```
// 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];
   }
}
```