// FILE: Sequence.cpp

// CLASS IMPLEMENTED: sequence (see sequence.h for documentation)

// INVARIANT for the sequence ADT:

// 1. The number of items in the sequence is in the member variable

// used;

// 2. The actual items of the sequence are stored in a partially

// filled array. The array is a dynamic array, pointed to by

// the member variable data. For an empty sequence, we do not

// care what is stored in any of data; for a non-empty sequence

// the items in the sequence are stored in data[0] through

// data[used-1], and we don't care what's in the rest of data.

// 3. The size of the dynamic array is in the member variable

// capacity.

// 4. The index of the current item is in the member variable

// current\_index. If there is no valid current item, then

// current\_index will be set to the same number as used.

// NOTE: Setting current\_index to be the same as used to

// indicate "no current item exists" is a good choice

// for at least the following reasons:

// (a) For a non-empty sequence, used is non-zero and

// a current\_index equal to used indexes an element

// that is (just) outside the valid range. This

// gives us a simple and useful way to indicate

// whether the sequence has a current item or not:

// a current\_index in the valid range indicates

// that there's a current item, and a current\_index

// outside the valid range indicates otherwise.

// (b) The rule remains applicable for an empty sequence,

// where used is zero: there can't be any current

// item in an empty sequence, so we set current\_index

// to zero (= used), which is (sort of just) outside

// the valid range (no index is valid in this case).

// (c) It simplifies the logic for implementing the

// advance function: when the precondition is met

// (sequence has a current item), simply incrementing

// the current\_index takes care of fulfilling the

// postcondition for the function for both of the two

// possible scenarios (current item is and is not the

// last item in the sequence).

#include <cassert>

#include "Sequence.h"

#include <iostream>

using namespace std;

namespace CS3358\_Sp2016

{

// CONSTRUCTORS and DESTRUCTOR

sequence::sequence(size\_type initial\_capacity)

: used(0), current\_index(0), capacity(initial\_capacity)

{

assert(initial\_capacity != 0);

data = new value\_type[capacity];

}

sequence::sequence(const sequence& source)

: used(source.used), current\_index(source.current\_index),

capacity(source.capacity)

{

data = new value\_type[capacity];

for (size\_type i = 0; i < used; i++) {

data[i] = source.data[i];

}

}

sequence::~sequence()

{

delete [] data;

}

// MODIFICATION MEMBER FUNCTIONS

void sequence::resize(size\_type new\_capacity)

{

assert(new\_capacity > 0);

// Ensure new\_capacity will be able to retain existing data

if (new\_capacity < 1 || new\_capacity < used) {

// Set capacity to exactly what is needed or to 1

new\_capacity = (used > 0) ? used : 1;

}

// If the capacity is growing

capacity = (new\_capacity == capacity) ? (new\_capacity + 1) : new\_capacity;

value\_type \*temp = new value\_type[capacity]; // Allocate space

if (temp == NULL) // Ensure memory is allocated

{

cerr << "\*\*\* Error: Memory not available. Exiting program. \*\*\*";

exit(0);

}

for (size\_type i = 0; i < used; i++) {

temp[i] = data[i]; // Copy existing values

}

delete [] data; // Free up old space

data = temp; // Update sequence::data to new memory location

}

void sequence::start() {

// The first item on the sequence becomes the current item

// (if sequence is empty, current\_index is already 0

if (used > 0) {

current\_index = 0;

}

}

void sequence::advance()

{

assert(is\_item());

// If item(s) exists in sequence

current\_index++;

}

void sequence::insert(const value\_type& entry)

{

if (used == capacity) { // If adding to sequence will exceed capacity

size\_type new\_capacity = (int)((capacity \* 1.25) + 1);

resize(new\_capacity);

}

if (current\_index == used && used == 0) { // If there is no current index

// and the sequence is empty

data[0] = entry; // insert entry into first position

used++;

return;

}

// If there is no current item and the sequence is not empty

if (current\_index == used && used > 0) {

current\_index = 0; // Set current\_index to beginning of sequence

}

// Copy data from end towards beginning to preserve data

for(size\_type i = used; i > current\_index; i--)

{

data[i] = data[i-1];

}

data[current\_index] = entry; // Insert the entry

used ++;

}

void sequence::attach(const value\_type& entry)

{

if (used == capacity) { // If adding to sequence will exceed capacity

size\_type new\_capacity = (int)((capacity \* 1.25) + 1);

resize(new\_capacity);

}

if (current\_index == used) {// If there is no current index

data[current\_index] = entry; // insert entry into last position

used++;

return;

}

for(size\_type i = used; i > current\_index; i--) {

data[i] = data[i-1];

}

data[current\_index + 1] = entry;

used ++;

current\_index++;

}

void sequence::remove\_current()

{

assert(is\_item());

// Shift the data after the current\_index one position "left"

for (size\_type i = current\_index; i < used; i++) {

data[i] = data[i+1];

}

used--;

}

sequence& sequence::operator=(const sequence& source)

{

if (this != &source) { // If sequence is not being set to itself

used = source.used;

current\_index = source.current\_index;

capacity = source.capacity;

data = new value\_type[capacity]; // Allocate space

if (data == NULL) // Ensure memory is allocated

{

cerr << "\*\*\* Error: Memory not available. Exiting program. \*\*\*";

exit(0);

}

for (size\_type i = 0; i < used; i++) {

data[i] = source.data[i];

}

}

return \*this;

}

// CONSTANT MEMBER FUNCTIONS

sequence::size\_type sequence::size() const

{

return used;

}

bool sequence::is\_item() const

{

// A current item exists only if current\_index is not the same as used

if (current\_index != used) {

return true;

}

return false;

}

sequence::value\_type sequence::current() const

{

assert(is\_item());

return data[current\_index];

}

}