```
/// sequence.h////
// FILE: sequence.h
////////////////////////////////////////////////////////////////////
// NOTE: Two separate versions of sequence (one for a sequence of real
//       numbers and another for a sequence characters are specified,
//       in two separate namespaces in this header file. For both
//       versions, the same documentation applies.
////////////////////////////////////////////////////////////////////
// CLASS PROVIDED: sequence (a container class for a list of items,
//                 where each list may have a designated item called
//                 the current item)
//
// TYPEDEFS and MEMBER functions for the sequence class:
//   typedef ____ value_type
//     sequence::value_type is the data type of the items in the sequence.
//     It may be any of the C++ built-in types (int, char, etc.), or a
//     class with a default constructor, an assignment operator, and a
//     copy constructor.
//   typedef ____ size_type
//     sequence::size_type is the data type of any variable that keeps
//     track of how many items are in a sequence.
//   static const size_type CAPACITY = _____
//     sequence::CAPACITY is the maximum number of items that a
//     sequence can hold.
//
// CONSTRUCTOR for the sequence class:
//   sequence()
//     Pre:  (none)
//     Post: The sequence has been initialized as an empty sequence.
//
// MODIFICATION MEMBER FUNCTIONS for the sequence class:
//   void start()
//     Pre:  (none)
//     Post: The first item on the sequence becomes the current item
//           (but if the sequence is empty, then there is no current item).
//   void end()
//     Pre:  (none)
//     Post: The last item on the sequence becomes the current item
//           (but if the sequence is empty, then there is no current item).
//   void advance()
//     Pre:  is_item() returns true.
//     Post: If the current item was the last item in the sequence, then
//           there is no longer any current item. Otherwise, the new current
//           item is the item immediately after the original current item.
//   void move_back()
//     Pre:  is_item() returns true.
//     Post: If the current item was the first item in the sequence, then
//           there is no longer any current item. Otherwise, the new current
//           item is the item immediately before the original current item.
//   void add(const value_type& entry)
//     Pre:  size() < CAPACITY.
//     Post: A new copy of entry has been inserted in the sequence after
//           the current item. If there was no current item, then the new
//           entry has been inserted as new first item of the sequence. In
//           either case, the newly added item is now the current item of
//           the sequence.
//   void remove_current()
//     Pre:  is_item() returns true.
//     Post: The current item has been removed from the sequence, and
//           the item after this (if there is one) is now the new current
//           item. If the current item was already the last item in the
//           sequence, then there is no longer any current item.
//
// CONSTANT MEMBER FUNCTIONS for the sequence class:
//   size_type size() const
//     Pre:  (none)
```

```
67  //     Post: The return value is the number of items in the sequence.
68  //   bool is_item() const
69  //     Pre:  (none)
70  //     Post: A true return value indicates that there is a valid
71  //           "current" item that may be retrieved by activating the current
72  //           member function (listed below). A false return value indicates
73  //           that there is no valid current item.
74  //   value_type current() const
75  //     Pre:  is_item() returns true.
76  //     Post: The item returned is the current item in the sequence.
77  // VALUE SEMANTICS for the sequence class:
78  //    Assignments and the copy constructor may be used with sequence
79  //    objects.
80
81  #ifndef SEQUENCE_H
82  #define SEQUENCE_H
83
84  #include <cstdlib>  // provides size_t
85
86  namespace CS3358_FA2021_A04
87  {
88      template <typename T>
89      class sequence
90      {
91      public:
92          // TYPEDEFS and MEMBER CONSTANTS
93          typedef T value_type;
94          typedef size_t size_type;
95          static const size_type CAPACITY = 10;
96          // CONSTRUCTOR
97          sequence();
98          // MODIFICATION MEMBER FUNCTIONS
99          void start();
100         void end();
101         void advance();
102         void move_back();
103         void add(const value_type& entry);
104         void remove_current();
105         // CONSTANT MEMBER FUNCTIONS
106         size_type size() const;
107         bool is_item() const;
108         value_type current() const;
109
110     private:
111         value_type data[CAPACITY];
112         size_type used;
113         size_type current_index;
114     };
115
116 }
117
118 #include "sequence.cpp"
119 #endif
120
121
122 //// sequence.cpp ////
123
124 // FILE: sequence.cpp
125 // CLASS IMPLEMENTED: sequence (see sequence.h for documentation).
126 // INVARIANT for the sequence class:
127 // INVARIANT for the sequence class:
128 //   1. The number of items in the sequence is in the member variable
129 //      used;
130 //   2. The actual items of the sequence are stored in a partially
131 //      filled array. The array is a compile-time array whose size
132 //      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>
#include "sequence.h"

namespace CS3358_FA2021_A04
{

    template <typename T>
    sequence<T>::sequence() : used(0), current_index(0) { }

    template <typename T>
    void sequence<T>::start() { current_index = 0; }

    template <typename T>
    void sequence<T>::end()
    { current_index = (used > 0) ? used - 1 : 0; }

    template <typename T>
    void sequence<T>::advance()
    {
        assert( is_item() );
        ++current_index;
    }

    template <typename T>
    void sequence<T>::move_back()
    {
        assert( is_item() );
        if (current_index == 0)
            current_index = used;
        else
            --current_index;
    }

    template <typename T>
```

```
199     void sequence<T>::add(const value_type& entry)
200     {
201         assert( size() < CAPACITY );
202 
203         size_type i;
204 
205         if ( ! is_item() )
206         {
207             if (used > 0)
208                 for (i = used; i >= 1; --i)
209                     data[i] = data[i - 1];
210             data[0] = entry;
211             current_index = 0;
212         }
213         else
214         {
215             ++current_index;
216             for (i = used; i > current_index; --i)
217                 data[i] = data[i - 1];
218             data[current_index] = entry;
219         }
220         ++used;
221     }
222 
223     template <typename T>
224     void sequence<T>::remove_current()
225     {
226         assert( is_item() );
227 
228         size_type i;
229 
230         for (i = current_index + 1; i < used; ++i)
231             data[i - 1] = data[i];
232         --used;
233     }
234 
235     template <typename T>
236     typename sequence<T>::size_type sequence<T>::size() const { return used; }
237 
238     template <typename T>
239     bool sequence< T >::is_item() const { return (current_index < used); }
240 
241     template <typename T>
242     typename sequence<T>::value_type sequence<T>::current() const
243     {
244         assert( is_item() );
245 
246         return data[current_index];
247     }
248 }
249 
250 
251 ////sequenceTest.cpp/////
252 
253 // FILE: sequence.cpp
254 // CLASS IMPLEMENTED: sequence (see sequence.h for documentation).
255 // INVARIANT for the sequence class:
256 // INVARIANT for the sequence class:
257 //   1. The number of items in the sequence is in the member variable
258 //      used;
259 //   2. The actual items of the sequence are stored in a partially
260 //      filled array. The array is a compile-time array whose size
261 //      is fixed at CAPACITY; the  member variable data references
262 //      the array.
263 //   3. For an empty sequence, we do not care what is stored in any
264 //      of data; for a non-empty sequence the items in the sequence
```

```
265 //      are stored in data[0] through data[used-1], and we don't care
266 //      what's in the rest of data.
267 //   4. The index of the current item is in the member variable
268 //      current_index. If there is no valid current item, then
269 //      current item will be set to the same number as used.
270 //      NOTE: Setting current_index to be the same as used to
271 //            indicate "no current item exists" is a good choice
272 //            for at least the following reasons:
273 //            (a) For a non-empty sequence, used is non-zero and
274 //                a current_index equal to used indexes an element
275 //                that is (just) outside the valid range. This
276 //                gives us a simple and useful way to indicate
277 //                whether the sequence has a current item or not:
278 //                a current_index in the valid range indicates
279 //                that there's a current item, and a current_index
280 //                outside the valid range indicates otherwise.
281 //            (b) The rule remains applicable for an empty sequence,
282 //                where used is zero: there can't be any current
283 //                item in an empty sequence, so we set current_index
284 //                to zero (= used), which is (sort of just) outside
285 //                the valid range (no index is valid in this case).
286 //            (c) It simplifies the logic for implementing the
287 //                advance function: when the precondition is met
288 //                (sequence has a current item), simply incrementing
289 //                the current_index takes care of fulfilling the
290 //                postcondition for the function for both of the two
291 //                possible scenarios (current item is and is not the
292 //                last item in the sequence).
293
294 #include <cassert>
295 #include "sequence.h"
296
297 namespace CS3358_FA2021_A04
298 {
299
300     template <typename T>
301     sequence<T>::sequence() : used(0), current_index(0) { }
302
303     template <typename T>
304     void sequence<T>::start() { current_index = 0; }
305
306     template <typename T>
307     void sequence<T>::end()
308     { current_index = (used > 0) ? used - 1 : 0; }
309
310     template <typename T>
311     void sequence<T>::advance()
312     {
313         assert( is_item() );
314         ++current_index;
315     }
316
317     template <typename T>
318     void sequence<T>::move_back()
319     {
320         assert( is_item() );
321         if (current_index == 0)
322             current_index = used;
323         else
324             --current_index;
325     }
326
327     template <typename T>
328     void sequence<T>::add(const value_type& entry)
329     {
330         assert( size() < CAPACITY );
```

```
331
332         size_type i;
333
334         if ( ! is_item() )
335         {
336             if (used > 0)
337                 for (i = used; i >= 1; --i)
338                     data[i] = data[i - 1];
339             data[0] = entry;
340             current_index = 0;
341         }
342         else
343         {
344             ++current_index;
345             for (i = used; i > current_index; --i)
346                 data[i] = data[i - 1];
347             data[current_index] = entry;
348         }
349         ++used;
350     }
351
352     template <typename T>
353     void sequence<T>::remove_current()
354     {
355         assert( is_item() );
356
357         size_type i;
358
359         for (i = current_index + 1; i < used; ++i)
360             data[i - 1] = data[i];
361         --used;
362     }
363
364     template <typename T>
365     typename sequence<T>::size_type sequence<T>::size() const { return used; }
366
367     template <typename T>
368     bool sequence< T >::is_item() const { return (current_index < used); }
369
370     template <typename T>
371     typename sequence<T>::value_type sequence<T>::current() const
372     {
373         assert( is_item() );
374
375         return data[current_index];
376     }
377 }
```