][40]; // 电梯的可达表

初始化电梯的可达表，行代表电梯列代表楼层，电梯能够判断某个楼层是否可达。在接送乘客时，电梯要判断该乘客所在的楼层与乘客所要前往的楼层自己是否能够到达，只有当二者电梯均能到达时，电梯才能接收这个请求：

If (destination is arrivable && currfloor is arrivable)

elevator get request;

在每轮模拟中，电梯会记录自身的状态，若为运行状态，则对运行状态计时的变量加一，反之则对记录空闲状态的变量加一，作为最后展示的统计数据。

1. 设计建筑Building类

所有的电梯和乘客都在建筑中，因此Building类有两个对象数组，一个用来储存电梯，一个用来储存乘客：

vector<Elevator\*> Elevators;  
vector<Passenger\*> Passengers;

在时钟的激励下，每轮模拟中，建筑调用模拟接口：

void b\_simulation();

首先模拟所有的电梯，更新电梯的状态，即对对象数组中的每个电梯，调用其接口

elevator->e\_Simulation(); // 更新所有电梯的状态

然后，对乘客调用其模拟接口

passenger->p\_Simulation(); // 更新乘客的状态

在更新乘客的状态后，根据乘客的当前状态，若乘客已经结束了停留过程，则为其分配电梯，并将其状态设置为等待乘客的到来。

如果乘客的电梯已经到达，则将乘客加入电梯中，将其状态设置为跟随电梯运行。

在对乘客和电梯分别进行模拟后，Building调用showData()接口，展示模拟信息的面板，面板内容每个电梯的当前所在楼层、当前运行状态、当前搭载的乘客。

在每轮模拟结束后，程序会统计本轮模拟过程中运行中的电梯的数目，并存入向量

vector<unsigned short> runningElevators; // 记录每一时间下正在运行的电梯数目

之中，作为判断当前时段是否为繁忙时段的一个指标。

1. 程序的运行逻辑简介

在主函数中，创建两个实例building与timer，在时钟的驱使下，building检查是否仍有乘客未结束仿真过程，若有，则仿真继续进行，若没有，则仿真结束。

时钟timer在每轮模拟中计时加一，building调用接口b\_simulation，依次对所有的电梯和乘客进行模拟，直至模拟过程结束，程序退出。

在模拟过程中记录个各个电梯和乘客的部分运行信息，在模拟过程结束后，building调用接口给showStatistics()，展示出这些信息。

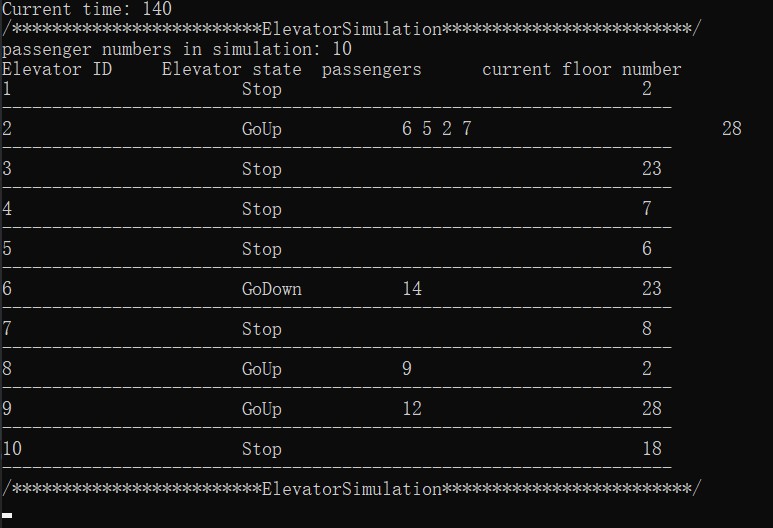
展示完信息后，即可释放掉building和timer的内存。程序正式结束。

1. **实验结果**

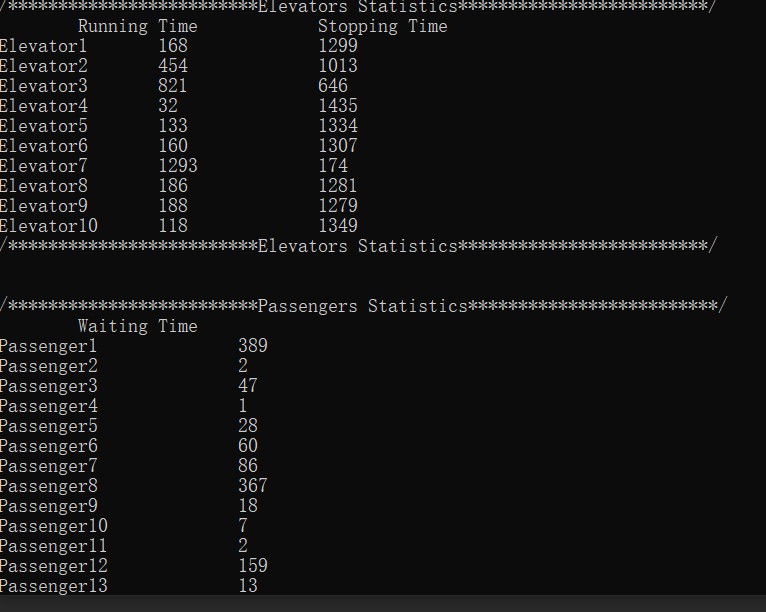
本实验做的是模拟程序，可以更改进入模拟过程的的总人数来进行测试，下面分别对最大模拟人数为20人、60人、100人三种情况进行测试。

1. 最大模拟人数为20人

模拟过程：



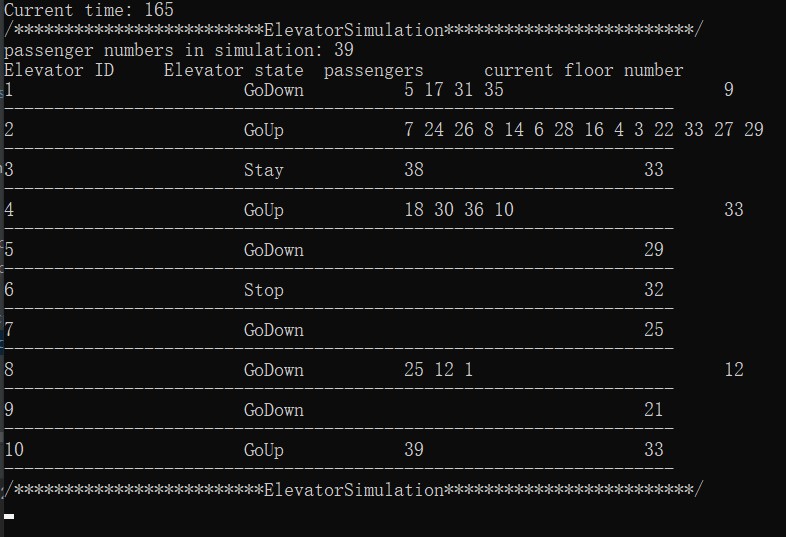
统计信息：



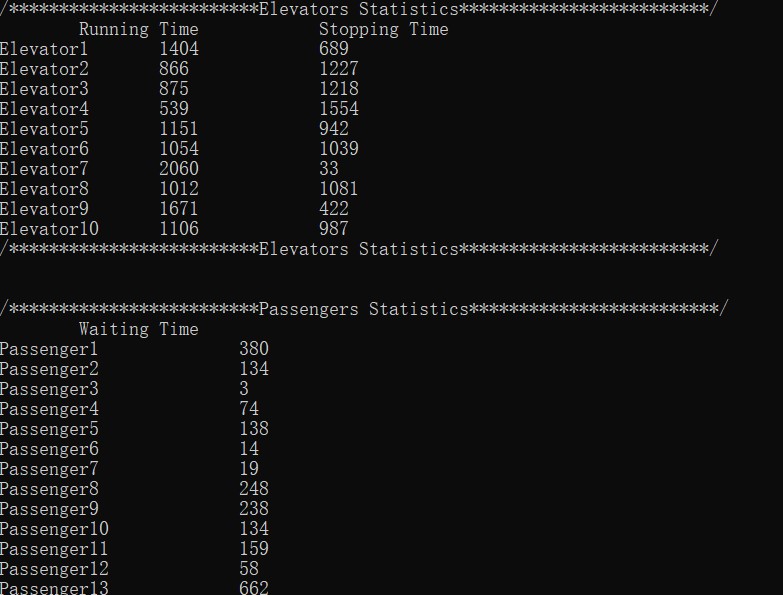
本次模拟产生了13名乘客，模拟过程持续了1349个时钟，可以看到，最繁忙的的电梯为编号为10的电梯，他是最后一个结束模拟的电梯。程序运行情况正常。

1. 最大模拟人数为60人

模拟过程：



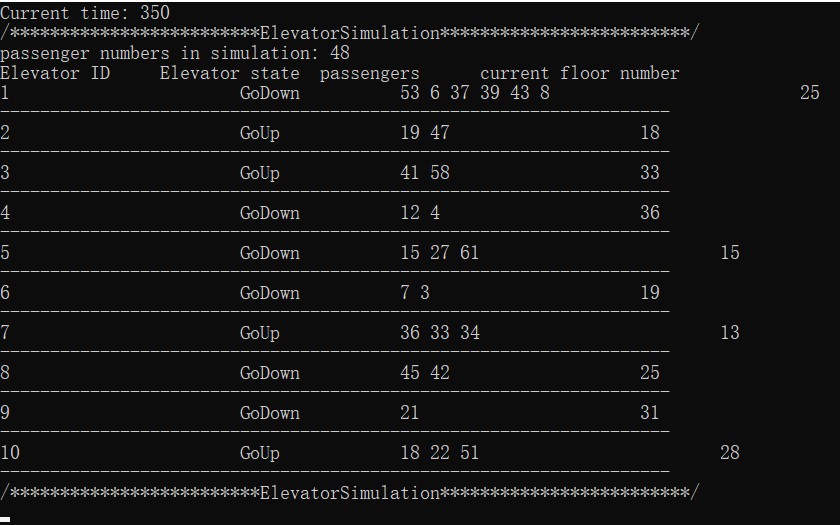
统计分析：



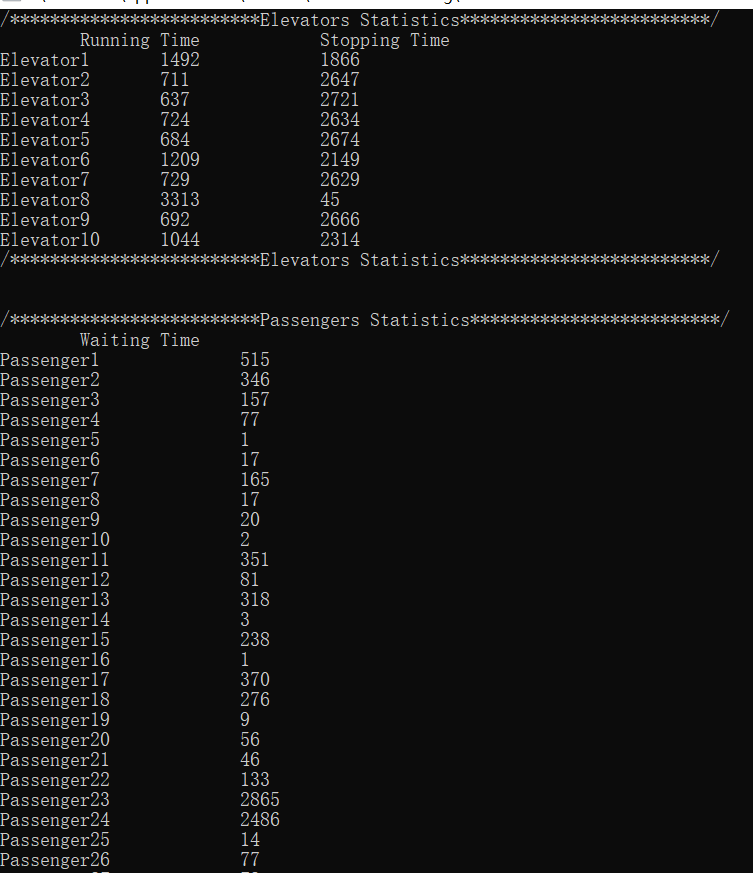
本次模拟生成了39名乘客，模拟过程持续了2203个时钟，运行时间最长的电梯为编号为9的电梯，运行时间持续了1671个时钟，停止时间最长的电梯为编号为2的电梯，有1227个时钟的时间都处于停止状态。

1. 最大模拟人数为100人

运行过程：



统计分析：



本次模拟过程生成了72名乘客，模拟过程持续了3349个时钟，运行时间最长的电梯为编号为8的电梯，运行时间持续了3313个时钟，停止时间最长的电梯为编号为3的电梯，有2721个时钟的时间都处于停止状态。

1. **实验总结**

在本次实验中，我根据实验要求设计了一个对40层楼高、拥有十台电梯的建筑运行过程的模拟。

该模拟程序在时钟的驱使下，根据乘客和电梯的状态对其进行模拟，并更新及记录其状态。每个乘客产生多次乘梯的需求，乘客发起请求，系统派遣合适的电梯接送乘客，乘客的最后一次请求一定是前往一楼，在电梯将其送往一楼后乘客结束模拟过程，当所有乘客都结束模拟后整个模拟过程结束。

通过本次实验，我应用了C++的面向对象思想设计了Building、Elevetor、Passenger三个类，按照封装继承多态的思想设计三个类之间的交互关系，以实现实验要求的模拟过程。

在实验过程中，我充分应用了在课堂上学习到的知识，使用了了C++的标准库中提供的许多数据结构与算法，如std::vector，std::queue；我在类设计过程中考虑了资源释放的问题，给每个类都编写了合适的析构函数；针对timer类，我重载了++运算符，使之能够增长计时器记录的时间；我应用了设计模式中的部分思想，如单例模式(计时器timer)是一个单例对象，观察者模式。

这是我第一次编写代码行数接近1000行的程序，这次实验挑战大，收货也打，通过这次实验，我对C++的OOP思想有了更深刻的理解。

**附录 源程序清单（要求注释不少于源代码行数的三分之一）**

**附录1 passenger类的设计与部分重要源码**

class Passenger {

private:

    static vector<std::pair<int, int>> busytime;    // 电梯运行的高峰期

    int waitTime;   // 乘客发起请求后等待电梯的时间

    int destination;    // 乘客的目的楼层

    int currFloor;     // 乘客的当前所在的楼层

    int id;             // 乘客的ID

    int evtID;          // 为乘客分配的电梯的ID

    PassengerState state;   // 乘客在模拟过程中的状态

    int stayingTime;    // 乘客在某层的停留时间

    int currSmuTimes;   // 乘客当前的仿真次数

    int maxSmuTimes;    // 乘客的总仿真次数

public:

    void p\_Simulation();    // 乘客的模拟过程

    Passenger(int \_id); // 乘客的构造函数

    ~Passenger();

    void staySimulation();

    void waitSimulation();

    void runSimulation();

    PassengerState getState();

};

// Passenger的构造函数

Passenger::Passenger(int \_id) : currFloor(1), currSmuTimes(0), state(BeforeSimulation), id(\_id), waitTime(0) {

    maxSmuTimes = rand() % maxSimulationTimes + 1;  //  每位乘客的仿真次数在 1-10次之间

    // 为乘客随机分配电梯

    evtID = rand() % evtNums + 1;

    destination = rand() % floorNums + 1;

}

// 乘客的模拟过程

void Passenger::p\_Simulation() {

    // 根据乘客现在不同的状态进行不同的模拟

    switch (state) {

            // 如果状态为开始模拟之前的话， 将状态转换为等待电梯到来

        case BeforeSimulation:

            state = WaitForElevatorArrive;

            break;

            //如果状态为随机停留的话，进行随机停留模拟

        case StayForRandomTime:

            staySimulation();

            break;

            // 进行等待电梯的模拟

        case WaitForElevatorArrive:

            waitSimulation();

            break;

            // 进行跟随电梯运行的模拟

        case RunWithElevator:

            runSimulation();

            break;

        case AfterSimulation:

            break;

        default:

            cout << "Unexpect Error." << endl;

            break;

    }

}

// 乘客在某层停留时的模拟

void Passenger::staySimulation() {

    // 如果结束了停留，则将状态改变为等待电梯，并为乘客重新分配目的地

    int endTime = timer->getTime() + stayingTime;

    if (inBusyTime(endTime)) {

        stayingTime += 20;

    }

    if (stayingTime > 0) {

        stayingTime--;

    }

}

// 乘客等待电梯时的模拟

void Passenger::waitSimulation() {

    waitTime++; //  乘客等待电梯时模拟时间自增

}

// 乘客跟随电梯运动时的模拟，实时更新乘客的楼层信息

void Passenger::runSimulation() {

    Elevator\* elevator = elevators[evtID-1];    // 乘客当前乘坐的电梯

    this->currFloor = elevator->getCurrFloor(); //  乘客所处的楼层与其乘坐的电梯所处的楼层相同

    ;

}

**附录2 Elevator类的设计与部分源码实现**

class Elevator {

private:

    int stopTime;   // 电梯在模拟过程中的总停留时间

    int runTime;    // 电梯在模拟过程中的总运行时间

    static int gapTime;

    ElevatorState state;    // 电梯的状态

    Direction direction;    // 电梯的方向

    int passTime;   // 电梯在两层之间已经运行的时间

    int currFloor;          // 电梯当前的楼层

    int id;                 // 电梯的编号

    int destination;    // 电梯的目的地

    vector<Passenger\*> passengers;  // 电梯中的乘客数量

    vector<int> requests;        // 电梯接收到的请求的楼层

public:

    static bool arrivedTable[10][40];    // 电梯的可达表

    Elevator(int \_id);                     // 电梯的构造函数

    ~Elevator();

    void e\_Simulation();    // 电梯的模拟过程

    void addPsg(Passenger \*pPassenger); // 添加新的乘客

    void goUpSimulation();

    void goDownSimulation();

    void StaySimulation();

    void StopSimulation();

};

**// 电梯的构造函数**

Elevator::Elevator(int \_id) : id(\_id), currFloor(1), direction(DirectionStay), state(Stop),

                              passTime(0), destination(1),

                              runTime(0), stopTime(0){

    // 一楼对电梯有请求

    requests.push\_back(1);

}

// 电梯的模拟过程，根据电梯的状态模拟

void Elevator::e\_Simulation() {

    switch (state) {

        case GoUp:

            goUpSimulation();

            break;

        case GoDown:

            goDownSimulation();

            break;

        case Stop:

            StopSimulation();

            break;

        case Stay:

            StaySimulation();

            break;

        default:

            cout << "Unexpected Error." << endl;

    }

}

// 电梯处于向上状态时的模拟

void Elevator::goUpSimulation() {

    runTime++;

    if (this->passengers.size() == 0 && this->requests.size() == 0) {

        state = Stop;

    }

    // 如果电梯在两层之间已经运行的时间小于需要的时间

    if (passTime < gapTime) {

        passTime++; //继续运行

    } else{

        currFloor++;

        passTime = 0;

        // 如果到达了某位乘客的目的地或者该楼层中有人发起请求，则停下来

        if (existReq() || isDestination()) {

            this->state = Stay;

        }

    }

    // 更新所有乘客的当前所在楼层信息

    for (auto passenger: passengers) {

        passenger->setCurrFloor(currFloor);

    }

    if (currFloor == 40 && state == GoUp) {

        state = GoDown;

    }

}

void Elevator::goDownSimulation() {

    runTime++;

    if (this->passengers.size() == 0 && this->requests.size() == 0) {

        state = Stop;

    }

    // 如果电梯在两层之间已经运行的时间小于需要的时间

    if (passTime < gapTime) {

        passTime++; //继续运行

    }

        // 等于

    else {

        currFloor--;

        passTime = 0;

        // 如果到达了某位乘客的目的地或者该楼层中有人发起请求，则停下来

        if (existReq() || isDestination()) {

            this->state = Stay;

        }

    }

    // 更新所有乘客的当前所在楼层信息

    for (auto passenger: passengers) {

        passenger->setCurrFloor(currFloor);

    }

    if (currFloor == 1 && state == GoDown) {

        state = GoUp;

    }

}

// 电梯在某层停留时的模拟

void Elevator::StaySimulation() {

    runTime++;

    // 如果有乘客到达了目的地

    for (auto iterator = passengers.begin(); iterator != passengers.end(); iterator++) {

        Passenger \*passenger = \*iterator;

        if (passenger->getDestination() == currFloor) {

//            floor.addPsgtoSC(passenger);    // 将该乘客加入该楼层中

            passengers.erase(iterator); // 将该乘客从电梯中删除

            passenger->setState(StayForRandomTime); // 重新设置乘客的状态

            if (passenger->getCurrSmuTimes() > passenger->getMaxSmuTimes()) {

                passenger->setState(AfterSimulation);

            }

            iterator--;

        }

    }

    // 如果该楼层中有人发起请求，删除请求

    for (auto iterator = requests.begin(); iterator != requests.end(); iterator++) {

        if (\*iterator == currFloor) {

            requests.erase(iterator);

            break;

        }

    }

    // 停留之后，根据电梯是否到达了目的地调整电梯的状态

    if (currFloor == destination) {

        //检查乘客是否为空

        // 所有乘客均已经到达目的地电梯设置为停止

        if (passengers.empty()) {

            state = Stop;

            direction = DirectionStay;

        }

            // 电梯中仍然有乘客，根据乘客的目的地选择新的目的地

        else {

            if (passengers[passengers.size()-1]->getDestination() > currFloor) {

                destination = passengers[passengers.size()-1]->getDestination();

                state = GoUp;

                direction = DirectionUp;

            }

            if (passengers[passengers.size()-1]->getDestination() < currFloor) {

                destination = passengers[passengers.size()-1]->getDestination();

                state = GoDown;

                direction = DirectionDown;

            }

        }

    }

        //未到达目的地，不改变方向，继续前进

    else if (destination > currFloor) state = GoUp;

    else state = GoDown;

}

// 电梯停止运行时的模拟

void Elevator::StopSimulation() {

    stopTime++;

    passTime = 0;

    // 检查是否有请求，有请求则前往请求地点

    if (!requests.empty()) {

        if (requests[requests.size()-1] == currFloor) {

            state = Stay;

            direction = DirectionStay;

        } else if (requests[requests.size()-1] > currFloor) {

            state = GoUp;

            direction = DirectionUp;

        } else {

            state = GoDown;

            direction = DirectionDown;

        }

    }

}

**附录3 Building类设计与部分源码**

class Building {

private:

    int psgNums;    //模拟过程中生成的总乘客数量

    vector<Elevator\*> Elevators;

    vector<Passenger\*> Passengers;

    vector<unsigned short> runningElevators;   // 记录每一时间下正在运行的电梯数目

public:

    Building();

    ~Building();

    void b\_simulation();

    int genePsgs();

    void showData();

    bool allocateElevatorForPassenger(Passenger \*passenger);

    int getPassengerNums();

    static bool isSatisfied(Elevator \*elevator, Passenger \*passenger);

    static void printState(ElevatorState state);

    static void printPassengers(Elevator\* elevator);

    void showStatistics() const;

    vector<unsigned short > getStatistics();

    vector<Passenger\*> getPsgs() const;

};

//building的构造函数

Building::Building() {

    // 逐一生成编号为 0 - evtNums - 1 的所有电梯

    for (int index = 0; index < evtNums; index++) {

        auto elevator = new Elevator(index + 1);

        Elevators.push\_back(elevator);

    }

    int psgNums = genePsgs(); // 生成乘客

}

// 生成一定数目的乘客

// 当前情况为： 乘客一次性到达建筑的第一层，并为其分配电梯

int Building::genePsgs() {

    // 生成随机数目的乘客

    int psgNums = rand() % maxPsgNums + 1;

    cout << "generate " << " " << psgNums << " passengers successfully" << endl;

    // 生成编号为 1 - maxPsgNums 的乘客

    for (int i = 0; i < psgNums; i++) {

        auto \*passenger = new Passenger(i + 1);

        Passengers.push\_back(passenger);

    }

    return psgNums;

}

// 建筑的模拟过程

void Building::b\_simulation() {

    // 对所有elevator都模拟

    for (auto elevator: Elevators) {

        elevator->e\_Simulation();   // 更新所有电梯的状态

    }

    // 对所有的乘客进行模拟

    for (auto iterator = Passengers.begin(); iterator != Passengers.end(); iterator++) {

        Passenger \*passenger = \*iterator;

        passenger->p\_Simulation();  // 更新乘客的状态

        PassengerState psgstate = passenger->getState();    //获取到乘客的状态

        int stayingTime = passenger->getStayingTime();    // 获取乘客的停留时间

        int evtID = passenger->getEvtID();  //获取乘客希望乘坐的电梯的ID

        int evtCurrFloor = Elevators[evtID-1]->getCurrFloor();  // 获取到电梯的当前楼层

        int psgCurrFloor = passenger->getcurrFloor();   //获取乘客的当前楼层

        // 乘客已经结束停留

        if (psgstate == StayForRandomTime && stayingTime == 0) {

            if (allocateElevatorForPassenger(passenger))  // 为乘客分配合适的电梯，如果成功分配，则重新设置乘客的状态

                passenger->setState(WaitForElevatorArrive); // 重新设置乘客的状态

        }

        // 如果乘客的电梯已经到达，将乘客加入电梯内

        else if (psgstate == WaitForElevatorArrive && evtCurrFloor == psgCurrFloor) {

            passenger->setState(RunWithElevator);   // 重新设置乘客的状态

            Elevators[evtID - 1]->addPsg(passenger);  // 将乘客加入到电梯中

        }

        // 如果乘客的状态为已经结束模拟

        else if (psgstate == AfterSimulation) {

            ;

        }

    }

    // 展示模拟信息面板

    showData();

    // 统计正在运行的电梯数目;

    unsigned short amount = std::count\_if(Elevators.begin(), Elevators.end(),[](const Elevator\* elevator){return elevator->getState() != Stop;});

    runningElevators.push\_back(amount);

}

// 为乘客分配合适的电梯， 并向电梯发送请求

bool Building::allocateElevatorForPassenger(Passenger \*passenger) {

    //  遍历10部电梯，寻找适合的电梯

    /\*

    \* 有三种情况的电梯符合要求:

    \* 1. 电梯的当前位置低于本楼层，电梯的目的地高于本楼层

    \* 2. 电梯的当前位置高于本楼层，电梯的目的地低于本楼层

    \* 3. 电梯为停止状态

    \*/

    // 如果存在合适的电梯，则将该电梯的ID传给乘客

    vector<Elevator\*> qualified;

    for (Elevator \*elevator: Elevators) {

        if (isSatisfied(elevator, passenger)) {

            qualified.push\_back(elevator);

        }

    }

    if (qualified.size() != 0) {

        std::sort(qualified.begin(), qualified.end(),

                  [passenger](Elevator \*ele1, Elevator \*ele2) {

                      return ele1->distance(passenger) < ele2->distance(passenger);

                  });

        Elevator\* elevator = qualified[0];

        passenger->setElevator(elevator->getID());

        int floorNumber = passenger->getcurrFloor();

        elevator->setRequest(floorNumber);   // 向指定的电梯发出请求

        return true;

    }

    return false;

}

// 判断电梯是否满足

// 上机要求三：不同的电梯所能到达的楼层不同

bool Building::isSatisfied(Elevator \*elevator, Passenger \*passenger) {

    int evtID = elevator->getID();  //  电梯的ID

    int evtFloor = elevator->getCurrFloor();    // 电梯的楼层

    int psgFloor = passenger->getcurrFloor();   // 乘客的楼层

    int evtDestination = elevator->getDestination();    //电梯的目的地

    int psgDestiantion = passenger->getDestination();   // 乘客的目的地

    // 如果乘客的目的地电梯能够到达

    if (Elevator::arrivedTable[evtID - 1][psgDestiantion - 1]) {

        if (evtFloor <= psgFloor && evtDestination >= psgDestiantion)

            return true;

        if (evtFloor >= psgFloor && evtDestination <= psgDestiantion)

            return true;

        if (elevator->getstate() == Stop)

            return true;

    }

    return false;

}

**附录4 程序的主过程**

nt main() {

    // 设置随机数

    srand(time(0));

    //生成建筑、电梯和乘客

    auto\* building = new Building();

    systemRun(building);

    return 0;

}

// 程序的主过程

void systemRun(Building\* building) {

    Timer\* timer = Timer::getInstance();

    vector<Passenger\*> passengers = building->getPsgs();

    while (std::any\_of(passengers.begin(), passengers.end(), [](Passenger \*p){ return p->getState() != AfterSimulation;})) {

        timer->showCurrTime();

        building->b\_simulation();

        system("cls");

        timer->increaseTime();

    }

    vector<unsigned short > statistics = building->getStatistics();

    building->showStatistics();

    system("pause");

}