实验二——电梯模拟

PB17000002

古宜民

2018.11.17

1. 实验基本要求

设计一个基于离散事件模拟的电梯模拟系统，使用模拟时钟决定每个活动体（电梯、乘客等）的动作发生的时间和顺序。

2. 实验实现内容

实现多电梯模拟和调度。

完整代码见：

https://github.com/ustcpetergu/Datastructure/tree/master/Course2018Spring/Elevator

（1）模拟 N 层楼 M 个电梯的活动情况，N 和 M 可变。初始时刻所有电梯停在一楼处于空闲状态并等待请求。

（2）乘客可以在模拟时间上限内的任何时间进出大楼的任一层并去往任一层，每个乘客有一个最长等待时间，如果等待时间超过最长等待时间，则乘客放弃并离去。

（3）模拟的假设条件：每个人进出电梯均要花费一定时间。人进出后电梯门保持开一段时间后才关闭。电梯开、关门均要花费时间。假设电梯匀速运动。电梯有最大载客量限制并可变。电梯在没有任务时会保持停在当前层，不会回到自己的“本垒层”。

3. 实验代码结构和关键代码讲述

使用 C++语言实现，基于回调进行事件处理，由 Linux 终端绘制字符界面。

每个乘客的来往楼层和最大等待时间从文件一次性读入。

（1）程序结构设计

数据结构概括：在主函数中由 EventCase 和 EventList 类进行事件处理；电梯系统

ElevatorSystem 类和单个电梯 Elevator 类与大楼 Building 类共同实现整个系统并向 EventList 不断发送请求；来往的乘客由 Person 类实现并记录。乘客到达某层会按下该层向上或向下的按钮，登上电梯后再按下电梯内要去楼层对应的按钮，实现与电梯系统的交互。

数据结构详细介绍：

Person 类：完成乘客的操作，包含乘客的来、去楼层、最大时间、是否已经离开等变量。

EventCase 类：从电梯等部分发送回主函数处理的内容使用 EventCase 类传递，其包含了时间发生时间等变量以及事件发出者的类型和事件名称，并且包含了一个 void* 型指针，用来指向事件的发出者，该指针在主函数中被转换成类指针。该类按事件发生先后定义了 operator<算符。

EventList 类：继承自 Queue<EventCase>类，重载了和时间有关的操作。Queue 类为队列模板类，只要定义了 operator<的元素就可以使用。Queue 类继承自之前写过的 LinkList 模板类，体现了模块化程序设计。

```
class EventList: public Queue<EventCase> {
        public:
                EventList(){
                        curtime = 0;
                }
                EventList(Time t_init){
                        curtime = t_init;
                }
                Status EnqEvent(EventCase& e){
                        e.occurtime = curtime + e.delaytime;
                        return this->Enqueue(e);
                }
                Status DeqEvent(EventCase& e){
                        if(ListEmpty())
                                return ERROR;
                        this->Dequeue(e);
                        return OK;
                }
                Status ListEmpty(){
                        return this->QueueEmpty();
                }
                void FastForward(Time t)
                {
                        this->curtime = t;
                }
                Time GetTime()
                {
                        return this->curtime;
                }
        private:
                Time curtime;
};
```

Building 类：包含了 N 层的 N 个链表存储每个楼层在等待的人，每个链表中元素为 Person*类型。

ElevatorSystem 类：电梯系统，存储了 M 个电梯的指针，Building 的指针，EventList 的指针，每层对电梯的分配 upassigns 和 downassigns，和每层的按钮 Button 类数组，每个 Button 对象含有向上、向下两个按钮（bool 型）。定义了 PressButton 函数，即每个乘客来到电梯前按下按钮；以及 AssignElevator 函数用于分配电梯，在下文调度算法中提到。

Elevator 类：单个电梯，含有指向 ElevatorSystem 的 father 指针，梯上乘客链表，当前楼层数，以及 FloorButton 类，为电梯轿厢内的 N 层楼 N 个按钮。另外还有电梯本身的配置和状态信息，如最大人数、门是否开着等。包含程序核心函数：

PressFloorButton：乘客进入电梯后按下电梯内的按钮。仅仅把按钮状态变为开，其他处理工作由其他函数进行。

DoorOpened：电梯开门。开门后，首先到达该层的人依次下电梯，之后在该层该

方向等待的人依次上电梯，如果人已经超时离开，就把人从等待队列中删除。最后，发送 DoorClosed 这个 Event 到主函数，表示等一段时间后电梯关门。为了实现“依次”上下的功能，该函数“递归”实现，即每次上或下一个人，然后再发送一个 DoorOpened 事件，之后直接函数结束，由主回调过程再次调用本函数。

上人：

```
void Elevator::DoorOpened()
{
        this->isdooropened = true;
        Person* p;
        EventCase e;
        //dump persons
        //judge first to increase some time performance
        if(this->floorbutton.isPressed(atfloor)){
                int l = personsonboard.GetLength();
                for(int i = 1; i <= l; i++){
                        personsonboard.GetElem(i, p);
                        if(p->gotofloor == atfloor){
                                this->personnum--;
                                p->timegetoff = father->eventlist->GetTime();
                                e = EventCase(PersonGetOffTime, "Person", "GetOff", p);
                                father->eventlist->EnqEvent(e);
                                e = EventCase(PersonGetOffTime, "Elevator", "DoorOpened", this);
                                father->eventlist->EnqEvent(e);
                                //note that the delete caused length to change, but then
                                //the func returns, so no bug
                                personsonboard.ListDelete(i, p);
                                //Handle one by one
                                return;
                        }
                }
        }
        //clear floor button, this is executed only after all persons got off
        this->floorbutton.Clear(atfloor);
```

下人：

```
    //collect persons
    //automatic stop collecting and go on when elev is full or all collected
    int l = father->building->persons[atfloor].GetLength();
    //first remove all unpatiently left persons
    for(int i = l; i >= 1; i--){
            father->building->persons[atfloor].GetElem(i, p);
            if(p->istimedout)
                    father->building->persons[atfloor].ListDelete(i, p);
    }
    //re-calc length
    l = father->building->persons[atfloor].GetLength();
    for(int i = 1; i <= l && !this->isFull(); i++){
            father->building->persons[atfloor].GetElem(i, p);
            //the person has already left? And only collect people on this dir
            if(p->dir == this->dir){
                    //if(!p->istimedout){
                    this->personnum++;
                    this->personsonboard.Append(p);
                    this->PressFloorButton(p->gotofloor);
                    p->timegeton = father->eventlist->GetTime();
                    e = EventCase(PersonGetOnTime, "Person", "GetOn", p);
                    father->eventlist->EnqEvent(e);
                    //}
                    father->building->persons[atfloor].ListDelete(i, p);
                    e = EventCase(PersonGetOnTime, "Elevator", "DoorOpened", this);
                    father->eventlist->EnqEvent(e);
                    return;
            }
    }
```

最后，人上下完，将该层该方向按钮熄灭。如果还有人没上，再次按按钮，这个过程在 DoorClosed 中实现。具体见下文电梯调度算法。

DoorClosed：关门，清理按钮状况。然后调用 Activate，决定下一步动作。

```
//Door closed, continue moving or stay still
void Elevator::DoorClosed()
{
        this->isdooropened = false;
        //seems too simple to be right, but i think it's OK
        //And assigns must be cleared ASAP because many judgements depend on this
        if(this->dir == UP)
                father->upassigns[this->atfloor] = -1;
        if(this->dir == DOWN)
                father->downassigns[this->atfloor] = -1;
        Person* p = NULL;
        int l = father->building->persons[atfloor].GetLength();
        //Clear btn even if some not satisfied,
        //and then re-press btn if someone left, to let elev sys to reassign a elev.
        //Because a mere elev don't know whether all persons had got on without overload.
        father->buttons[atfloor].Clear(this->dir);
        for(int i = 1; i <= l; i++){
                father->building->persons[atfloor].GetElem(i, p);
                if(!p->istimedout && p->dir == this->dir){
                        father->PressButton(atfloor, this->dir);
                        //cout << father->downassigns[atfloor]<< endl;
                        //cout << "Re-pressed";cin.get();
                        break;
                }
        }
        this->Activate();
}
```

ArrivedNextFloor：到达了下一层。对电梯当前层变量 atfloor 进行++或--操作，之后调用 Activate，进行下一步判断。

Activate：单个电梯最重要的活动判断函数。只要知道了电梯目前的状态（方向、当前楼层数、是否满载、内部按钮情况）和电梯系统为其分配的任务（关于分配见下文调度算法），就能唯一确定电梯下一步该如何行动，是继续走，还是开门，或是换方向，或者停止并进入空闲状态。流程如下：

首先判断是否需要上、是否需要下。

如果恰好当前层需要开门上下人，那么：

如果正在上，而当前层要下，而正好没有继续上的任务，那么方向转为下

反之（下）亦然

如果空闲，就开始工作

发送事件 DoorOpened，函数返回

如果不需要上，也不需要下，就进入空闲。函数返回

如果正在上，而还需要上，那么继续上，不改变状态，发送 ArrivedNextFloor 事件，表示一段时间后到达下一层。函数返回

反之（下）亦然

如果正在空闲，那么开始工作，如果需要上，那么设方向为上，发送 ArrivedNextFloor，返回。需要下同理。

这段代码是逻辑最复杂、花最多时间调试的。代码如下：

```
//This is a single elev Main Ctrl Function
//Decide what to do next, considering all possible conditions
//Be called frequently
void Elevator::Activate()
{
     //Point: down and up assigns array is at same status with button[].Get, but
     //we assume that an elev should do it's own work without bothering others, but when
     //it comes that it just pass by other floor, it will also open the door if this floor need
     //to be served, even if the floor's job is not assigned to this elev.

     //First, have a judge of need to up or need to down
     bool isneedtoup = false;
     bool isneedtodown = false;
     for(int i = atfloor + 1; i <= FloorNumber; i++)
          //even if an upper floor need down, then the elev still need up, *sooner or later*
          if(father->downassigns[i] == index || father->upassigns[i] == index || floorbutton.isPressed(i)){
               isneedtoup = true;
               break;
          }
     for(int i = 1; i <= atfloor - 1; i++)
          if(father->upassigns[i] == index || father->downassigns[i] == index || floorbutton.isPressed(i)){
               isneedtodown = true;
               break;
          }
```

```
          //Should stop at this floor.
          //At this floor, first judge whether to change direction(important), then open door
and start working
          if((father->buttons[atfloor].Get(UP) && !(this->dir == DOWN && isneedtodown)
&& !isFull()) || \
                    (father->buttons[atfloor].Get(DOWN) && !(this->dir == UP && isneedtoup)
&& !isFull()) || \
                    floorbutton.isPressed(atfloor)){
               //It's time to change direction?
               //Change dir: now up, no need to up further, on this floor no UP pressed,
               //and DOWN MUST BE PRESSED, or means error HAD occured, because the elev
SHOULDN'T HAVE GONE to this floor
               //But, but, when executed here, downassigns (or upassigns) HAD BEEN Cleared!!
               if(this->dir == UP && !(father->buttons[atfloor].Get(UP)) && !isneedtoup){
                    this->dir = DOWN;
               }
               if(this->dir    ==    DOWN    &&    !(father->buttons[atfloor].Get(DOWN))
&& !isneedtodown){
                    this->dir = UP;
               }
               //start working from idle when a person precisely come to the floor the elev
resting at
               if(this->dir == NODIRECTION){
                    if(father->buttons[atfloor].Get(UP))
                         this->dir = UP;
                    if(father->buttons[atfloor].Get(DOWN))
                         this->dir = DOWN;
               }
               EventCase e = EventCase(this->timeDoorOpen, "Elevator", "DoorOpened", this);
               this->father->eventlist->EnqEvent(e);
               return;
          }
          //no work, be idle no matter was idle or busy
          //(no need to up, no need to down. executed here means no need to open door)
          if(!isneedtoup && !isneedtodown){
               this->dir = NODIRECTION;
               this->isactive = false;
               return;
          }
          //was moving and continue moving in old direction
          //the most simple case
          if(this->isactive){
               if((this->dir == UP && isneedtoup) || (this->dir == DOWN && isneedtodown)){
                    EventCase    e    =    EventCase(this->timeGotoNextFloor,    "Elevator",
```

```
"ArrivedNextFloor", this);
                    this->father->eventlist->EnqEvent(e);
                    return;
               }
          }
          //was idle and should start moving
          else{
               if(isneedtoup){
                    this->dir = UP;
                    this->isactive = true;
                    EventCase   e   =   EventCase(this->timeGotoNextFloor,   "Elevator",
"ArrivedNextFloor", this);
                    this->father->eventlist->EnqEvent(e);
                    return;
               }
               else if(isneedtodown){
                    this->dir = DOWN;
                    this->isactive = true;
                    EventCase   e   =   EventCase(this->timeGotoNextFloor,   "Elevator",
"ArrivedNextFloor", this);
                    this->father->eventlist->EnqEvent(e);
                    return;
               }
               else
                    Panic("wrong ctrl flow in Activate");
          }
          Panic("Nothing done in Activate. Another wrong ctrl flow in Activate");
     }
```

isFull：判断电梯是否已满

Button 和 FloorButton 类：电梯上的两种按钮，各自定义了 Get，Press 和 Clear 函数实现各自的功能。

主函数：主要用于回调，不断从 EventList 中获取事件并执行事件、推进时钟，并进行界面绘制。一段代码示例：

```
else if(e.obj == "Elevator"){
        if(e.cmd == "ArrivedNextFloor")
                static_cast<Elevator*>(e.ptr)->ArrivedNextFloor();
        else if(e.cmd == "DoorOpened"){
                static_cast<Elevator*>(e.ptr)->DoorOpened();
        }
        else if(e.cmd == "DoorClosed"){
                static_cast<Elevator*>(e.ptr)->DoorClosed();
        }
        else err_notify((empty + "No such event cmd " + e.cmd + " in obj " + e.obj).c_str());
}
else if(e.obj == "Person"){
        if(e.cmd == "Arrived"){
                Person* p = static_cast<Person*>(e.ptr);
                p->timearrival = elist.GetTime();
                building.persons[p->fromfloor].Append(p);
                es.PressButton(p->fromfloor, p->dir);
                //Enqueue the person leave angrily event, but if person is satisfied,
                //the event will just be ignored
                f = EventCase(p->maxwaittime, "Person", "Leave", p);
                elist.EnqEvent(f);
        }
        else if(e.cmd == "GetOn"){
                static_cast<Person*>(e.ptr)->timegeton = curtime;
        }
        else if(e.cmd == "GetOff"){
                static_cast<Person*>(e.ptr)->timegetoff = curtime;
                static_cast<Person*>(e.ptr)->timeleave = curtime;
        }
        else if(e.cmd == "Leave"){
                //mark timed out here, but keep person in queue
```

电梯调度算法：使用朴素的算法。当电梯系统里某层的某方向按钮被按下时，电梯系统会预测出当前状况下的 M 台电梯中以该方向运行到达该层所需时间最短并未达到满载的电梯，并将该层的该方向分配给该电梯，该电梯便能知道该层的该方向是“自己的任务”并对其做出反应，而其他电梯不会对其做出反应。但当某台电梯路过某层，该层有该方向不属于该电梯的需求时，该电梯也会停下并完成该层该方向的任务，完成后因为任务被取消，所以该任务原先被分配给的那一台电梯会继续完成其他任务，不会因此受影响。这样一定程度上可以提高效率。

预测采用的也是朴素的方法，预测路径并进而预测时间。比如某电梯在 n1 层向上，则如果它要接受 n2 层向上的任务，如果 n1>n2，则它要先上升到最高层，然后下降到最低层，再去往 n2 层。因为最低层层数未知，最后一步能是上升也可能是下降。而若 n2>n1，则直接上升到 n2 即可接到 n2 层要向上的人。而对时间的预测就是把电梯在预测路径上的活动所需时间计算出来，包括中途因有人在该层上或下停顿开关门等。注意为了保持与生活相似，电梯只能获得来自自己的按钮的信息，不能获得其他信息，比如

电梯只知道某层的按钮被按下，不能知道该层有几个人，人是否已经离开。

如果出现了电梯门关闭后电梯所在的该层又来了新乘客的情况，电梯门会再次打开，因为关门之后会调用 Activate 进行判断，判断结果是该层有人，于是再次调用 DoorOpened 方法，门再次打开。

有趣的是，在代码编写过程中，出现了一个现实生活中也存在的 bug：如果电梯在某层乘客登梯，但电梯满了还有人没有上来，电梯应该离开，而人会再次按本层的按钮（电梯关门后按钮应熄灭，因为电梯本身不知道外面是否还有人等或是都已经上完）。而在程序中，由于人会立刻重新按按钮，于是电梯门又开了。现实生活中人会达成共识，等一会再按，但在程序中这很难实现，于是本程序规定电梯只要满了，就不会再开门了，也就解决了这个问题。

另一个问题：怎么算电梯满？程序中电梯内部的 persons 链表存储了梯内人数，可以直接用 persons.length()<maxperson 判断，而本程序也是这么实现的。但现实中电梯是不知道其内有几个人的！但是，电梯内部的重力传感器可以得到当前载重量 now 和最大载重量 max，如果当|max-now|<Epsilon，Epsilon 大约为一个人体重（固定值）的时候，就可以认为电梯满了，这也是可行的，所以程序并没有破坏现实生活中的情况。

代码中，ElevatorSystem 的 upassigns[]和 downassigns[]存储了分配的电梯情况，AssignElevator 函数用于分配电梯，它调用了所以电梯的 Elevator::EvaluateTimeToFloor 并选出最小的一个。EvaluateTimeToFloor 函数比较繁杂，共 100 行左右对所有情况判断，片段如图。

```
Time Elevator::EvaluateTimeToFloor(int floor, Direction d) const
{
        if(!this->isactive)
                return this->timeGotoNextFloor * fabs(floor - this->atfloor);
        int maxfloor = this->atfloor;
        int minfloor = this->atfloor;
        int dooropentimes = 0;
        //up to top, then down to bottom, and then (up or down without stop) to fetch new person
        if(this->dir == UP && d == UP && atfloor >= floor){
                for(int i = atfloor; i <= FloorNumber; i++)
                        if(father->buttons[i].Get(UP) || this->floorbutton.isPressed(i)){
                                maxfloor = i;
                                dooropentimes++;
                        }
                for(int i = maxfloor; i >= 1; i--)
                        //use && to avoid recount dooropentimes
                        if(father->buttons[i].Get(DOWN) || (this->floorbutton.isPressed(i) && i < atfloor)){
                                minfloor = i;
                                dooropentimes++;
                        }
                //only one of the two following cases will be executed
                for(int i = minfloor; i < floor; i++)
                        if(father->buttons[i].Get(UP))
                                dooropentimes++;
                for(int i = minfloor; i > floor; i--)
                        if(father->buttons[i].Get(DOWN))
                                dooropentimes++;
                return this->timeGotoNextFloor * (maxfloor - this->atfloor + maxfloor - minfloor + fabs(floor - minfloor)) + (timeDoorOpen + timeDoorKee
pOpen + timeDoorClose) * dooropentimes;
        }
        //up(or then down after) and fetch person
        if(this->dir == UP && d == UP && this->atfloor < floor){
                for(int i = this->atfloor; i <= floor; i++)
                        if(this->father->buttons[i].Get(UP) || this->floorbutton.isPressed(i))
                                dooropentimes++;
                return this->timeGotoNextFloor * fabs(floor - this->atfloor) + (timeDoorOpen + timeDoorKeepOpen + timeDoorClose) * dooropentimes;
```

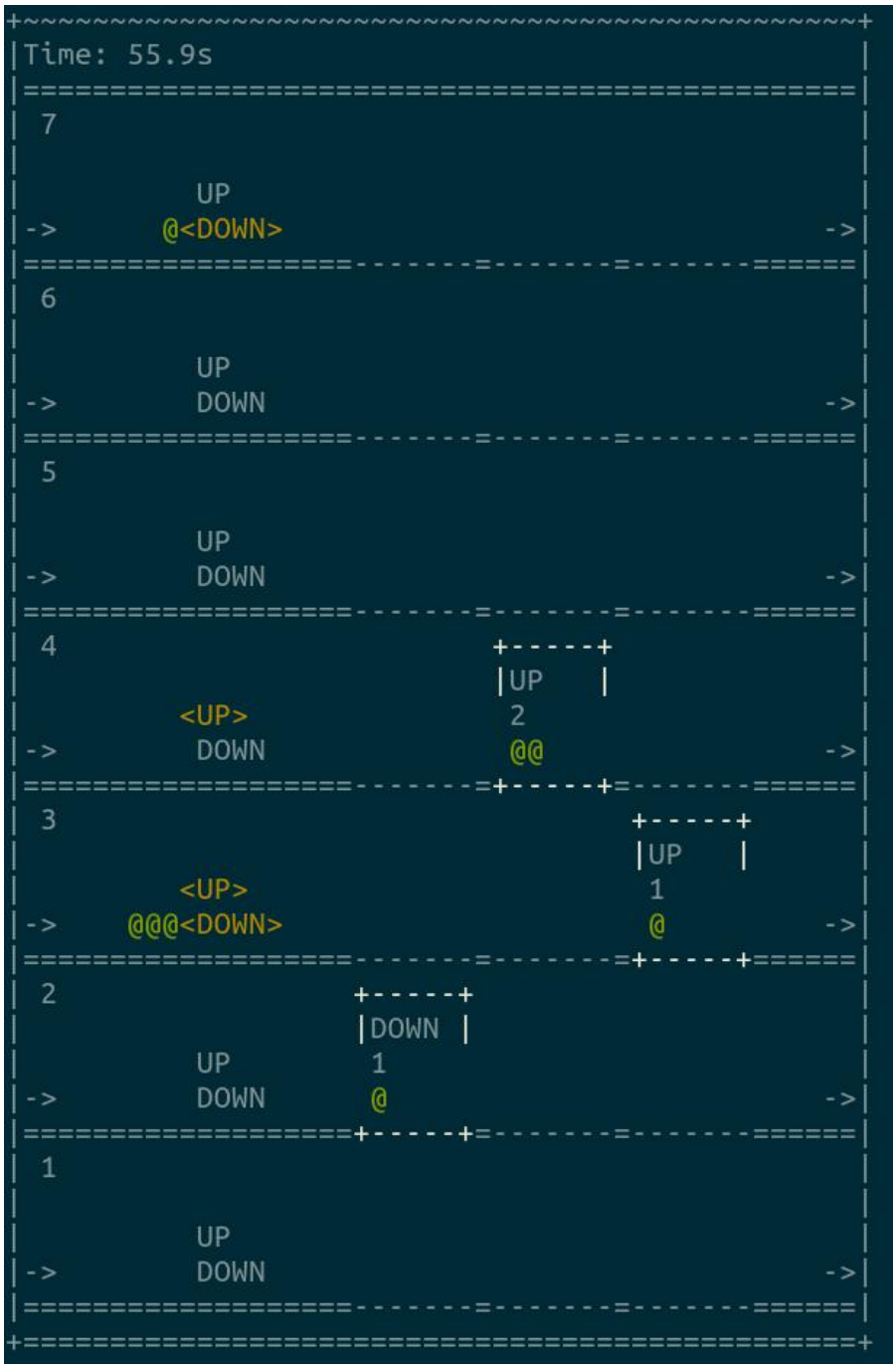
（2）界面绘制

由 SimpleCanvas 类在 Linux 终端中使用终端转义序列进行字符画输出。用户先将要输出的内容添加到 SimpleCanvas 上，然后在需要刷新屏幕的时候统一输出显示。该类可用于任何类似的简易界面绘制。

类内部使用两个二维数组存储每个“像素点”的字符和颜色，提供了 Show, ClearCanvas, ClearScreen, AddString, AddChar, AddRectangle, AddColLine, AddRowLine, ChangeColor 等方法，可满足本程序需求。

4. 实验结果及分析

实验能完善地模拟多电梯调度的行为，是成功的。运行图片：



可改进处：

电梯调度算法仍有改进空间，可以考虑动态分配和规划电梯；

应该使用真正的图形界面；

实现电梯空闲是回到“本垒”层；

头文件和程序文件的安排需要整理得更相关和有序。

5. 实验小结

本实验代码相对较为复杂，通过本实验，我感觉编写复杂代码的能力有了很大提升，对数据结构、基于回调的设计、离散模拟也有了更好的理解，同时体会到了编写可重用代码带来的方便之处。