#include<iostream>

using namespace std;

const int MAXSIZE = 100;

template<class ElemType>

class SqQueue {

private:

ElemType\* elem; // 存储空间基址

int front; // 队头指针，注意队头指针和队尾指针都不是指针类型，它只是一个记录对应标号的int型

int rear; // 队尾指针

int maxSize; //注意：maxSize-1才是允许的最大存储容量，以后用这个ADT的时候应当考虑到这一点

public:

//初始化顺序队列

SqQueue(int ms = 20) {

maxSize = ms;

elem = new ElemType[maxSize];

front = rear = 0;

}

//拷贝构造函数

SqQueue(const SqQueue<ElemType>& Q);

//拷贝赋值运算符

SqQueue<ElemType>& operator=(const SqQueue& Q);

//删除顺序队列

~SqQueue() { QueueDestroy(); /\*cout << "调用了析构函数" << endl; \*/}

//将顺序队列置为空

bool QueueClear();

//设置顺序栈的长度

bool setLength(int len);

//判断顺序队列是否为空

bool isEmpty() const { return front == rear; }

//判断顺序队列是否为满

bool isFull() const;

//返回队列中现有元素的个数

int getLength()const;

//返回队列中最多可以容纳的元素个数

int capacity() const{

return maxSize;

}

//用e返回某一位置的元素:

//注意：不建议在用户调用，只是用于实现内部的拷贝功能时需要用到，否则的话会破坏队列的FIFO的特性

//注意：使用这个函数必须确保队列中的值不是未定义的，否则会引发很严重的后果

bool getElem(int i, ElemType& e)const;

//用e返回队头元素

bool getFront(ElemType& e);

//入队

bool enQueue(const ElemType& e);

//出队

bool deQueue(ElemType& e);

//销毁顺序队列

bool QueueDestroy();

//顺序队列最大存储空间加倍

bool doubleSpace();

//队列遍历函数

void traverse();

//测试用：必须确保每一个操作之后的front和rear都指到正确的位置，否则可能会引发严重后果

/\*int getRear() {

return rear;

}

int getFront() {

return front;

}\*/

};

//拷贝构造函数

template<class ElemType>

SqQueue<ElemType>::SqQueue(const SqQueue<ElemType>& Q) {

maxSize = Q.capacity();

elem = new ElemType[maxSize];

front = rear = 0;

if (!Q.isEmpty()) {

const int L = Q.getLength();

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

ElemType e;

if (Q.getElem(i + 1, e)) {

\*(elem + i) = e;//注意虽然从总体上来说我们用到了循环数组的思想，但是从本质上来说

//数组依然是笔直排列且非循环的，而这里的拷贝只是使得其中的元素以及元素的相对位置一样，

//但是每个两个队列相同元素在各自数组中所处的相对位置确是不同的

//这样拷贝所得的队列的front一定为0，其rear值一定为Q.getLength()

}

}

rear = L;

}

}

//拷贝赋值运算符

template<class ElemType>

SqQueue<ElemType>& SqQueue<ElemType>::operator=(const SqQueue& Q) {

maxSize = Q.capacity();

elem = new ElemType[maxSize];

front = rear = 0;

if (!Q.isEmpty()) {

const int L = Q.getLength();

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

ElemType e;

if (Q.getElem(i + 1, e)) {

\*(elem + i) = e;//注意虽然从总体上来说我们用到了循环数组的思想，但是从本质上来说

//数组依然是笔直排列且非循环的，而这里的拷贝只是使得其中的元素以及元素的相对位置一样，

//但是每个两个队列相同元素在各自数组中所处的相对位置确是不同的

//这样拷贝所得的队列的front一定为0，其rear值一定为Q.getLength()

}

}

rear = L;

}

//cout << "调用了拷贝赋值运算符函数" << endl;

return \*this;

}

//将顺序队列置为空

template<class ElemType>

bool SqQueue<ElemType>::QueueClear() {

front = rear=0;//实际上只是改变了队头和队尾指针的指向，并没有真正地将队列中不同位置的值删除，要想真正地删除

return true;//必须对值进行一定的覆盖

}

//设置顺序栈的长度

//注意：使用这个函数的时机应当是在队列中还没有元素的时候，我们对其长度进行一个提前规划

template<class ElemType>

bool SqQueue<ElemType>::setLength(int len) {

delete[]elem;

maxSize = len;

elem = new ElemType[maxSize];

front = rear = 0;

if (elem == NULL) return false;

else return true;

}

//判断顺序队列是否为满：（这里采用的策略是少使用顺序表的一个位置去存储元素，使得当rear的后一位为front时即为队列满的标志）

template<class ElemType>

bool SqQueue<ElemType>::isFull() const {

if ((rear + 1) % maxSize == front) {

return true;

}

else {

return false;

}

}

//返回队列中现有元素的个数

template<class ElemType>

int SqQueue<ElemType>::getLength()const {

return (rear - front + maxSize) % maxSize;

}

//用e返回某一位置的元素:

//注意：不建议在用户调用，只是用于实现内部的拷贝功能时需要用到，否则的话会破坏队列的FIFO的特性

//注意：使用这个函数必须确保队列中的值不是未定义的，否则会引发很严重的后果

template<class ElemType>

bool SqQueue<ElemType>::getElem(int i, ElemType& e)const {

if (this->isEmpty()) {

return false;

}

else if (i-1 == rear || i > this->getLength() || i <= 0) {

return false;

}

else {

e = \*(elem +(front + i - 1)%maxSize);//一定要时刻注意这个循环

return true;

}

}

//用e返回队头元素

template<class ElemType>

bool SqQueue<ElemType>::getFront(ElemType& e) {

if (this->isEmpty()) {

return false;

}

else {

e = \*(elem + front);

return true;

}

}

//入队

template<class ElemType>

bool SqQueue<ElemType>::enQueue(const ElemType& e) {

if (this->isFull()) { return false; }

else {

\*(elem + rear) = e;

rear = (rear + 1) % maxSize;

//cout << "入完队后的front和rear的值为:" << front << ' ' << rear << endl;

return true;

}

}

//出队

template<class ElemType>

bool SqQueue<ElemType>::deQueue(ElemType& e) {

if (this->isEmpty()) {

return false;

}

else {

e = \*(elem + front);

front = (front + 1) % maxSize;

//cout << "出完队后front和rear的值分别为:" << front << ' ' << rear << endl;

return true;

}

}

//销毁顺序队列

template<class ElemType>

bool SqQueue<ElemType>::QueueDestroy() {

if (!elem) {

return false;

}

else {

delete[] elem;

return true;

}

}

//顺序队列最大存储空间加倍，可以将其应用于队列中已经有元素的场合

template<class ElemType>

bool SqQueue<ElemType>::doubleSpace() {

if (!elem) {

return false;

}

else {

int L = this->getLength();

if (L == 0) {

delete[] elem;

maxSize = 2 \* maxSize;

elem = new ElemType[maxSize];

front = rear = 0;

}

else {//表明原来这个队列中有元素，要先对元素进行一个拷贝

ElemType\* tmp = elem;

maxSize = 2 \* maxSize;

elem = new ElemType[maxSize];

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

\*(elem + i) = \*(tmp + front + i);

}

delete[] tmp;//这里一定要记得把原来的空间delete掉

}

if (!elem) return true;

}

}

//队列遍历函数:

template<class ElemType>

void SqQueue<ElemType>::traverse() {

if (this->isEmpty()) { return; }

else {

const int L = this->getLength();

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

cout << \*(elem + (front + i) % maxSize) << ' ';

}

//cout << endl;

//cout << "这个队列的长度为:" << L;

}

}