停车场管理实验报告

题目：编制一个模拟停车场管理的程序

班级：F1702128　姓名：沈嘉欢　学号：517021910872　完成日期：2018 年 9 月 28 日

一、 需求分析

本程序可实现模拟停车场的管理包括车辆的进入、离开、价格计算、停车场满时在便道等候，也能够模拟汽车直接从便道离开的需求。

用户设定停车场容量、单位时间停车费用，并给出每辆汽车的到来、离开（停车场或便道）时间，可以计算出车辆的停车费用。

1.1 测试样例

输入样例：

```
2 1
A 1 5
A 2 10
D 1 15
A 3 20
A 4 25
A 5 30
L 5 0
D 2 35
D 4 40
E 0 0
```

输出样例：

```
10 10
25 25
5 5
```

1.2 样例说明

输入第一行为两个整数，分别代表停车场的容量和单位时间停车费用；接下来的若干行，由一个字母和两个数字组成：第一个字母表示发生事件状态，第二个数字表示车牌号，第三个数字表示发生时间，其中：

1. 对于字母：`A` 表示到达 (Arrival)，`D` 表示离去 (Departure)，`L` 表示直接从便道离开 (Left)，`E` 表示输入结束 (End)；
2. 对于状态 `L`，其时间可以是任意的，因为从便道离开不需要收费，我们对离开时间并不关心；
3. 对于状态 `E`，其车牌号和时间均可以是任意的；
4. 输入数据保证若状态为 `L`，则该车当前必然处于便道；

5. 输入数据保证若状态为 D，则该车当前必然处于停车场。

该停车场有如下规则：

1. 若停车场未满，则车辆进入停车场，且停车场满足后进先出规则;
2. 若停车场已满，则车辆进入便道等候，便道满足先进先出规则；
3. 便道上的车辆可以选择直接离开，此时，前方车辆依次排至需要离开的车辆的后方。

若车辆选择离开，输出其所需停车费。

二、 概要设计

对于停车场模拟问题，可以使用栈和队列的数据结构：

- 停车场后进先出，使用栈进行模拟；
- 便道先进先出，使用队列进行模拟；
- 车辆从停车场离开时，前方车辆需要为其让路，为保持前方车辆顺序不变，使用栈进行模拟。

2.1 ParkingLot 类基本操作

```
ParkingLot(size, unitPrice)
```

操作结果：用停车场容量和收费单价初始化一个停车场。

```
arrival(id, arrivalTime)
```

初始条件：ParkingLot 对象已建立。
操作结果：车牌号为 id 的车辆在 arrivalTime 时刻到达 ParkingLot 实例。

```
departure(id, departureTime)
```

初始条件：ParkingLot 对象已建立，车牌号为 id 的车辆在停车场内。
操作结果：车牌号为 id 的车辆在 arrivalTime 时刻从停车场离开 ParkingLot 实例，并返回其停车时间和收费金额。

```
departureFromLane(id)
```

初始条件：ParkingLot 对象已建立，车牌号为 id 的车辆在便道内。
操作结果：车牌号为 id 的车辆在 arrivalTime 时刻从便道离开 ParkingLot 实例，便道内其余车辆改变位置。

2.2 NewStack 类基本操作

NewStack 类的行为与 STL 基本一致，但仅实现了所需的几个功能。

```
NewStack()
```

操作结果：创建一个空的栈。

```
expand()
```

初始条件：栈空间已满。
操作结果：栈空间增倍。

```
push(data)
```

初始条件：NewStack 对象已建立。
操作结果：将数据压入栈中。

```
top()
```

初始条件：NewStack 对象已建立，且栈非空。
操作结果：返回栈顶元素值。

```
pop()
```

初始条件：NewStack 对象已建立，且栈非空。
操作结果：栈顶元素出栈。

```
size()
```

初始条件：NewStack 对象已建立。
操作结果：返回栈的大小。

2.3 NewQueue 类基本操作

NewQueue 类的行为与 STL 基本一致，但仅实现了所需的几个功能。

```
NewQueue()
```

操作结果：创建一个空的队列。

```
push(data)
```

初始条件：NewQueue 对象已建立。
操作结果：将数据入队。

```
front()
```

初始条件：NewQueue 对象已建立，且队列非空。
操作结果：返回队首元素值。

```
pop()
```

初始条件：NewQueue 对象已建立，且队列非空。
操作结果：对首元素出队列。

```
size()
```

初始条件：NewQueue 对象已建立。
操作结果：返回队列的大小。

2.4 模块说明

本程序分为六个模块：

- 主程序模块，测试数据的输入输出；
- ParkingLot 模块，定义停车场类；
- NewStack 模块，定义栈类；
- NewQueue 模块，定义队列类；
- TestStack 模块，NewStack 类的单元测试；
- TestQueue 模块，NewQueue 类的单元测试；

三、 详细设计

3.1 ParkingLot 类声明

ParkingLot 类启用了条件编译，编译时若启用 DEBUG 宏，则使用 STL 自带的栈和队列；若禁用，则使用本项目中 NewStack 类实现栈，NewQueue 类实现队列。

```
//
// ParkingLot.hpp
// ParkingLot
//
// Created by 沈嘉欢 on 2018/9/28.
// Copyright © 2018 沈嘉欢. All rights reserved.
//

#ifndef ParkingLot_hpp
#define ParkingLot_hpp
```

```cpp
#include <string>
#include <cassert>

#ifdef DEBUG
#include <stack>
#include <queue>
template <typename T>
using stack = std::stack<T>;
template <typename T>
using queue = std::queue<T>;
#else
#include "NewStack.hpp"
#include "NewQueue.hpp"
template <typename T>
using stack = NewStack<T>;
template <typename T>
using queue = NewQueue<T>;
#endif /* DEBUG */

class ParkingLot {
private:
    struct Car {
        std::string id;
        uint32_t arrivalTime;
        uint32_t departureTime;
        Car() {};
        Car(const std::string &id, uint32_t arrivalTime): id(id), arrivalTime(arrivalTime),
            departureTime(0) {};
    };
    uint32_t size;
    double unitPrice;
    stack<Car> lot; // 停车场
    queue<Car> lane; // 便道
    std::pair<uint32_t, double> calc(const Car &c);
public:
    ParkingLot(uint32_t size, double unitPrice);
    void arrival(const std::string &id, uint32_t arrivalTime);
    std::pair<uint32_t, double> departure(const std::string &id, uint32_t departureTime);
    void departureFromLane(const std::string &id);

};

#endif /* ParkingLot_hpp */
```

3.2 ParkingLot 类实现

```
//
// ParkingLot.cpp
// ParkingLot
//
// Created by 沈嘉欢 on 2018/9/28.
// Copyright © 2018 沈嘉欢. All rights reserved.
//

#include "ParkingLot.hpp"

ParkingLot::ParkingLot(uint32_t size, double unitPrice) {
    this->size = size;
    this->unitPrice = unitPrice;
};

std::pair<uint32_t, double> ParkingLot::calc(const ParkingLot::Car &c) {
    assert(c.departureTime >= c.arrivalTime);
    uint32_t duration = c.departureTime - c.arrivalTime;
    double price = duration * unitPrice;
    return std::make_pair(duration, price);
}


// 若停车场未满，车辆进入停车场，否则进入便道
void ParkingLot::arrival(const std::string &id, uint32_t arrivalTime) {
    Car c(id, arrivalTime);
    if (lot.size() < size) {
        lot.push(c);
    } else {
        lane.push(c);
    }
}

std::pair<uint32_t, double> ParkingLot::departure(const std::string &id, uint32_t
     departureTime) {
    Car c;
    if (lot.top().id == id) {
        c = lot.top();
        lot.pop();
        c.departureTime = departureTime;
    } else { // 停车场前方车辆让路
        stack<Car> tmp;
        while (lot.top().id != id) {
            tmp.push(lot.top());
            lot.pop();
```

```
            assert(lot.size() > 0);
        }
        c = lot.top();
        lot.pop();
        c.departureTime = departureTime;
        while (tmp.size() != 0) {
            lot.push(tmp.top());
            tmp.pop();
        }
    }
    // 便道可以移动一辆车至停车场
    if (lane.size() != 0) {
        Car tmp = lane.front();
        tmp.arrivalTime = departureTime;
        lot.push(tmp);
        lane.pop();
    }
    return calc(c);
}

void ParkingLot::departureFromLane(const std::string &id) {
    uint32_t cnt = 0;
    // 便道前方车辆让路，依次将队首车辆移至队尾直至队首为需要离开的车辆
    while (lane.front().id != id && cnt != lane.size()) {
        lane.push(lane.front());
        lane.pop();
        cnt++;
    }
    // 如果不相等意味着输入存在异常，试图将一辆不在便道的车从便道移出
    assert(lane.front().id == id);
    lane.pop();
}
```

3.3 NewStack 类实现

由于 NewStack 类被定义为模版类，故在类声明内部编写类实现代码。

由于本人前期已对常规动态数组的 new、delete 操作实验较多，本次实验在 NewStack 类使用智能指针实现动态内存分配（需要 C++11 标准支持），优点在于其可自动回收无效内存空间而不需要手动释放。

```
//
// NewStack.hpp
// ParkingLot
//
// Created by 沈嘉欢 on 2018/9/28.
```

```cpp
// Copyright © 2018 沈嘉欢. All rights reserved.
//

#ifndef NewStack_hpp
#define NewStack_hpp

#include <memory>

template <typename T>
class NewStack {
private:
    uint32_t stackSize;
    uint32_t stackMaxSize;
    std::unique_ptr<T[]> list;
    // 栈空间满时，扩容为原来的两倍
    void expand() {
        stackMaxSize *= 2;
        std::unique_ptr<T[]> newList(new T[stackMaxSize]);
        for (uint32_t i = 0; i < stackSize; ++i) {
            newList[i] = list[i];
        }
        list.reset(newList.release());
    }
public:
    NewStack(): stackSize(0), stackMaxSize(8), list(new T[stackMaxSize]) {};
    void push(const T &data) {
        if (stackSize == stackMaxSize) {
            expand();
        }
        list[stackSize] = data;
        ++stackSize;
    }
    T top() const {
        assert(stackSize != 0);
        return list[stackSize - 1];
    }
    void pop() {
        assert(stackSize != 0);
        --stackSize;
    }
    uint32_t size() const {
        return stackSize;
    }
};

#endif /* NewStack_hpp */
```

3.4 NewQueue 类实现

由于 NewQueue 类被定义为模版类，故在类声明内部编写类实现代码。

由于本人前期已对常规动态数组的 new、delete 操作实验较多，本次实验在 NewQueue 类使用智能指针实现动态内存分配（需要 C++11 标准支持），优点在于其可自动回收无效内存空间而不需要手动释放。

```
//
// NewQueue.hpp
// ParkingLot
//
// Created by 沈嘉欢 on 2018/9/28.
// Copyright © 2018 沈嘉欢. All rights reserved.
//

#ifndef NewQueue_h
#define NewQueue_h

#include <memory>

template <typename T>
class NewQueue {
private:
    // 内嵌的结点结构体
    struct Node {
        T data;
        std::shared_ptr<Node> next;
        Node(): next(nullptr) {};
        Node(T data): data(data), next(nullptr) {};
    };
    std::shared_ptr<Node> rearPtr, frontPtr;
    uint32_t queueSize;
public:
    NewQueue(): rearPtr(nullptr), frontPtr(nullptr), queueSize(0) {};
    void push(const T &data) {
        auto current = std::make_shared<Node>(data);
        if (queueSize != 0) {
            rearPtr->next = current;
            rearPtr = current;
        } else {
            rearPtr = frontPtr = current;
        }
        ++queueSize;
    }
    T front() const {
        assert(queueSize != 0);
```

```
        return frontPtr->data;
    }
    void pop() {
        assert(queueSize != 0);
        frontPtr = frontPtr->next;
        --queueSize;
    }
    uint32_t size() const {
        return queueSize;
    }
};
#endif /* NewQueue_h */
```

3.5 主程序模块

```
//
// main.cpp
// ParkingLot
//
// Created by 沈嘉欢 on 2018/9/28.
// Copyright © 2018 沈嘉欢. All rights reserved.
//

#include <iostream>
#include "ParkingLot.hpp"

int main(int argc, const char * argv[]) {
    int n, unit_price;
    std::cin >> n >> unit_price;
    ParkingLot p(n, unit_price);
    bool alive = true;
    while (alive) {
        char choice;
        std::string id;
        uint32_t time;
        std::cin >> choice >> id >> time;
        switch (choice) {
            case 'A': {
                p.arrival(id, time);
                break;
            }
            case 'D': {
                auto property = p.departure(id, time);
                std::cout << property.first << " " << property.second << "\n";
                break;
```

```
            }
            case 'E': {
                alive = false;
                break;
            }
            case 'L': {
                p.departureFromLane(id);
                break;
            }
            default:
                std::cout << "Error!\n";
                break;
        }
    }
    return 0;
}
```

四、 调试分析

4.1 代码评价

本程序实现了停车场的模拟。

程序遵循了面向对象编程的要求，将各个模块独立开来，类接口定义合理、简洁。

NewStack 和 NewQueue 类定义为模版类，可以很方便地移植到其他程序中。

4.2 时间复杂度分析

4.2.1 NewStack 类

- 平均情况下，push(data) 操作的时间复杂度为 $O(1)$，最坏情况下，如果栈需要扩容，则 push(data) 操作的时间复杂度为 $O(n)$；
- top()、pop()、size() 操作的时间复杂度都为 $O(1)$。

4.2.2 NewQueue 类

- push(data)、front()、pop()、size() 操作的时间复杂度都为 $O(1)$。

4.2.3 ParkingLot 类

- 平均情况下，arrival(id, arrivalTime) 操作的时间复杂度为 $O(1)$，最坏情况下，如果栈需要扩容，则 arrival(id, arrivalTime) 操作的时间复杂度为 $O(n)$；
- departure(id, departureTime) 操作的时间复杂度为 $O(n)$；

- `departureFromLane(id)` 操作的时间复杂度为 $O(n)$。

4.3 性能测试

为了测试程序性能，我使用了一个 Python 程序随机生成一些数据（程序和数据见附件），测试程序运行时间。为简单起见，仅测试停车场无限大的情况（即不会有车辆进入便道）。比较 STL 和自定义栈、队列的运行时间，同不开优化的条件下，STL 略快于自定义的栈和队列。

4.4 存在的问题

`NewStack` 类没有设定 `shrink` 的私有成员函数，即当栈所需空间远小于其最大空间时，缩减栈的大小。程序时间并不复杂，但是问题在于停车场后方车辆离开，需要前面所有车辆临时出栈再入栈。这一过程中栈的最大容量很可能频繁地减小、增大，十分影响程序性能。出于时间、空间性能的综合考虑，我取消了栈空间自动收缩的功能。

五、 用户手册

本程序在 Apple LLVM version 10.0.0 (clang-1000.11.45.2) 环境编译通过，文件组织结构为:

```
ParkingLot
    main.cpp
    ParkingLot.hpp
    NewStack.cpp
    NewQueue.cpp
    ParkingLot.cpp
TestStack
    main.cpp
TestQueue
    main.cpp
```

使用编译命令 `$ clang++ -o ParkingLot main.cpp ParkingLot.cpp -std=c++11`，会生成名为 `ParkingLot` 的 Unix 可执行文件（启用自定义栈、队列类）。

使用编译命令 `$ clang++ -o ParkingLot main.cpp ParkingLot.cpp -std=c++11 -DDEBUG`，会生成名为 `ParkingLot` 的 Unix 可执行文件（启用 STL 栈、队列容器）。

六、 测试结果

```
$ clang++ -o ParkingLot main.cpp ParkingLot.cpp -std=c++11
```

```
$ ./ParkingLot
2 1
A 1 5
A 2 10
D 1 15
10 10
A 3 20
A 4 25
A 5 30
L 5 0
D 2 35
25 25
D 4 40
5 5
E 0 0
$
```