/\*

Assignment: Provided for DEque ADT

Platform/IDE : Windows10/VS2015

Description: Basic Singly Linked List ADT enhanced with array-like s

 subscripting. Implemented with a Node class and SLinkedList class.

(C)Copyright 2015 cbell4 - For use by MSU CSC232 students. Do not post

on publicly accessible websites without permission.

\*/

#include <iostream>

#include <string>

#include <ctime> //for time()

#include <cmath> //for round()

// Global constants.

const int LISTSIZE = 10;

// For cleaner code.

using std::cout;

using std::endl;

using std::ostream;

using std::string;

// Forward declarations required for << overload

template <typename E>

class SLinkedList;

template <typename E>

ostream& operator<< (ostream& out, const SLinkedList<E>& sll);

/\* Template class for nodes of singly linked list.

\*/

template <typename E>

class Node {

public:

 /\* By the Rule of All or Nothing, since we don't need to do anything

 special in constructors, destructors, etc. we don't need any

 of the 6 standard methods.

 \*/

 /\* Practical Tips - Using Templates:

 1) VS13/15 works with template <typename E> before

 each of these two friend declarations. Xcode 6/7

 does not, giving an error about "declaration

 shadows template parameter".

 2) VS13/15 needs the <E> after << even though

 intellisense may complain about "function definition

 not found". Xcode works with or without.

 The code below works with \*both\* IDEs. \*/

 // Give SLinkedList and << access to private members.

 friend class SLinkedList<E>;

 friend ostream& operator<< <E>(ostream& out, const SLinkedList<E>& sll);

private:

 E elem; //datatype independent element

 Node<E>\* next; //next list item

};

/\*

Template class for standard singly linked list.

\*/

template <typename E>

class SLinkedList {

public:

 /\* Rule of All or None: must use All since need a custom destructor.

 The destructor is where we will delete new nodes that are

 created in push().

 \*/

 SLinkedList(); //default constructor

 SLinkedList(const SLinkedList<E>& sll); //copy constructor

 SLinkedList(SLinkedList&& rhs); //move constructor

 ~SLinkedList(); //destructor

 SLinkedList<E>& operator= (SLinkedList<E> rhs); //copy assignment

 SLinkedList<E>& operator= (SLinkedList<E>&& rhs); //move assignment

 // Basic Operations

 bool empty() const; //is list empty?

 int size() const; //get size of list

 E& front() const; //get front element

 void push(const E& e); //add element to front

 void pop(); //delete front element.

 // For copy assignment.

 void swap(SLinkedList<E> rhs);

 // Output methods/functions.

 void printDetails() const; //for debugging

 friend ostream& operator<< <E>(ostream& out, const SLinkedList<E>& sll);

private:

 Node<E>\* head; //head of list

 int sllSize; //size of list

};

// Default Constructor.

template <typename E>

SLinkedList<E>::SLinkedList() {

 head = nullptr;

 sllSize = 0;

}

// Copy constructor. head and sllSize set via initializer list.

/// Note: Manipulating linked lists can sometimes be confusing so

/// I have extra comments in this section.

template <typename E>

SLinkedList<E>::SLinkedList(const SLinkedList<E>& rhs) :

 head(nullptr), sllSize(0) {

 if (!rhs.empty()) {

 Node<E>\* node = new Node<E>;

 head = node;

 Node<E>\* litr = head; //iterators for each list

 Node<E>\* ritr = rhs.head; //

 while (ritr->next != nullptr) { //do all but last node

 litr->elem = ritr->elem; //copy element

 Node<E>\* node1 = new Node<E>; //create next node

 litr->next = node1; //'link in' node

 litr = node1; //increment list itrs

 ritr = ritr->next; //

 }

 litr->elem = ritr->elem; //copy last node element

 litr->next = nullptr; //set end of list

 sllSize = rhs.sllSize; //copy size

 } //else rhs was empty & lhs set in initializer list so done

}

// Move constructor.

template <typename E>

SLinkedList<E>::SLinkedList(SLinkedList&& rhs) : head(nullptr), sllSize(0) {

 // Transfer ownership to new object.

 head = rhs.head;

 sllSize = rhs.sllSize;

 // Reset rhs - it will be destroyed

 rhs.head = nullptr;

 rhs.sllSize = 0;

}

// Custom destructor to delete the nodes created by push().

template <typename E>

SLinkedList<E>::~SLinkedList() {

 while (!empty())

 pop();

}

/\* Copy assignment overload uses copy and swap. Note list is passed

in by value, not by reference, so copy constructor has already been called. \*/

template <typename E>

SLinkedList<E>& SLinkedList<E>::operator= (SLinkedList<E> rhs) {

 swap(rhs); //swaps lhs with rhs

 return \*this;

}

// Move assignment.

template <typename E>

SLinkedList<E>& SLinkedList<E>::operator= (SLinkedList<E>&& rhs) {

 if (this != rhs) { //avoid self-assignment

 // Delete class object in context.

 Node<E>\* old = head;

 if (!empty()) {

 head = old->next;

 delete old;

 }

 // Transfer ownership to the lhs object.

 head = rhs.head;

 sllSize = rhs.sllSize;

 // Reset rhs - it will be destroyed

 rhs.head = nullptr;

 rhs.sllSize = NULL;

 }

 return \*this;

}

// Is list empty?

template <typename E>

bool SLinkedList<E>::empty() const {

 return head == nullptr;

}

// Return size of list.

template <typename E>

int SLinkedList<E>::size() const {

 return sllSize;

}

// Get front element.

template <typename E>

E& SLinkedList<E>::front() const {

 return head->elem;

}

// Add to front of list.

template <typename E>

void SLinkedList<E>::push(const E& e) {

 Node<E>\* node = new Node<E>;

 node->elem = e;

 node->next = head;

 head = node;

 sllSize++;

}

// Remove front element and delete it.

template <typename E>

void SLinkedList<E>::pop() {

 Node<E>\* old = head;

 if (!empty()) {

 head = old->next;

 delete old;

 }

 sllSize = 0;

}

/\* Swap two lists. The lhs is the current object. Used

here as an implementation of the copy and swap idiom.

There's a good write-up about it on stackoverflow. \*/

template <typename E>

void SLinkedList<E>::swap(SLinkedList<E> rhs) {

 // head and sllSize are from lhs

 std::swap(head, rhs.head);

 std::swap(sllSize, rhs.sllSize);

}

// Print details - for debugging. In case linking in a node makes a circular list.

template <typename E>

void SLinkedList<E>::printDetails() const {

 cout << "size = " << sllSize << "\n";

 Node<E>\* itr = head;

 int i = 0;

 while (itr != nullptr) {

 cout << "index: " << i << " data: " << itr->elem

 << " node: " << itr << "\n";

 itr = itr->next;

 i++;

 if (i > LISTSIZE + 1) //if structure of list gets messed up

 break; //avoid infinite loop

 }

}

/\*

Non-class overloads.

\*/

// Output operator overload. Print as: {1.11, 2.22, 3.33}.

template <typename E>

ostream& operator<< (ostream& out, const SLinkedList<E>& sll) {

 cout << "{";

 Node<E>\* itr = sll.head;

 if (sll.head->next == nullptr) //empty list

 cout << "}";

 else //print elements

 while (itr->next != nullptr) { //print all but last

 cout << itr->elem << ", ";

 itr = itr->next;

 }

 cout << itr->elem << "}"; //print last and close

 return out;

}

/\*

Non-class functions.

\*/

/\*\* Populate a list with 10 random floats. \*/

template <typename E>

SLinkedList<E>& createList(SLinkedList<E>& sll) {

 E elem;

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

 // generate a random number between 0.00 and 9.99

 float num = (float) (rand() / (RAND\_MAX / 9.99));

 // Use rounding 'trick' to get only 2 decimals

 elem = (float) (std::round(num / 0.01) \* 0.01);

 sll.push(elem);

 }

 return sll;

}

/\*\*\*\*\* MAIN \*\*\*\*\*/

int main() {

 srand((int) time(0)); //seed rand num generator

 SLinkedList<float> sll;

 sll = createList(sll); //populate list with 10 random floats

 cout << "Initial list, size=" << sll.size() << "\n";

 cout << "Random list is " << sll << "\n";

 cout << "Print details of sll\n";

 sll.printDetails();

 cout << "\n\n";

 // Test SLL copy constructor.

 SLinkedList<float> sll1 = sll;

 SLinkedList<float> sll2 = sll; //extra list for testing moves

 cout << "Test copy constructor; sll1 should = sll\n";

 cout << "sll1 = " << sll1 << "\n\n";

 // Test SLL copy assignment (= overload).

 cout << "Test assignment overload; sll1 should = sorted sll\n";

 sll1 = sll;

 cout << "sll1 = " << sll1 << "\n\n";

 // Test move constructor.

 SLinkedList<float> sllm = createList(sll2);

 cout << "move ctor sllm = sll2 " << sllm << "\n\n";

 // Test move assignment.

 sllm = createList(sll);

 cout << "move assignement, now sllm = sll " << sllm << "\n\n";

 // Test subscript.

 // Add code here.

 cout << endl;

 getchar();

 return 0;

}