栈与队列实验报告

8207200203 计科2006班 翟衍博

目录：

1.栈和队列操作的实现

1.1 需求分析

（1）输入的形式和输入值的范围

对顺序栈进行创建、压栈、弹栈、取栈顶元素、判栈空、栈满等操作。

对循环队列进行创建、入队、出队、取队首元素、判队空、队满等操作。

（2）输出的形式

在初始化顺序栈、循环队列后。进行一些操作测试，可以输出栈与队列是否为空、栈顶元素的值、队首元素的值等等。

（3）程序所能达到的功能

- 建立一个顺序栈
- 建立一个循环顺序队列
- 分别实现栈和队列的基本操作

（4）测试数据

- 初始化整型顺序栈后，进行判空与判满

- 依次将1、2、3压栈，进行判空与判满，获取栈中元素个数
- 再次将1、2、3压栈，然后输出栈中全部元素
- 进行判空和判满

1.2 概要设计

（1）抽象数据类型定义

栈：

```
ADT Stack{
    数据对象:D={ai|ai∈ElemSet,i=1,2,...,n,n>=0}
    数据关系:R={<ai-1,ai>|ai-1,ai∈D,i=2,...,n}
    基本操作:
    InitStack(&S)
        操作结果：构造一个空栈S
    DestoryStack(&S)
        初始条件：栈S已存在
        操作结果：栈S被销毁
    StackEmpty(S)
        初始条件：栈S已存在
        操作结果：若栈S为空栈，则返回true,否则返回false
    StackLength(S)
        初始条件：栈S已存在
        操作结果：返回S的元素个数，即栈的长度
    GetTop(S)
        初始条件：栈S存在且非空
        操作结果：返回S的栈顶元素，不修改栈顶指针
    Push(&S,e)
        初始条件：栈S已存在
        操作结果：插入元素e为新的栈顶元素
    Pop(&S,e)
        初始条件：栈S已存在且非空
        操作结果：删除S的栈顶元素，并用e返回其值
}ADT Stack
```

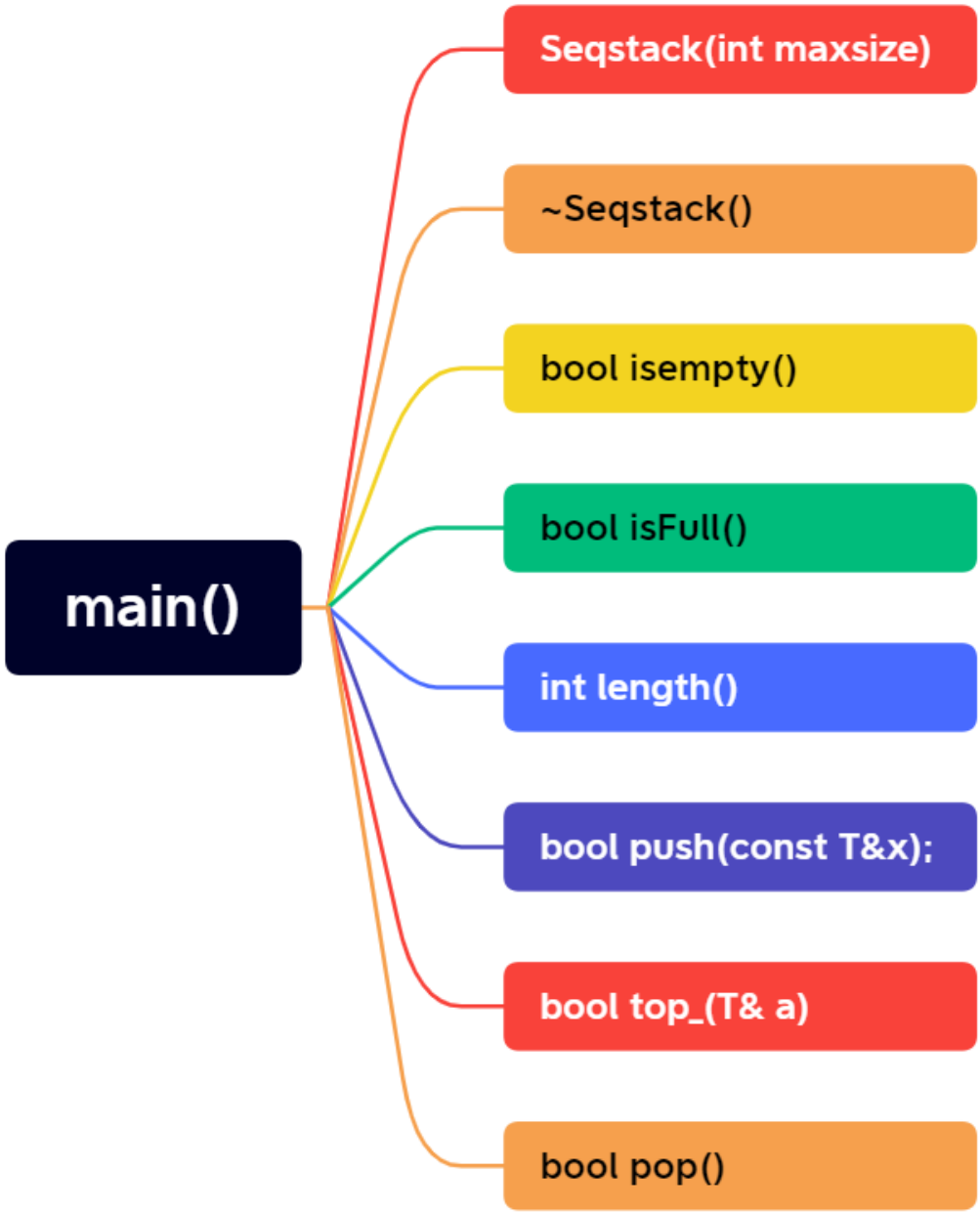
队列：

```
ADT Queue{
    数据对象:D={ai|ai∈ElemSet,i=1,2,...,n,n>=0}
    数据关系:R={<ai-1,ai>|ai-1,ai∈D,i=2,...,n}
             约定其中ai端为队列头，an端为队列尾。
    基本操作:
    InitQueue(&Q)
        操作结果：构造一个空队列Q
    DestoryQueue(&Q)
        初始条件：队列Q已存在
        操作结果：队列Q被销毁
    QueueEmpty(Q)
        初始条件：队列Q已存在
        操作结果：若队列Q为空队列，则返回true,否则返回false
    QueueLength(Q)
        初始条件：队列Q已存在
        操作结果：返回Q的元素个数，即队列的长度
    GetHead(Q)
```

```
        初始条件：队列Q存在且非空
        操作结果：返回Q的队首元素，不修改队首指针
    EnQueue(&Q,e)
        初始条件：队列Q已存在
        操作结果：插入元素e为新的队尾元素
    DeQueue(&Q,e)
        初始条件：队列Q已存在且非空
        操作结果：删除Q的队首元素，并用e返回其值
}ADT Stack
```

（2）主程序流程和程序模块之间的关系

在main（）函数中实例化一个顺序栈对象，然后调用它的函数。



1.3 详细设计

（1）公共接口。根据需求分析的结果，得到的顺序栈类的模板类如下：

```
template<typename T>
class Seqstack
{
private:
    int maxsize;//栈的最大容量
    int top;    //栈顶
    T* element; //动态数组
public:
    Seqstack(int maxsize);  //构造函数，创建栈
    ~Seqstack();            //析构函数，销毁栈
    bool isempty();     //判断栈是否为空
    bool isFull();      //栈是否已满
    int length();       //获取栈中元素个数
    bool push(const T&x);   //入栈，成功返回true
    bool top_(T& a);     //获取栈顶元素，成功返回true
    bool pop();         //删除栈顶元素，成功返回true
    void show();        //将栈顺序输出
};
```

循环队列的模板类如下：

```
template<typename T>
class CirQueue
{
private:
    T data[MaxQueueSize];
    int front;
    int rear;
    int count;
public:
    CirQueue();     //构造函数
    ~CirQueue();    //析构函数
    bool isempty(); //判断循环队列是否为空
    bool isfull();  //判断循环队列是否为满
    int getlength();//获取循环队列中元素个数
    bool enqueue(const T& x);   //入队
    bool dequeue(T &x);         //出队
    bool getfront(T &x);        //取队首元素
};
```

（2）数据成员。

顺序栈类中有栈的最大容量 maxsize、栈顶指针top、T类型指针element三个数据成员，为私有数据成员。

循环队列类中有存储数据的数组，表示头指针和尾指针的变量，此外还有表示循环队列中元素个数的变量（仅用头尾变量无法区分是队空还是队满的情况）。

（3）主要操作的实现

顺序栈中主要操作的实现：

```
//构造函数
template<typename T>
Seqstack<T>::Seqstack(int maxsize)
{
    this->maxsize=maxsize;
    element=new T[maxsize];
    if(!element)
    {
        cout<<"内存申请失败！";
        exit(0);
    }
    this->top=0;    //栈顶指针指向栈顶元素的上方一个位置，因此top初值为0

}

//析构函数
template<typename T>
Seqstack<T>::~Seqstack()
{
    delete []element;
    //将申请的空间释放掉
}

//判断栈是否为空
template<typename T>
bool Seqstack<T>::isempty()
{
    return (this->top==0);
}

//判断栈是否为满
template<typename T>
bool Seqstack<T>::isFull()
{
    return (this->top==this->maxsize);
}

//获取栈中元素个数
template<typename T>
int Seqstack<T>::length()
{
    return this->top;
}

//入栈
template<typename T>
bool Seqstack<T>::push(const T &x)
{
    if(this->isFull())    return false;
    element[top++]=x;
    return true;
}
```

```
//获取栈顶元素
template<typename T>
bool Seqstack<T>::top_(T& a)
{
    if(this->isempty())   return false;
    a= element[top-1];
    return true;
}

//出栈
template<typename T>
bool Seqstack<T>::pop()
{
    if(this->isempty())   return false;
    top--;
    return true;
}
```

循环队列中主要操作的实现：

```
//构造函数
template<typename T>
CirQueue<T>::CirQueue()
{
    this->front=0;  //队首指针
    this->rear=0;   //队尾指针
    this->count=0;  //元素个数
    /*
        这里的count记录队列中元素的个数，
        不用浪费一个空间，
        通过判断count=0还是count=MaxQueueSize即可
    */
}

//析构函数
template<typename T>
CirQueue<T>::~CirQueue()
{}

//判断循环队列是否为空
template<typename T>
bool CirQueue<T>::isempty()
{
    return this->count==0;
}

//判断循环队列是否为满
template<typename T>
bool CirQueue<T>::isfull()
{
    return this->count==MaxQueueSize;
}

//获取循环队列中元素个数
```

```
template<typename T>
int CirQueue<T>::getlength()
{
    return this->count;
}

//入队
template<typename T>
bool CirQueue<T>::enqueue(const T& x)
{
    if(this->isfull())  return false;
    data[rear]=x;
    rear=(rear+1)%MaxQueueSize;
    count++;
    return true;
}

//出队
template<typename T>
bool CirQueue<T>::dequeue(T& x)
{
    if(this->isempty())  return false;
    x=data[front];
    front=(front+1)%MaxQueueSize;
    return true;
}

//取队首元素
template<typename T>
bool CirQueue<T>::getfront(T &x)
{
    if(this->isempty())  return false;
    x=data[front];
    return true;
}
```

1.4 调试分析

（1）调试过程中遇到的主要问题及解决过程

在调试过程中对循环队列判空和判满操作起初有些疑惑，经过考虑采用了余老师书上的第三种方法，即在类中用一个私有成员count来表示元素个数，这样就可以通过count=0或count=maxsize来判断。

（2）程序代码的质量

在一定的测试数据集上表现正确。在一些异常情况下也能够处理，如在队空时出队等等，这样会返回一个false值。

1.5 使用说明

顺序栈模板类的实现包含在一个独立的C++头文件"SeqStack.h"中,要在程序中使用这个类只需包含该头文件即可；同样的，循环队列模板类的实现包含在一个独立的C++头文件"CirQueue.h"中,要在程序中使用这个类只需包含该头文件即可。

1.6 测试结果

测试数据见1.1（4）,测试输出结果如下图所示：

```
Seqstack<int>mystack(5);
cout<<mystack.isempty()<<endl;
cout<<mystack.isFull()<<endl;
mystack.push(1);
mystack.push(2);
mystack.push(3);

cout<<mystack.isempty()<<endl;
cout<<mystack.isFull()<<endl;
cout<<mystack.length()<<endl;

mystack.push(1);
mystack.push(2);
mystack.push(3);

int n;
while(mystack.top_(n))
{
    mystack.pop();
    cout<<n<<endl;
}
cout<<mystack.isFull()<<endl;
cout<<mystack.length()<<endl;

return 0;
```

```
1
0
0
0
3
2
1
3
0
0
```

2.商品货架管理

2.1 需求分析

（1）输入的形式和输入值的范围

输入一个要添加的商品的日期，包括年、月、日。三者均为整型变量，年的范围考虑为[2010,2022],月的范围为[1,12]，日的范围为[1,30]。

（2）输出的形式

为了查看货架上货物的摆放顺序，我们可以将货架上货物的日期从栈顶到栈底依次输出。

（3）程序所能达到的功能

可以实现商品货架的管理过程，每次新增货物时，都能将货架上的商品的生产日期按顺序排列，栈顶商品的生产日期最早，栈底商品的生产日期最晚。

（4）测试数据

- 初始化一个容量为3空货架
- 向货架上添加生产日期为2021.1.1日的货物

- 向货架上添加生产日期为2021.12.1日的货物
- 查看货架上的货物信息
- 向货架上添加生产日期为2022.1.1日的货物
- 查看货架上的货物信息
- 向货架上添加生产日期为2022.4.11日的货物

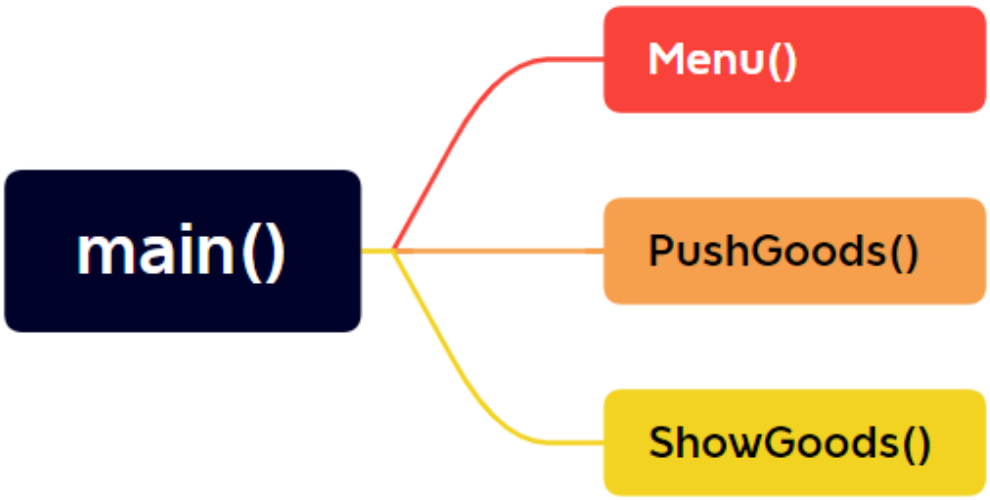
2.2 概要设计

（1）抽象数据类型的定义

本实验用到了1中的顺序栈，见1.1 需求分析

（2）主程序流程和程序模块之间的关系

在主函数中调用菜单函数、放置货物、以及展示货物函数，如下图所示：



2.3 详细设计

定义了一个货物类：

```
//货物类
class Goods
{
private:
    int year;
    int month;
    int day;
public:
    bool setdate(int year,int month,int day);   //设置日期
    bool operator>(Goods d);      //重载比较运算符，用于日期间的比较
    void showdate();             //展示日期
};
```

数据成员有整型变量年月日三个私有成员。

成员函数的实现：

```
//设置日期
bool Goods::setdate(int year,int month,int day)
{
    if(year<2010||year>2022||month<1||month>12
    ||day<1||day>30)
        return false;
    this->year=year;
    this->month=month;
    this->day=day;

    return true;
}

//重载比较运算符，用于日期间的比较
bool Goods::operator>(Goods d)
{
    return (this->year>d.year)
        || (this->year==d.year)&&(this->month>d.month)
        || (this->year==d.year)&&(this->month==d.month)&&(this->day>d.day);
}

//展示日期
void Goods::showdate()
{
    cout<<this->year<<"年"<<this->month<<"月"<<this->day<<"日"<<endl;
}
```

供主函数调用的三个函数的实现：

```
//显示菜单
void showMenu()
{
    cout<<"*****欢迎来到商品货架管理系统*****"<<endl;
    cout<<"1.向货架上增添新货物"<<endl;
    cout<<"2.查看当前货架上货物信息"<<endl;
    cout<<"0.退出系统"<<endl;
    cout<<"请输入选择：";
}
//货架上放置货物
void pushGoods(Seqstack<Goods>& shelf,Goods g)
{
    Seqstack<Goods>temp(shelf.length());  //临时栈，用于倒货架
    Goods gd;
    //有栈顶元素并且日期比g小（就是比g旧），那么让栈顶元素出栈，放入临时栈中
    while(shelf.top_(gd)&&g>gd)
    {
        temp.push(gd);
        shelf.pop();
    }
    shelf.push(g);
    //然后将临时栈中的元素再全部放入shelf货架上

    while(!temp.isempty())
    {
```

```
            temp.top_(gd);
            temp.pop();
            shelf.push(gd);
        }
    }

    //在货架上从前到后（从栈顶到栈底）展示货物
    void showGoods(Seqstack<Goods>& shelf)
    {
        if(shelf.isempty())
        {
            cout<<"该货架为空"<<endl;
            return ;
        }
        cout<<"该货架上商品的生产日期依次为：";
        Goods gd;
        while(!shelf.isempty())
        {
            shelf.top_(gd);
            gd.showdate();
            shelf.pop();
        }

    }
```

2.4 调试分析

（1）程序代码的质量

在一定的测试数据集上表现正确。在一些异常情况下也能够处理，如在货架满时添加货物会提示货架已满。

（2）程序优化考虑

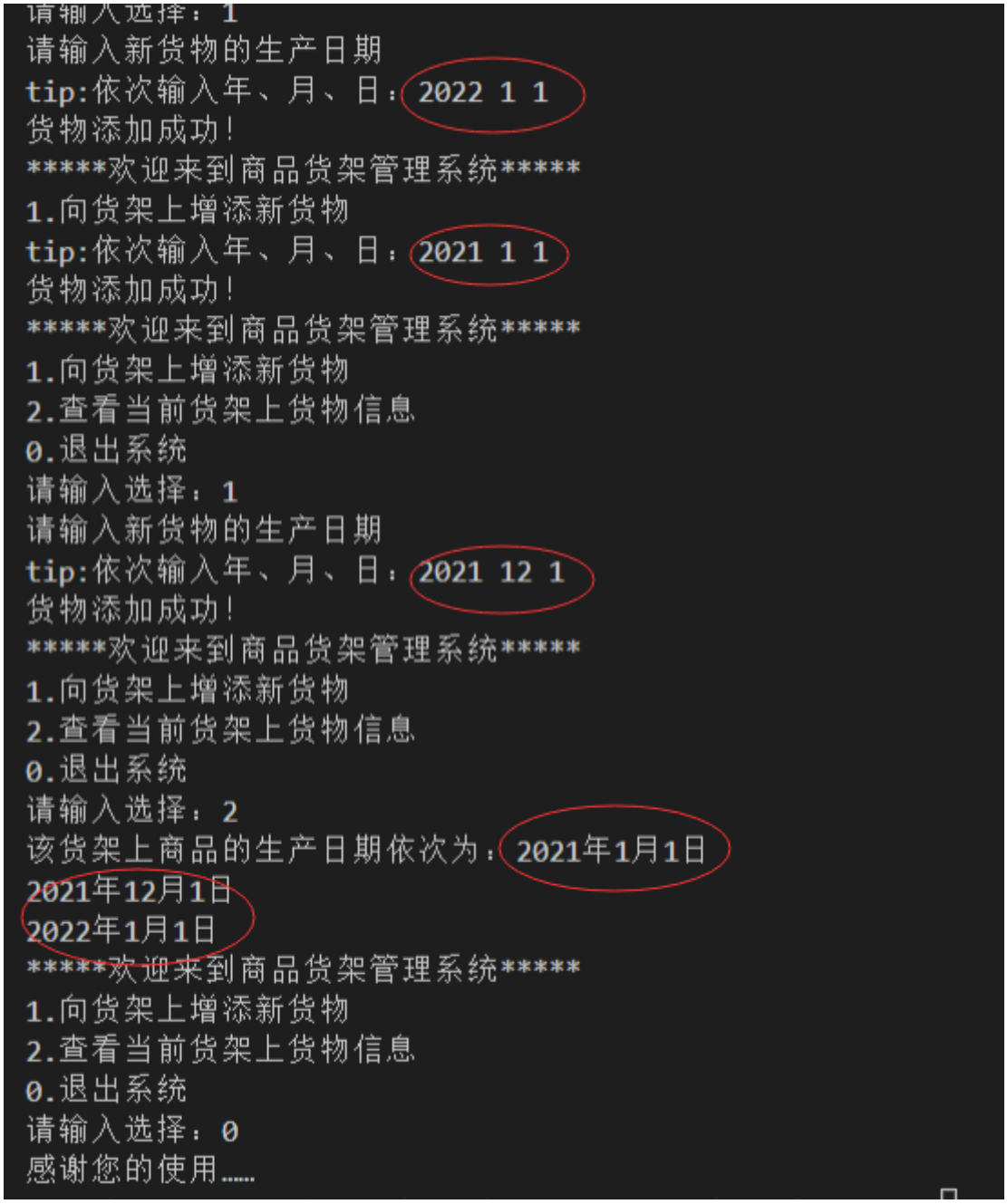
在进行“倒货架”操作时需要一个辅助栈，这里我是新开了一个辅助栈，可以考虑双向栈，这样可以节省一定的空间。

2.5 使用说明

货物类的实现及放置新货物、查看货物信息的函数均包含在一个独立的C++头文件"Goods.h"中，通过引入该头文件，即可完成相关操作。

2.6 测试结果

测试结果如下图所示：



3.停车场管理

3.1 需求分析

（1）输入的形式和输入值的范围

输入多个“（mode,tag,time）”字符串形式，每一个字符串包括三个数据项。其中：

mode表示汽车“到达”或“离去”信息，可能的取值有“A，D，E”，A表示汽车到达，D表示汽车离去、E表示输入结束。

tag表示汽车的车牌号，这里使用自然数表示，1，2.....

time表示离开或者到达的时刻。

（2）输出的形式

当输入状态结束后，进入查询状态，然后输如要查询的汽车的车牌号，如果是到达的车辆，则输出其在停车场内或便道上的停车位置；如果是离去的车辆，则输出其在停车场内的停留时间和应交的费用（在便道上停留的时间不收费）。

（3）程序所能达到的功能

- 借助栈和队列数据结构来模拟停车场管理

（4）测试数据

- 输入数据：

  (A,1,5)(A,2,10)(D,1,15)(A,3,20)(A,4,25)(A,5,30)(D,2,35)(D,4,40)(E,0,0)

- 依次查询车辆信息：

  A、B、C、D、E、F

3.2 概要设计

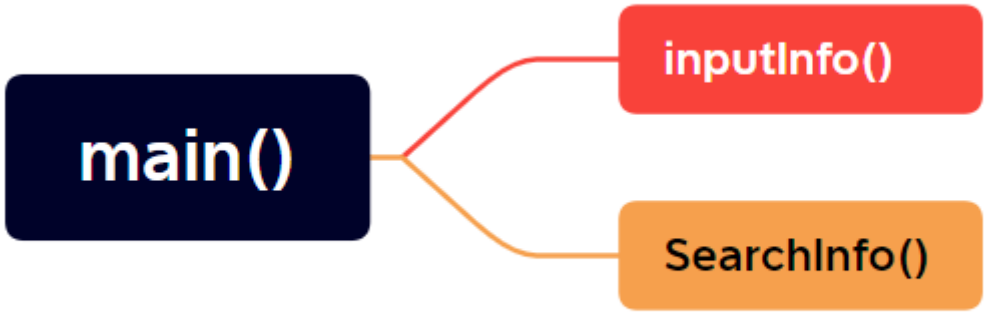
（1）抽象数据类型的定义

本实验除了用到了1中的顺序栈，见1.1 需求分析。

还用到了链队列。

（2）主程序流程和程序模块之间的关系

在主函数中调用输入车辆信息、查询车辆信息函数，如下图所示：



3.3 详细设计

设计了一个车辆类：

```
//汽车类
class Car
{
    int tag;            //汽车的牌照
    int arrival;        //汽车到达时间
    int depatrue;       //汽车离开时间
    public:
    Car():tag(0),arrival(0),depatrue(0){};  //构造函数
    Car(int tag,int arrival):tag(tag),arrival(arrival),depatrue(0){}
    bool operator>(Car c)   //运算符重载，用于比较车到达快慢
    {return this->arrival>c.arrival;}
    int gettag(){return tag;};   //获取牌照
    void setarrival(int arrival)    //设置到达时间
    {this->arrival=arrival;}
    void setdepatrue(int depatrue)    //设置离开时间
```

```
    {this->depatrue=depatrue;}
    int getduration()       //获取停留时间
    {return depatrue-arrival;}
};
```

其中包括汽车牌照、汽车到达时间、汽车离开时间私有数据成员。

成员函数包括构造函数、析构函数、>运算符的重载（用于比较汽车到达快慢），获取牌照、设置到达时间、设置离开时间、获取停留时间（用于计算费用）。

InputInfo()和SearchInfo()函数的实现：

```
//输入车辆信息
void inputInfo(Seqstack<Car>&parking,LinkQueue<Car>&sidewalk,vector<Car>&away)
{
    cout<<"请输入车辆状态信息："<<endl;
    while(true)
    {
        char p1,p2,cm,mode;
        int car_tag;
        int time;
        cin>>p1>>mode>>cm>>car_tag>>cm>>time>>p2;
        //E表示程序终止
        if(mode=='E')
        {
            cout<<"输入状态已结束……"<<endl;
            break;
        }
        //A表示汽车到达
        else if(mode=='A')
        {
            Car newc(car_tag,time);
            //栈不满就入栈
            if(!parking.isFull())
                parking.push(newc);
            //栈满就入队
            else
                sidewalk.enqueue(newc);
        }
        //D表示汽车离去
        else if(mode=='D')
        {
            //先要找到目标元素
            Seqstack<Car>temp(parking_size);    //辅助栈
            Car a_car;
            while(parking.top_(a_car)&&a_car.gettag()!=car_tag)
            {
                parking.pop();
                temp.push(a_car);
            }
            //离开的元素将其放入away数组中
            parking.pop();
            a_car.setdepatrue(time);
            away.push_back(a_car);
            //再将辅助栈中的元素重新放入栈中
```

```
            while(temp.top_(a_car))
            {
                temp.pop();
                parking.push(a_car);
            }
            //如果链上有车等候，就让其进栈
            if(!sidewalk.isempty())
            {
                sidewalk.dequeue(a_car);
                a_car.setarrival(time);
                parking.push(a_car);
            }
        }
    }
}

//查询车辆信息
void
SearchInfo(Seqstack<Car>&parking,LinkQueue<Car>&sidewalk,vector<Car>&away,const
int price)
{
    while(1)
    {
        start:
        int target;
        cout<<"**************查询页面**************"<<endl;
        cout<<"请输入要查询的车牌号(输入0退出该界面):";
        cin>>target;
        if(!target) return;
        //在数组中（已经离开）查询
        for(int i=0;i<away.size();i++)
        {
            if(target==away[i].gettag())
            {
                cout<<"该车已离开停车场,停车时间为"<<away[i].getduration()<<",应交费用
为"<<away[i].getduration()*price<<endl;
                goto start;
            }
        }
        //在栈中（停车场）查询
        Car a_car;
        int position=1;      //表示位置的变量
        Seqstack<Car>tempstack(parking_size);     //临时栈
        while(parking.top_(a_car)&&a_car.gettag()!=target)
        {
            parking.pop();
            tempstack.push(a_car);
            position++;
        }
        //在栈中找到了
        if(!parking.isempty())
        {
            cout<<"该车在停车场,停在第"<<parking_size+1-position<<"个车位（从北向南
数）"<<endl;
            //再将辅助栈中的元素挪到parking栈中
```

```
            while(tempstack.top_(a_car))
            {
                tempstack.pop();
                parking.push(a_car);
            }
            goto start ;
        }
        while(tempstack.top_(a_car))
        {
            tempstack.pop();
            parking.push(a_car);
        }

        //在队列（便道）中找
        LinkQueue<Car> tempq;   //辅助队列
        position=1;
        while(sidewalk.dequeue(a_car)&&a_car.gettag()!=target)
        {
            tempq.enqueue(a_car);
            position++;
        }
        //在队列中找到了
        if(!sidewalk.isempty())
        {
            cout<<"该车在便道上，排在第"<<position<<"个元素的位置"<<endl;
            //再把便道剩余的车辆都移入辅助队列中
            while (sidewalk.dequeue(a_car))
                tempq.enqueue(a_car);
            //再将辅助队列的车辆移入便道上
            while (tempq.dequeue(a_car))
                sidewalk.enqueue(a_car);
            continue ;
        }
        while (tempq.dequeue(a_car))
                sidewalk.enqueue(a_car);
        //在队列中也没有找到
        cout<<"没有查询到此车相关信息"<<endl;
    }
}
```

3.4 调试分析

（1）调试过程中遇到的主要问题及解决过程

- 在调试过程中发现在输入信息结束后，查询时总是找不到相关车辆的信息，后来才发现是犯了个很明显的错误，就是在输入信息是传入的参数是数组而不是引用，这样就导致了实参数组的值并不会由于函数而改变。
- 另一个问题就是在自己编写好链队列的模板后并没有严格地去测试，这样在引入了这个头文件出现问题后在本程序中花费了很大力气都没有找出问题，后来才发现是头文件中链队列实现的问题，在进行入队操作时，没有给新申请的节点赋值。
- 再进行出队操作时可能引起尾指针的改变，再刚开始编写代码时未考虑到这一点，因此在析构或者查找时经常遇到错误。

（2）程序代码的质量

在一定的测试数据集上表现正确。输入要查询的车辆牌号会输出相关信息。如果是到达的车辆，则输出其在停车场内或便道上的停车位置；如果是离去的车辆，则输出其在停车场内的停留时间和应交的费用（在便道上停留的时间不收费）。如果是该车牌号从未入队入栈，则输出未查询到该车信息。

（3）心得体会

- 在编写类模板时，编写完成后一定要经过测试后，确保没有什么明显问题再供其他cpp文件引入并使用。
- 类模板具有复用性，编写完一个类模板后常常可以供几个程序一起使用。
- 在遇到一些存储特殊的数据类型时，如货物、汽车，可以自己定义一个类，然后通过该类实例化类模板进行使用。

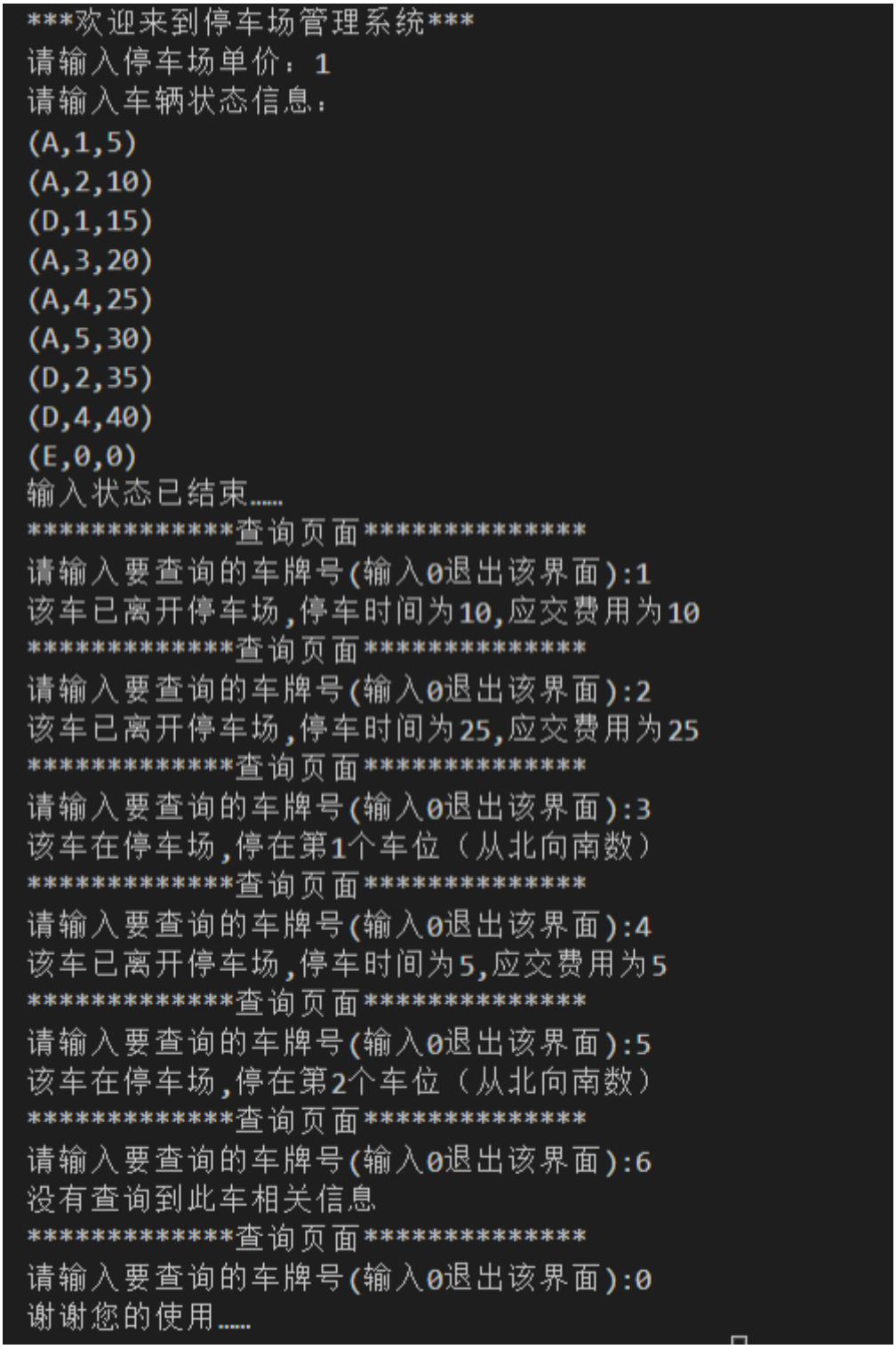
3.5 使用说明

在”停车场管理“的系统中需要用到顺序栈和链队列，因此引入”seqstack.h“,"LinkQueue"头文件，这两个头文件中分别实现了顺序栈与链队列的类模板。

然后在程序中先进行汽车信息的输入，再进行汽车信息的查询即可。

3.6 测试结果

测试数据见3.2（4），测试结果如下图所示：



4.附录

各程序源代码随本实验报告电子版一起打包，存放在src子目录。

文件清单如下：

CirQueue.h->循环队列的实现

goods_shelf.cpp->商品货架管理程序代码

goods_shelf.exe->商品货架管理程序

Goods.h->货物类的实现

LinkQueue.h->链队列的实现

parking.cpp->停车场管理程序代码

parking.exe->停车场管理程序

seqstack.h->顺序栈的实现

test1.cpp->顺序栈的测试程序代码

test1.exe->顺序栈的测试程序

test2.cpp->循环队列的测试程序代码

test2.exe->循环队列的测试程序代码

test3.cpp->链队列的测试程序代码

test3.exe->链队列的测试程序