```cpp
// 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>
```

```
44 using namespace std;
45
46 namespace CS3358_FA2021
47 {
48    // CONSTRUCTORS and DESTRUCTOR
49    sequence::sequence(size_type initial_capacity) :
50         capacity(initial_capacity >= 1 ?
51         initial_capacity :
52         DEFAULT_CAPACITY),
53         used(0),
54         current_index(0)
55     {
56         data = new value_type[capacity];
57     }
58
59    sequence::sequence(const sequence& source) :
60        data(new value_type[source.capacity]),
61        current_index(source.current_index),
62        used(source.used),
63        capacity(source.capacity)
64    {
65
66        for (size_type i = 0; i < used; i++)
67            data[i] = source.data[i];
68    }
69
70    sequence::~sequence()
71    {
72        delete [] data;
73        data = nullptr;
74    }
75
76    // MODIFICATION MEMBER FUNCTIONS
77    void sequence::resize(size_type new_capacity)
78    {
79        if (new_capacity <= 0) capacity = DEFAULT_CAPACITY;
80        else if (new_capacity <= used) capacity = used;
81        else capacity = new_capacity;
82
83        value_type* newData = new value_type[capacity];
84
85        for (size_type i = 0; i < used; i++)
86            newData[i] = data[i];
```

```
 87
 88        delete [] data;
 89        data = newData;
 90    }
 91
 92    void sequence::start()
 93    {
 94       current_index = 0;
 95    }
 96
 97    void sequence::advance()
 98    {
 99        assert(is_item());
100        current_index += 1;
101    }
102
103    void sequence::insert(const value_type& entry)
104    {
105        if (used == capacity)
106            resize(int(1.5 * capacity) + 1);
107
108         if (!is_item())
109             current_index = 0;
110
111        for(size_type i = used + 1; i > current_index; i--)
112            data[i] = data[i-1];
113
114         used += 1;
115         data[current_index] = entry;
116    }
117
118    void sequence::attach(const value_type& entry)
119    {
120        if (used == capacity)
121            resize(int(1.5 * capacity) + 1);
122
123        if (is_item()) {
124            current_index += 1;
125            for (size_type i = used + 1; i > current_index; i--) {
126                data[i] = data[i-1];
127            }
128        }
129
```

```
130         used += 1;
131         data[current_index] = entry;
132     }
133
134     void sequence::remove_current()
135     {
136         assert(is_item());
137
138         for (size_type i = current_index; i < used - 1; i++) {
139             data[i] = data[i + 1];
140         }
141         used -= 1;
142     }
143
144     sequence& sequence::operator=(const sequence& source)
145     {
146         if (this == &source) return *this;
147
148         value_type *newData = new value_type[source.capacity];
149
150         for (size_type i = 0; i < source.used; i++)
151             newData[i] = source.data[i];
152
153         delete [] data;
154
155         data = newData;
156         used = source.used;
157         capacity = source.capacity;
158         current_index = source.current_index;
159
160         return *this;
161     }
162
163     // CONSTANT MEMBER FUNCTIONS
164     sequence::size_type sequence::size() const
165     {
166        return used;
167     }
168
169     bool sequence::is_item() const
170     {
171        return current_index != used;
172     }
```

```
173
174     sequence::value_type sequence::current() const
175     {
176          assert(is_item());
177          return data[current_index];
178     }
179 }
180
181
```