

**NOTE: circq.h is the same for both programs.**

/\* circq.h

\*

\* Declares CircularQ data type and associated operators

\*

\* Johnathan Lee CSCI 1107

\* Lab 10 Due 04/03/18

\*

\*

\* MODIFIED FOR Lab 12 on 04/10/18

\*/

#ifndef CIRCQ\_H

#define CIRCQ\_H

#include <iostream>

using namespace std;

/// \class CircularQ

/// \brief A Queue implemented with a circular linked list.

/// \note Uses sizeof(QueueElement) + sizeof(POINTER) per element.

template <typename T>

class CircularQ {

public:

/// \brief Default (Empty) constructor

/// \param None

/// \post \*this is initialized with absolutely no elements.

**CircularQ**();

/// \brief Copy constructor:

/// \param orig The CircularQ to copy elements from

/// \post All elements from orig are now ALSO contained in \*this.

/// \note This does NOT transfer ownership, it is a deep copy, orig is

/// unchanged.

**CircularQ**(const CircularQ& orig);

/// \brief Destructor

/// \post All elements in this queue are deleted

~**CircularQ**();

/// \brief Assignment

/// \param rhs The queue to copy elements from.

/// \post All elements are deep copied to \*this. See copy constructor.

const CircularQ& *operator*=(const CircularQ& rhs);

/// \brief Check if the queue is empty.

/// \return Whether there are no elements in the queue

bool **empty**() const;

/// \brief Add a value to the end of the queue.

/// \param value The value to add into the queue.

/// \post value is now at the end of the queue.

void **enqueue**(const T& value);

/// \brief Outputs \*this

/// \param out The stream to write to

/// \post The contents of this list are written to out, space delimited.

void **display**(ostream& out) const;

/// \brief Get the front of the queue.

/// \returns The first element in the queue. (I.E been there the longest).

T **front**() const;

/// \brief Remove the element at the front of the queue.

/// \post Current this->front() is no longer in memory.

void **dequeue**();

private:

/// \class Node

/// \brief Internal storage for data elements. Not relevent outside

/// CircularQueue class internals.

class Node {

public:

/// \brief Constructor

/// \param value The data value to assign to data

/// \param link The Node \*this should link to.

**Node**(T value, Node\* link = NULL) : data(value), next(link) {

}

T data; ///>! Our data element

Node\* next; ///>! The next link in the chain

};

typedef Node\* NodePointer;

NodePointer myBack; ///>! Our view into the list. Use myBack->next for front

};

/// \brief Output stream operator for CircularQ

/// \param out The stream to write to.

/// \param q The queue to display

/// \post Writes all elements in q to out. See CircularQ::display for more.

template <typename T>

ostream& *operator*<<(ostream& out, const CircularQ<T>& q);

*template*<typename T>

CircularQ<T>::CircularQ() {

myBack = NULL;

}

// Since it's just 2 statements, should really be inlined...

*template*<typename T>

CircularQ<T>::CircularQ(const CircularQ& orig) {

myBack = NULL;

// Since it's the same code and the (this == &rhs) still works...

\*this = orig;

}

*template*<typename T>

CircularQ<T>::~**CircularQ**() {

if (myBack) {

Node\* cur = myBack->next;

myBack->next = NULL;

while (cur != myBack) {

Node\* old = cur;

cur = cur->next;

delete old;

}

delete myBack; // Putting out here to avoid problems with size() == 1.

}

myBack = NULL; // Accomodate Microsoft

}

*template*<typename T>

const CircularQ<T>& CircularQ<T>::operator=(const CircularQ& rhs) {

if (this != &rhs) {

*this*->~CircularQ(); // Delete old list

if (!rhs.empty()) {

Node\* curRhs = rhs.myBack->next;

// Use our methods - think deeply of simple things.

// (I'm gonna have nightmares about that phrase)

do {

enqueue(curRhs->data);

curRhs = curRhs->next;

} while (curRhs != rhs.myBack->next);

}

}

return \*this;

}

*template*<typename T>

bool CircularQ<T>::empty() const {

return myBack == NULL; // If myBack is null we have no nodes, so empty.

}

*template*<typename T>

void CircularQ<T>::enqueue(const T& value) {

if (myBack) { // !empty

Node\* newNode = new Node(value, myBack->next);

myBack->next = newNode;

myBack = newNode;

}

else {

myBack = new Node(value, myBack);

// Because assigning it in the constructor would assign to NULL.

// You don't wanna know how long that one took me.

myBack->next = myBack;

}

}

*template*<typename T>

void CircularQ<T>::display(ostream& out) const {

if (myBack) {

Node\* cur = myBack->next;

do {

out << cur->data << " ";

cur = cur->next;

} while (cur != myBack->next);

}

else {

// To help distinguish between no output and outputting blank queue.

/\*//\*/ out << "EMPTY\_CIRCULAR\_QUEUE ";

}

}

*template*<typename T>

T CircularQ<T>::front() const {

return myBack->next->data;

}

*template*<typename T>

void CircularQ<T>::dequeue() {

if (myBack->next == myBack) {

delete myBack;

myBack = NULL;

}

else {

Node\* oldFront = myBack->next;

myBack->next = oldFront->next;

delete oldFront;

}

}

*template*<typename T>

ostream& *operator*<<(ostream& out, const CircularQ<T>& q) {

q.display(out);

return out;

}

#endif

/// \file circqtester.cpp

/// \brief Testing driver for the CircularQueue class

/// \author Johnathan Lee for CSCI 1107, Lab 12, Proj 7.2

/// \date 04/10/18

#include <cassert>

#include <iostream>

#include <sstream>

#include <string>

#include "circq.h"

using namespace std;

/// \brief Helper function to say WHAT we're testing.

void **testing**(string what) {

cout << "Testing " << what << "...\n";

}

/// \brief Helper function to say we passed a test. (assert didn't fail).

/// \note Only gets called if the assert before it passed.

void **passed**() {

cout << "PASSED!\n\n\n";

}

int **main**() {

testing("Constructor and empty");

CircularQ<int> intQueue;

// Constructor should make an empty queue.

assert(intQueue.empty());

passed();

testing("Enqueue, Dequeue, Front, and empty");

for (int i = 10; i < 100; i += 10)

intQueue.enqueue(i);

for (int i = 10; i < 100; i += 10) {

// Make sure each went in properly

assert(intQueue.front() == i);

intQueue.dequeue();

}

// Should be empty after dequeing all elements

assert(intQueue.empty());

passed();

{

testing("Display");

for (int i = 10; i < 100; i += 10)

intQueue.enqueue(i);

stringstream ss;

intQueue.display(ss);

// display should output each followed by 2 spaces

cout << "Output was: " << ss.str() << endl;

assert(ss.str() == "10 20 30 40 50 60 70 80 90 ");

passed();

stringstream

ss2; // Since for some reason ss.clear doesn't work properly...

testing("operator<<");

ss2 << intQueue;

cout << "Output was: " << ss2.str() << endl;

// display should output each followed by 2 spaces

assert(ss2.str() == "10 20 30 40 50 60 70 80 90 ");

passed();

}

testing("Destructor");

{

CircularQ<int> desTester;

desTester.enqueue(10);

desTester.dequeue();

// Now destruct. You'll know if it works using magic.

}

passed();

{

testing("Copy constructor");

CircularQ<int> q1;

for (int i = 10; i < 100; i += 10)

q1.enqueue(i);

CircularQ<int> q2(q1);

for (int i = 10; i < 100; i += 10) {

// Make sure each went in properly

assert(q2.front() == i);

q2.dequeue();

}

passed();

}

{

testing("Assignment operator");

CircularQ<int> q1, q2;

for (int i = 10; i < 100; i += 10)

q1.enqueue(i);

q2 = q1;

for (int i = 10; i < 100; i += 10) {

// Make sure each went in properly

assert(q2.front() == i);

q2.dequeue();

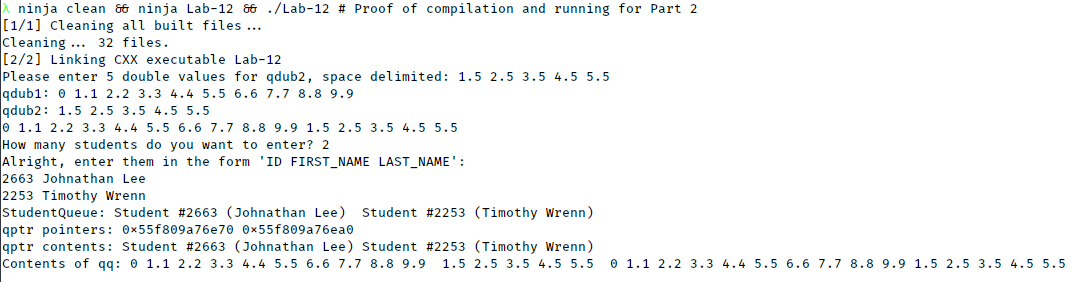
}

passed();

}

return 0;

} // end main

****

/// \file Project-7-2-Part2.cpp

/// \brief Testing operations on CircularQ, as well as a student class.

/// \author Johnathan Lee for CSCI 1107, Project 7.2

/// \date 04/10/18

///

/// \note Code is organized as follows:

/// INCLUDES

/// NAMESPACES

/// CLASS DECS (This part has the docs)

/// FUNCTION PROTOS (This part has the docs)

/// MAIN DRIVER

/// FUNCTION DEFS

/// CLASS DEFS

#include <iostream>

#include <sstream>

#include <string>

#include "circq.h"

using namespace std;

/// \class Student

/// \brief A class to contain student information.

class Student {

public:

/// \brief Default and explicit constructor

/// \param theID The student's ID number

/// \param theName The student's name, first and last.

**Student**(int theID = 0, string theName = "");

/// \brief Getter for this->id.

/// \post No special post conditions

int **getID**() const;

/// \brief Setter for this->id

/// \post No special post conditions beyond setting id.

void **setID**(int theID);

/// \brief Getter for this->name.

/// \post No special post conditions

string **getName**() const;

/// \brief Setter for this->name

/// \post No special post conditions beyond setting name

void **setName**(string theName);

/// \brief Display this student as a string

/// \param out The stream to write to.

/// \post \*this is written to out in the format #<this->id> (<this->name>)

void **display**(ostream& out) const;

/// \brief Set this student's values to those contained in a stream.

/// \param in The stream to read from

/// \post \*this contains values from in that were in the format ID FIRST\_NAME

/// LAST\_NAME

/// \note Takes in 2 words for the name, in format FIRST LAST

void **parse**(istream& in);

private:

int id; ///>! The student ID, typically > 0

string name; ///>! The student's first and last name.

};

/// \brief Input a student

/// \param in Stream to read from

/// \param student Student to parse into

/// \post Consumed characters from in; Student contains information found in in.

istream& *operator*>>(istream& in, Student& student);

/// \brief Output a student

/// \param out Stream to write to

/// \param student Student to output

/// \post Student info has been output. See Student::display for more info.

ostream& *operator*<<(ostream& out, const Student& student);

/// \brief Join 2 circular queues.

/// \tparam T the type shared by both CircularQ's

/// \param first The first queue

/// \param second The second queue

/// \returns A queue containing all elements of first followed by all of second.

template <typename T>

CircularQ<T> join(CircularQ<T> first, CircularQ<T> second);

int main() {

// Note: I have put comments denoting each section in the lab manual.

// 1.a

CircularQ<double> qdub1, qdub2, qdub;

CircularQ<string> qstr;

qstr.enqueue("a few strings of your choosing");

qstr.enqueue("a string not of your choosing.");

qstr.enqueue("COBOL is the bane of all.");

qstr.enqueue("And Java isn't much better");

for (int i = 0; i < 10; i++)

qdub1.enqueue(i \* 1.1);

cout << "Please enter 5 double values for qdub2, space delimited: ";

for (int i = 0; i < 5; i++) {

double qEl;

cin >> qEl;

qdub2.enqueue(qEl);

}

// 1.b

cout << "qdub1: " << qdub1 << endl << "qdub2: " << qdub2 << endl;

// 2

qdub = join(qdub1, qdub2);

cout << qdub;

// 3

CircularQ<Student> studentQueue;

int numStudents = 0;

cout << "\nHow many students do you want to enter? ";

cin >> numStudents;

cout << "Alright, enter them in the form 'ID FIRST\_NAME LAST\_NAME':\n";

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

Student newStudent;

cin >> newStudent;

studentQueue.enqueue(newStudent);

}

cout << "StudentQueue: " << studentQueue;

// 4

CircularQ<Student\*> qptr;

CircularQ<Student> studentCopy = studentQueue;

while (!studentCopy.empty()) {

qptr.enqueue(new Student(studentCopy.front()));

studentCopy.dequeue();

}

cout << "\nqptr pointers: " << qptr;

cout << "\nqptr contents: ";

while (!qptr.empty()) {

cout << \*qptr.front();

qptr.dequeue();

}

// 5

CircularQ<CircularQ<double>> qq;

qq.enqueue(qdub1);

qq.enqueue(qdub2);

qq.enqueue(qdub);

cout << "\nContents of qq: " << qq << endl;

}

template <typename T>

**CircularQ**<**T**> **join**(**CircularQ**<**T**> **first**, **CircularQ**<**T**> **second**) {

// DEBUG: cout << endl << "joining " << first << " with " << second << endl;

while (!second.empty()) {

first.enqueue(second.front());

second.dequeue();

}

return first;

}

// Student methods and helper funcs:

ostream& *operator*<<(ostream& out, const Student& student) {

student.display(*out*);

}

istream& *operator*>>(istream& in, Student& student) {

student.parse(*in*);

}

Student::Student(int theID, string theName) {

id = theID;

name = theName;

}

int Student::getID() const {

return id;

}

void Student::setID(int theID) {

id = theID;

}

string Student::getName() const {

return name;

}

void Student::setName(string theName) {

name = theName;

}

void Student::display(ostream& out) const {

// I.E "Student #12345 (Johnathan Lee)"

out << "Student #" << id << " (" << name << ") ";

}

void Student::parse(istream& in) {

string lastName;

in >> id >> name >> lastName;

name = name + " " + lastName;

}