```
// 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_SP2023
{
   // CONSTRUCTORS and DESTRUCTOR
   sequence::sequence(size_type initial_capacity)
```

```
        :used(0), capacity(initial_capacity),current_index(0)
    {
        if (initial_capacity <= 0)
        {
            capacity = 1;
        }

        data = new value_type[capacity];
    }

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

        //initialize all elem of data[]
        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)
    {
        //set new_capacity to the minimum if needed
        if (new_capacity < used)
        {
            new_capacity = used;
        }

        //prevent array of capacity 0
        if (new_capacity == 0)
        {
            new_capacity = DEFAULT_CAPACITY;
        }


        value_type* newData = new value_type[new_capacity];

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

        //deallocate data and replace it with newData
        delete[] data;
        data = newData;

        capacity = new_capacity;
    }

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

    void sequence::advance()
    {
        if (this->is_item())
        {
            current_index++;
        }
    }

    void sequence::insert(const value_type& entry)
    {
        if (this->is_item())
        {
            data[current_index + 1] = data[current_index];
            data[current_index] = entry;
            current_index--;
            used++;
        }
        else
        {
            //if no current_index entry is set to the front
            data[0] = entry;
            current_index = 0;
            used++;
        }
    }

    void sequence::attach(const value_type& entry)
    {
        if (this->is_item())
        {
            data[current_index + 1] = entry;
            current_index++;
            used++;
        }
        else
        {
            //if no current_index entry is set to the end
            data[current_index] = entry;
            used++;
        }
    }

    void sequence::remove_current()
    {
        if (this->is_item())
        {
            //if current is the last item
            if (current_index + 1 == used)
            {
                used--;
            }
            else
            {
                for (int i = current_index; i < used - 1; i++)
                {
                    data[i] = data[i + 1];
                }
                used--;
            }
        }
    }

    sequence& sequence::operator=(const sequence& source)
    {
        //self assignment check
        if (this != &source)
        {
            used = source.used;
            capacity = source.capacity;
            current_index = source.current_index;

            //Dealocate data array
            delete[] data;

            //Initialize new data array
            data = new value_type[capacity];

            //repopulate this->data[] with rhs.data[]
            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
    {
        if (current_index == used)
        {
            return false;
        }

        return true;
    }

    sequence::value_type sequence::current() const
    {
        if (this->is_item())
        {
            return data[current_index];
        }
    }
}


```