// Austin Keith Faulkner: a\_f408

// September 25, 2019

//

// 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\_FA2019

{

// CONSTRUCTOR

sequence::sequence(size\_type initial\_capacity) : used(0),

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

{

if (initial\_capacity <= 0)

{

cerr << "initial\_capacity must be 1 or greater." << endl

<< "Setting capacity equal to 1." << endl;

capacity = 1;

}

data = new value\_type[capacity];

}

// COPY CONSTRUCTOR

sequence::sequence(const sequence& source) : used(source.used),

current\_index(source.current\_index), capacity(source.capacity)

{

data = new value\_type[capacity];

// Perform a deep copy of the sequence array.

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

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

}

// DESTRUCTOR

sequence::~sequence()

{

// Deleting dynamically allocated data.

delete [] data;

data = nullptr; // To prevent a stale pointer; I'm a purist.

}

// MODIFICATION MEMBER FUNCTIONS

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

{

if (new\_capacity < 1)

{

cerr << "new\_capacity must be 1 or greater." << endl

<< "Setting new\_capacity equal to 1." << endl;

new\_capacity = 1;

}

if (new\_capacity < used)

{

cerr << "new\_capacity must be size used or greater." << endl

<< "Setting new\_capacity equal to used." << endl;

new\_capacity = used;

}

capacity = new\_capacity;

value\_type\* newData = new value\_type[capacity];

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

newData[i] = data[i];

delete [] data;

data = newData;

}

void sequence::start() { current\_index = 0;}

void sequence::advance()

{

assert(is\_item());

current\_index = current\_index + 1;

}

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

{

if (capacity == used)

resize(size\_type(capacity \* 1.5) + 1);

if(!is\_item()) // If there is no current item, then

// the new entry has been inserted at the

// front of the sequence.

{

current\_index = 0;

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

{

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

}

data[current\_index] = entry;

++used;

}

else // Otherwise, a new copy of entry has been inserted in the

// sequence before the current item.

{

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

{

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

}

data[current\_index] = entry;

++used;

}

// In either case, the newly inserted item is now the current

// item of the sequence.

}

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

{

if(used == capacity)

resize(size\_type(capacity \* 1.5) + 1);

if(!is\_item()) // If there is no current item, then the new

// entry has been attached to the end of the

// sequence.

{

data[current\_index] = entry;

++used;

}

else // Otherwise, a new copy of entry has been inserted in the

// sequence after the current item.

{

current\_index = current\_index + 1;

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

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

data[current\_index] = entry;

++used;

}

// In either case, the newly inserted item is now the current

// item of the sequence.

}

void sequence::remove\_current()

{

assert(is\_item());

// The current item is removed from the sequence, and

// the item after this (if there is one) is now the new current

// item. If the current item is already the last item in the

// sequence, then there is no longer any current item.

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

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

--used;

}

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

{

if (this != &source) // Trapping self-assignment should there

// be such.

{

value\_type\* newData = new value\_type[source.capacity];

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

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

delete [] data;

data = newData;

used = source.used;

current\_index = source.current\_index;

capacity = source.capacity;

}

return \*this;

}

// CONSTANT MEMBER FUNCTIONS

sequence::size\_type sequence::size() const { return used; }

bool sequence::is\_item() const

{

// A true return value indicates that there is a valid

// "current" item that may be retrieved by activating the

// current member function (listed below). A false return value

// indicates that there is no valid current item.

return (current\_index != used);

}

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

{

assert(is\_item());

// The item returned is the current item in the sequence.

return data[current\_index];

}

}