```cpp
// FILE: sequence.cpp
// CLASS IMPLEMENTED: sequence (see sequence.h for documentation).
// INVARIANT for the sequence class:
// INVARIANT for the sequence class:
//   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 compile-time array whose size
//      is fixed at CAPACITY; the  member variable data references
//      the array.
//   3. 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.
//   4. The index of the current item is in the member variable
//      current_index. If there is no valid current item, then
//      current item 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>


namespace CS3358_SP2023_A04_sequenceOfNum
{
   //member constants
   template <class value_type>
   const typename sequence<value_type>::size_type
```

```
        sequence<value_type>::CAPACITY;
50
51
52      //constructor
53      template <class value_type>
54      sequence<value_type>::sequence() : used(0), current_index(0){ }
55
56
57      //modification member functions
58      template <class value_type>
59      void sequence<value_type>::start() { current_index = 0; }
60
61      template <class value_type>
62      void sequence<value_type>::end()
63      { current_index = (used > 0) ? used - 1 : 0; }
64
65      template <class value_type>
66      void sequence<value_type>::advance()
67      {
68         assert( is_item() );
69         ++current_index;
70      }
71
72      template <class value_type>
73      void sequence<value_type>::move_back()
74      {
75         assert( is_item() );
76         if (current_index == 0)
77            current_index = used;
78         else
79            --current_index;
80      }
81
82      template <class value_type>
83      void sequence<value_type>::add(const value_type& entry)
84      {
85         assert( size() < CAPACITY );
86
87         size_type i;
88
89         if ( ! is_item() )
90         {
91            if (used > 0)
92               for (i = used; i >= 1; --i)
93                  data[i] = data[i - 1];
94            data[0] = entry;
95            current_index = 0;
96         }
97         else
```

```
 98        {
 99           ++current_index;
100           for (i = used; i > current_index; --i)
101              data[i] = data[i - 1];
102           data[current_index] = entry;
103        }
104        ++used;
105     }
106
107     template <class value_type>
108     void sequence<value_type>::remove_current()
109     {
110        assert( is_item() );
111
112        size_type i;
113
114        for (i = current_index + 1; i < used; ++i)
115           data[i - 1] = data[i];
116        --used;
117     }
118
119
120     //constant member functions
121     template <class value_type>
122     typename sequence<value_type>::size_type sequence<value_type>::size()
          const { return used; }
123
124     template <class value_type>
125     bool sequence<value_type>::is_item() const { return (current_index <
          used); }
126
127     template <class value_type>
128     typename sequence<value_type>::value_type sequence<value_type>::current
          () const
129     {
130        assert( is_item() );
131
132        return data[current_index];
133     }
134  }
135
136  namespace CS3358_SP2023_A04_sequenceOfChar
137  {
138     //member constants
139     template <class value_type>
140     const typename sequence<value_type>::size_type
          sequence<value_type>::CAPACITY;
141
142     //constructor
```

```
143     template <class value_type>
144     sequence<value_type>::sequence() : used(0), current_index(0) { }
145
146
147     //modification member functions
148     template <class value_type>
149     void sequence<value_type>::start() { current_index = 0; }
150
151     template <class value_type>
152     void sequence<value_type>::end() { current_index = (used > 0) ? used -
          1 : 0; }
153
154     template <class value_type>
155     void sequence<value_type>::advance()
156     {
157        assert( is_item() );
158        ++current_index;
159     }
160
161     template <class value_type>
162     void sequence<value_type>::move_back()
163     {
164        assert( is_item() );
165        if (current_index == 0)
166           current_index = used;
167        else
168           --current_index;
169     }
170
171     template <class value_type>
172     void sequence<value_type>::add(const value_type& entry)
173     {
174        assert( size() < CAPACITY );
175
176        size_type i;
177
178        if ( ! is_item() )
179        {
180           if (used > 0)
181              for (i = used; i >= 1; --i)
182                 data[i] = data[i - 1];
183           data[0] = entry;
184           current_index = 0;
185        }
186        else
187        {
188           ++current_index;
189           for (i = used; i > current_index; --i)
190              data[i] = data[i - 1];
```

```
191            data[current_index] = entry;
192         }
193         ++used;
194      }
195
196      template <class value_type>
197      void sequence<value_type>::remove_current()
198      {
199         assert( is_item() );
200
201         size_type i;
202
203         for (i = current_index + 1; i < used; ++i)
204            data[i - 1] = data[i];
205         --used;
206      }
207
208
209      //constant member functions
210      template <class value_type>
211      typename sequence<value_type>::size_type sequence<value_type>::size()
           const { return used; }
212
213      template <class value_type>
214      bool sequence<value_type>::is_item() const { return (current_index <
           used); }
215
216      template <class value_type>
217      typename sequence<value_type>::value_type sequence<value_type>::current
           () const
218      {
219         assert( is_item() );
220
221         return data[current_index];
222      }
223   }
224
225
```