第 1.3 题：停车场管理

实验报告

题目：编制一个实现停车场管理的程序。
班级：F1702120 学号：517021910526 姓名：陈聪 作业贡献度：55%
班级：F1702127 学号：517021910737 姓名：叶嘉迅 作业贡献度：45%

一、 需求分析

1. 以栈模拟停车场，以队列模拟车场外的便道，按照从终端读入的输入数据序列进行模拟管理。每一组输入数据包括三个数据项：汽车“到达”或“离开”信息、汽车牌照号码以及到达或离去的时刻。对每一个数据数据进行操作后的输出信息为：若是车辆到达，则输出汽车在停车场或便道上的停车位置；若是车辆离开，则输出汽车在停车场内停留的时间和应交纳的费用（在便道上停留的时间不收费）。栈以顺序结构实现，队列以链表结构实现。
2. 输入为以逗号隔开的操作顺序字符串。其中：’A’表示到达（Arrival）；’D’表示离去（Departure）；’E’表示输入结束（End）。
3. 需要另外设一个栈，临时停放为给要离开的汽车让路而从停车场退出的汽车，也用顺序存储结构实现。输入数据按照到达或离开的时刻有序。栈中每一个元素表示一辆汽车，包括两个数据项：汽车的拍照号码和进入停车场的时刻。
4. 汽车可以直接从便道上开走，此时排在它前面的汽车要先开走让路，然后再依次排到队尾。

二、 概要设计

本程序基本框架很清晰，即前述所说的“车辆进入停车场—>如果有位则进入停车，如果没位则在便道等候—>在便道上的车想要离开参考 4/在停车场内的车想要离开参考 1。”

为了对车进行更好的描述，程序中设计了车类 car。

停车场使用栈结构进行设计，便道使用队列进行设计。

选作题目所提出的便道上的车离开的情况要求便道队列为循环队列，因为顺序队列实现起来时间空间复杂度会很高。

此外，本程序中因为输入的字符串长度已确定，所以对栈、队列长度的设定可以在选择的时候就确定不会溢出，进而不需要 double space 函数。

1. 车类 car

对象：

```
int time_come;      //记录车第一次出现的时间
int real;           //记录车进入停车场的时间（用来算费用）
int token;          //记录车的代号
```

基本操作：

```
car(int token = 0, int time_come = 0);   //构造函数，初始化参数
void park(int time);                     //停车成功时调用，对 real 赋值
```

2. 栈类 seqStark

对象：

```
car* elements;      //存储车
int top_p;          //指向栈顶，便于栈的各种操作
int max_size;       //栈的最大大小
```

基本操作：

```
seqStack(int initial_size);          //构造函数，初始化一个栈
~seqStack();                         //析构函数，返还开辟的空间
void push (const car&x);             //将一个车压入栈，为栈顶
car pop ();                          //弹出栈顶的车
bool isEmpty();                      //返回栈是否为空
bool isFull();                       //返回栈是否为满
int length();                        //返回此时栈的长度
car top();                           //返回栈顶元素
```

3. 队列类 queue

对象：

```
car *elements;                       //存储此时队列中的车
int max_size;                        //队列的最大长度
int start, rear;                     //分别表示此时队列的起点和终点的下标
```

基本操作：

```
queque(int max_size = 0);            //构造函数，根据所给出最大大小初始化一个队列
~queque();                           //析构函数，返还开辟的所有空间
void enQueue(car target);            //进队函数，将一个车放进队尾
car deQueue();                       //出队函数，将队首的车放出
int length();                        //返回当前队列的长度
bool isEmpty();                      //返回当前队列是否为空
car head();                          //返回当前队列最后的车（这个命名稍微有点迷）
car tail();                          //返回当前队列最前方的车
int where(int target);               //寻找当前队列中是否有给定 target 车，有的话返回索引，
                                     //没有的话返回-1
```

4. 其他函数

string process(string sequence, int *sequence_number, int *sequence_time);

//字符串处理函数，将输入的字符串拆分成动作 action、车牌号 number、动作时间 time

void parking(string &sequence_action, int *sequence_number, int *sequence_time, int size_park);

//停车函数，整个程序算法部分，实现了停车的过程

停车过程通过输出以下内容：入停车场“Successful parking! ”、出停车场"Successful departure of car "、进排队"Please wait~"、出排队"The waiting car cant wait any more"来表现。

三、 详细设计

1. 车类

car.h

```
class car
{
public:
	int time_come;		//记录车第一次出现的时间
	int real;		//记录车进入停车场的时间（用来算费用）
	int token;		//记录车的代号
	car(int token = 0, int time_come = 0);	//构造函数，初始化参数
	void park(int time);				//停车成功时调用，对 real 赋值
};
```

car.cpp

```
//对参数初始化
car::car(int number, int time)
{
	token = number;
	time_come = time;
	real = 0;
}
//在停入停车场时调用，用来更新入库时间
void car::park(int time)
{
	real = time;
}
```

2. 循环队列类

queue.h

```
class queque
{
private:
    car *elements;                  //存储此时队列中的车
    int max_size;                   //队列的最大长度
    int start, rear;                //分别表示此时队列的起点和终点的下标

public:
    queque(int max_size = 0);       //构造函数，根据所给出最大大小初始化一个队列
    ~queque();                      //析构函数，返还开辟的所有空间
    void enQueue(car target);       //进队函数，将一个车放进队尾
    car deQueue();                  //出队函数，将队首的车放出
    int length();                   //返回当前队列的长度
    bool isEmpty();                 //判断当前队列是否为空
    car head();                     //返回当前队列最后的车（这个命名稍微有点迷）
    car tail();                     //返回当前队列最前方的车
    int where(int target);          //寻找当前队列中是否有给定 target 车，有的话返回索
                                    引，没有的话返回-1
};
```

queue.cpp

```
//构造函数，根据所给出最大大小初始化一个队列
queque::queque(int size_)
{
    elements = new car[size_];    //开辟空间
    max_size = size_;
    start = rear = 0;
}
//析构函数，返还开辟的所有空间
queque::~queque()
{
    delete [] elements;           //返还空间
    elements = nullptr;
}
//进队函数，将一个车放进队尾
void queque::enQueue(car target)
{
```

```
    elements[rear] = target;            //队尾
    rear = ( rear+1 ) % max_size;       //rear/start 中有一个不存储值也是循环链表的一个特
                                        //点
}


//出队函数，将队首的车放出
car queque::deQueue()
{
    int temp = start;
    start = ( start + 1)%max_size;      //非常重要的一步，决定了这个结构是循环队列
    return elements[temp];
}
//返回当前队列的长度
int queque::length()
{
    return start<rear? (rear - start) : (rear + max_size -start + 1) ;
}
//返回当前队列是否为空
bool queque::isEmpty()
{
    return start == rear;
}
//返回当前队列最后的车
car queque::head()
{
    int position = (start !=0 && rear ==0)? max_size - 1: rear - 1;
    return elements[position];
}
//返回当前队列最前方的车
car queque::tail()
{
    return elements[start];
}

//寻找当前队列中是否有给定 target 车，有的话返回索引，没有的话返回-1
//基本思路是：因为是循环队列，start 有可能大于 rear，针对两种情况对队列分别进行遍历
int queque::where(int target)
{
    if(start == rear) {return -1;}
    if(start < rear || start > 0 && rear == 0)          //情况 1：rear 大于 start（因为 rear 指向
                                                        //的位置不存储车，rear= 0 即为队列末
                                                        //尾元素在队尾）。此时队列不“断开”
    {
```

```
        int point = (rear == 0) ? max_size : rear;
        for (int i = start; i < rear; i++)
        {
            if(elements[i].token == target){return i;}
        }
    }
    if(start > rear && rear !=0)                    //情况 2，循环链表首尾“断开”了，先
    {                                               //遍历到 max_size，再从头遍历
        for (int i = start; i< max_size;i++)
        {
            if(elements[i].token == target){return i;}
        }
        for (int i = 0; i<rear; i++)
        {
            if(elements[i].token == target){return i;}
        }
    }
    return -1;
}
```

3. 栈类

seqStark.h

```
class seqStack
{
private:
    car* elements;      //存储车

public:
    queque(int max_size = 0);          //构造函数，根据所给出最大大小初始化一个队列
    ~queque();                         //析构函数，返还开辟的所有空间
    void enQueue(car target);          //进队函数，将一个车放进队尾
    car deQueue();                     //出队函数，将队首的车放出
    int length();                      //返回当前队列的长度
    bool isEmpty();                    //判断当前队列是否为空
    car head();                        //返回当前队列最后的车（这个命名稍微有点迷）
    car tail();                        //返回当前队列最前方的车
    int where(int target);             //寻找当前队列中是否有给定 target 车，有的话返回索
                                       引，没有的话返回-1
};
```

seqStark.cpp

```
//停车函数，用于实现程序所要求的停车过程，具体的停车过程通过 cout 给出
void parking(string &sequence_action, int *sequence_number, int *sequence_time, int
size_park)
{

    int number = sequence_action.length(),iteration_number = 0;   //the number of actions
    seqStack park(size_park);
    queque waiting(sequence_action.length()-size_park + 1);
    seqStack temp(size_park);
```

number: 所有将要采取的动作数，主要用于后面对 action 的遍历
park: 停车场栈，用于存放停车场中的车辆
temp：停车场外地区，用于在停车场中有车出栈时，存储暂时出栈让出位置的车辆

```
    for (int time_step =1; time_step <= sequence_time[number-1]; time_step ++)
    {
        std::cout<<"Time step "<<time_step<<std::endl;
        //用于表现时间的流逝。Time step 相当于计时器，每过一分钟，+1。
        if(time_step == sequence_time[iteration_number])
        //当当前的时间步有动作时，进行动作。（有可能时间步没有动作，比如输入时间
```

```
        //分别为 5、10，则 1~4 就是空的时间步）
        {

            //当动作为车到来时，车辆尝试进入停车场，若停车场已满，则进入便道
            if(sequence_action[iteration_number] == 'A')
            {
                car
newcar(sequence_number[iteration_number],sequence_time[iteration_number]);
                if(park.isFull() == 0)                    //停车场未满时，进入停车场
                {
                    newcar.park(time_step);
                    newcar.real = time_step;
//对车的两个时间属性赋值，便于之后的计算
                    park.push(newcar);
                    std::cout<<"Successful parking! The position of car
"<<park.top().token<<" is in PARK "<<park.length()+1<<std::endl;
                }
                else                                      //停车场满了时，停入便道
                {
                    waiting.enQueue(newcar);
                    std::cout<<"Please wait~ The position of car "<<waiting.head().token<<"
is in Waiting"<<waiting.length()<<std::endl;
                }
            }
/*---------------------------------------------------------------------------
动作为到达的部分完成，以下完成动作为出停车场操作
---------------------------------------------------------------------------
*/

            else                                          //动作为 D 时
            {
                int position_of_number_in_waiting = 0;
                position_of_number_in_waiting                                    =
waiting.where(sequence_number[iteration_number]);
//尝试在便道中寻找这辆车，如果找不到，则该车在停车场中，若找到了，按照选作的要求
进行操作
                if(position_of_number_in_waiting != -1)
//便道中找到了这辆车，则按照选作中的要求退出这辆车
                {
                    queque temp_queue(sequence_action.length()-size_park + 1);
                    while (waiting.tail().token != sequence_number[iteration_number])
                    {
                    temp_queue.enQueue(waiting.deQueue()); //在前面的车到别处等待
                    }
```

```
                    std::cout<< "The waiting car "<<waiting.tail().token<<" cant wait any
more"<<std::endl;
                    waiting.deQueue();
                    while(temp_queue.isEmpty() == 0)
                    {
                        waiting.enQueue(temp_queue.deQueue());
                    //暂时到别处的车返回队列
                    }
                }
                //如果在便道中没有找到这辆车，则从停车场中开出这辆车
                else{
                //先把挡在出来的车后面的车开出去
                while(sequence_number[iteration_number] != park.top().token)
                {
                    temp.push(park.top());
                    park.pop();
                }
                //输出出车信息
                car temp2 = park.pop();
//输出总停车时间和有效停车时间（算钱的时间）
                std::cout<<"Successful departure of car "<<temp2.token<<" ! The time is "<<
time_step - temp2.time_come
                <<" and the fee is " << time_step - temp2.real<<std::endl;
                //暂时开到别处的车返回
                while(temp.isEmpty() == 0)
                {
                    park.push(temp.top());
                    temp.pop();
                }
                //便道中有一辆车此时可以进入停车场，前提是便道有车的话
                if(waiting.isEmpty() == 0)
                {
                    car temp1 = waiting.deQueue();
                    temp1.real = time_step;
                    park.push(temp1);
                    std::cout<<"The waiting car "<<park.top().token<<" comes into the park!
The position is in PARK "<<park.length()+1<<std::endl;
                }
                }
            }
            ++iteration_number;   }
    }}
//将一个 string 的四位数字转化为 int 格式的数字
    int trans(string a )
```

```
{
	int length = a.length();
	int target = 0;
	for (int i = 0; i<length;++i)
	{
		target = target * 10;
		target += a[i] - 48;
	}
	return target;
}


//对输入的字符串进行处理的函数，将动作、车号、时间分别按顺序存入相应的字符串、数
//组中
string process(string sequence, int *sequence_number, int *sequence_time)
{
	int counter = 0, dot = 0;
	string sequence_action;
	while (sequence != "")
	{
		if(sequence[2] == 'E'){break;}

		sequence_action.insert(counter,1, sequence[2]);
		sequence = sequence.substr(int(sequence.find(',')) + 1);
		dot = sequence.find(',');
		string number = sequence.substr(0,dot);
		int temp_number = trans(number);
		sequence_number[counter] = temp_number;

		sequence = sequence.substr(int(sequence.find(',')) + 1);
		int right = sequence.find(')');
		string time = sequence.substr(0,right);
		int temp_time = trans(time);
		sequence_time[counter] = temp_time;

		sequence = sequence.substr(int(sequence.find(',')) + 1);

		counter ++;
	}
	return sequence_action;
}

//构造函数，开辟空间
seqStack::seqStack(int initial_size = 10)
```

```
{
	max_size = initial_size;
	elements = new car[max_size];
	top_p = -1;
}
//析构函数，返还开辟的空间
seqStack::~seqStack()
{
	delete [] elements;
	elements = nullptr;
}
//将数据压入栈
void seqStack::push(const car &x)
{
	elements[++top_p] = x;
}
//将栈顶元素弹出栈
car seqStack::pop()
{

	return elements[top_p--];
}
//判断栈是否为空
bool seqStack::isEmpty()
{
	return top_p == -1;
}
//判断栈是否为满
bool seqStack::isFull()
{
	return (top_p == max_size - 1);
}
//返回栈顶元素
car seqStack::top()
{
	return elements[top_p];
}
//返回栈的长度
int seqStack::length()
{
	return top_p;
}
```

4. 主函数部分

```
int main()
{
	string sequence, sequence_action;
	int *sequence_number, *sequence_time, size_park;

	//输入停车场的进出车序列和停车场规模
	cout<<"Please give the sequence:"<<endl;
	cin>>sequence;
	cout<<"Please give the size of park:"<<endl;
	cin>>size_park;

	//对字符串进行处理，得到动作、车号、时间的值
	sequence_number = new int[sequence.length()/10 + 1];
	sequence_time = new int[sequence.length()/10 + 1];
	sequence_action = process(sequence,sequence_number,sequence_time);
	cout<<endl;

	parking(sequence_action, sequence_number, sequence_time, size_park); 停车//

	//返还开辟的空间
	delete [] sequence_number;
	delete [] sequence_time;
	sequence_number = nullptr;
	sequence_time = nullptr;

	return 0;
}
```

四、 调试分析

1. 早期程序中，park 函数没有单独进行封装而是整体写在 main 函数中，导致了 main 函数过于冗长，在程序设计后期才想起来单独进行封装，类似的函数封装以后应该尽早进行，否则迁移会很占用时间。
2. 队列设置有问题。最开始的设计并没有考虑到选作需求，所以队列一开始是顺序队列，但看了选做题之后发现顺序队列想要实现选作的效果的话可能会需要反复不必要的进出列，时间复杂度空间复杂度都很高，随后改成了循环队列来实现。
3. 注释习惯没有统一。这个问题想要解决还需要多加练习…在写完代码后复看自己代码的过程中发现自己注释的习惯并不一致，比如有的时候会像 github 的注释一样专门画个好看的方框，有的时候是在行前注释，有的时候在行后注释。以后会尽量对风格进行统一，暂时的想法是都画方框注释。

4. 算法的复杂度分析

1） 时间复杂度

队列类中：

```
queque(int max_size = 0);          //开辟空间，时间复杂度为 O(1)
~queque();                         //释放空间，时间复杂度 O(1)
void enQueue(car target);          //入队列，时间复杂度为 O(1)
car deQueue();                     //出队列，时间复杂度为 O(1)
int length();                      //返回当前的长度，时间复杂度 O(1)
bool isEmpty();                    //调用当前的队首队尾，时间复杂度 O(1)
car head();                        //返回当前队尾车辆，时间复杂度 O(1)
car tail();                        //返回当前队首车辆，时间复杂度 O(1)
int where(int target);             //按照车辆号码查找当前，时间复杂度 O(n)
```

栈类中：

```
seqStack(int initial_size);        //构造函数开辟空间，时间复杂度 O(1)
~seqStack();                       //析构函数释放空间，时间复杂度 O(1)
void push (const car&x);           //入栈，将一个元素放入栈顶，时间复杂度 O(1)
car pop ();                        //出栈，将栈顶元素弹出，时间复杂度 O(1)
bool isEmpty();                    //判断栈此时是否为空，时间复杂度 O(1)
bool isFull();                     //判断栈是否为满，时间复杂度 O(1)
int length();                      //返回当前栈长度，访问 top_p，时间复杂度 O(1)
car top();                         //返回当前栈顶元素，时间复杂度 O(1)
```

车类中：

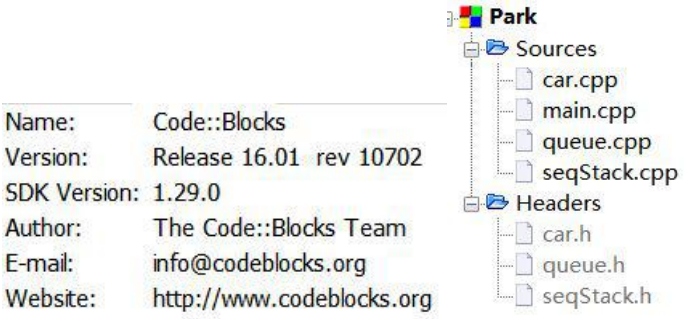
```
car(int token = 0, int time_come = 0);  //构造函数，对参数进行初始化，时间复杂度 O(1)
void park(int time);                    //停车函数，对参数进行赋值，时间复杂度 O(1)
```

2) 空间复杂度

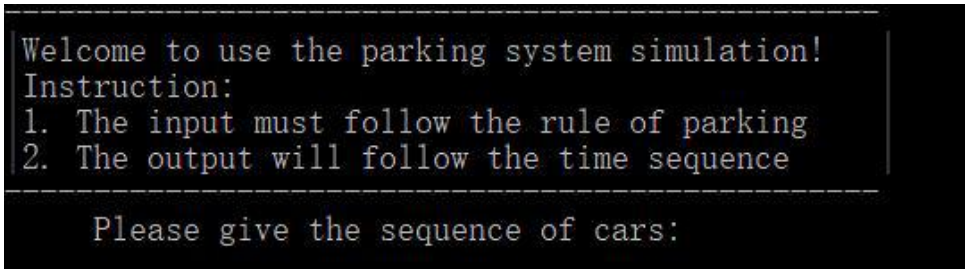
在所有函数中，只有栈和列表的构造函数(seqStack(int initial_size),queque(int max_size = 0))的空间复杂度为 O(n)，其他函数的空间复杂度都是 O(1)。

五、 用户手册

1. 本程序使用的 Code::Blocks 16.01 IDE，程序以项目（project）方式组织，如图 1 所示：



2. 依次点击菜单“Build”-> “Build and run”，显示文本方式的用户界面，如图 2 所示：



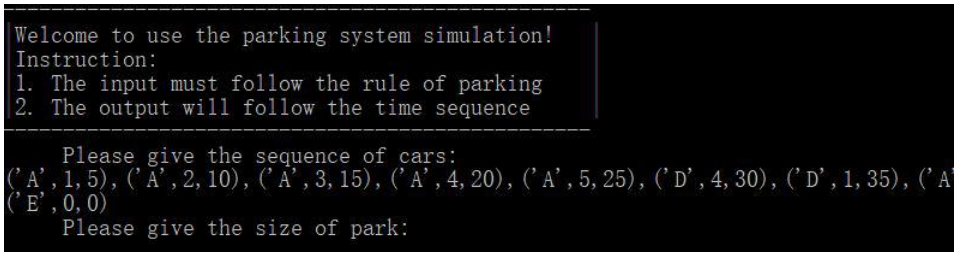
请注意：

1） 在文件夹中，输入样例.txt 提供了两组测试数据（其中一组是 pdf 给出的）。*请始终保持英文输入法*

2） 输入过程中，车号与时间必须为整数。

3） 输入过程中，所提供的 sequence 必须按照时间序列给出（模拟停车场顺序进出车）。

3. 在输入车辆出入队列后，还须输入停车场规模：



4. 再输入回车后，即可按照时间序列得到停出车情况。

六、 测试结果

('A',1,5),('A',2,10),('D',1,15),('A',3,20),('A',4,25),('D',4,27),('A',5,30),('D',2,35),('D',5,40),('E',0,0)
作为输入。
输出：
Time step 1
Time step 2
Time step 3
Time step 4
Time step 5
Successful parking! The position of car 1 is in PARK 1
Time step 6
Time step 7
Time step 8
Time step 9
Time step 10
Successful parking! The position of car 2 is in PARK 2
Time step 11
Time step 12
Time step 13
Time step 14
Time step 15
Successful departure of car 1 ! The time is 10 and the fee is 10
Time step 16
Time step 17
Time step 18
Time step 19
Time step 20
Successful parking! The position of car 3 is in PARK 2
Time step 21
Time step 22
Time step 23
Time step 24
Time step 25
Please wait~ The position of car 4 is in Waiting1
Time step 26
Time step 27
The waiting car 4 cant wait any more
Time step 28
Time step 29
Time step 30
Please wait~ The position of car 5 is in Waiting1

Time step 31
Time step 32
Time step 33
Time step 34
Time step 35
Successful departure of car 2 ! The time is 25 and the fee is 25
The waiting car 5 comes into the park! The position is in PARK 2
Time step 36
Time step 37
Time step 38
Time step 39
Time step 40
Successful departure of car 5 ! The time is 10 and the fee is 5

七、 附录