```
// 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_FA2024
{
   // CONSTRUCTORS and DESTRUCTOR
   sequence::sequence(size_type initial_capacity): used(0),
capacity(initial_capacity)
   {
      assert(initial_capacity > 0);

      data = new value_type[initial_capacity];
```

```
    }

    sequence::sequence(const sequence& source): used(source.used),
current_index(source.current_index), capacity(source.capacity)
    {
       data = new value_type[capacity];

       for(int 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);

       if(new_capacity < used)
       {
          new_capacity = used;
       }

       value_type* newArr = new value_type[new_capacity];

       for(int i = 0; i < used; i++)
       {
          newArr[i] = data[i];
       }

       delete [] data;
       data = newArr;
       capacity = new_capacity;
    }

    void sequence::start()
    {
       current_index = 0;
    }

    void sequence::advance()
    {
       assert(is_item());

       if(is_item())
       {
          current_index++;
       }
    }
```

```
void sequence::insert(const value_type& entry)
{
   if(used == capacity)
   {
      resize(int(capacity*1.5));
   }

   if(!is_item())
   {
      current_index = 0;
   }

   for(size_type i = used; i > current_index; i--)
   {
      data[i] = data[i-1];
   }

   data[current_index] = entry;
   used++;
}

void sequence::attach(const value_type& entry)
{
   if(used == capacity)
   {
      resize(int(1.5*capacity));
   }

   if(!is_item())
   {
      current_index = used;
   }

   else
   {
      for(size_type i = used; i > current_index+1; i--)
      {
         data[i] = data[i-1];
      }
      current_index++;
   }

   data[current_index] = entry;
   used++;
}

void sequence::remove_current()
{
   assert(is_item());

   if (current_index == used - 1)
   {
      used--;
```

```
    }
    else
    {
        for (size_type i = current_index; i < used - 1; i++)
        {
            data[i] = data[i + 1];
        }
        used--;
    }

    if (current_index >= used)
    {
        current_index = used;
    }
}

sequence& sequence::operator=(const sequence& source)
{
    if (this == &source)
    {
        return *this;
    }

    if(used != 0)
    {
        delete [] data;
    }

    data = new value_type[source.capacity];
    used = source.used;
    current_index = source.current_index;
    capacity = source.capacity;


    for(int i = 0; i < source.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
{
    return current_index < used;
}

sequence::value_type sequence::current() const
```

```
    {
        assert(is_item());

        return data[current_index];
    }
}
```