```
 1: #ifndef LIST_H_
 2: #define LIST_H_
 3:
 4: #include <stdexcept>
 5: #include <cstddef>
 6:
 7: /** Variant of Koffman and Wolfgang list class.*/
 8: namespace KW {
 9:   template<typename Item_Type>
10:   class list {
11:   private:
12:
13:     struct DNode {
14:       Item_Type data;       // A copy of the data
15:       DNode* next;          // A pointer to the next DNode
16:       DNode* prev;          // A pointer to the previous DNode
17:       DNode(const Item_Type& the_data,
18:             DNode* prev_val = NULL, DNode* next_val = NULL) :
19:         data(the_data), next(next_val), prev(prev_val) {}
20:     };
21:
22:     DNode* head;     // A reference to the head of the list
23:     DNode* tail;     // A reference to the end of the list
24:     int num_items;   // The size of the list
25:
26:
27:   public:
28:
29:     /** Construct an empty list. */
30:     list() {
31:       head = NULL;
32:       tail = NULL;
33:       num_items = 0;
34:     }
35:
36:     /** Return size of the list. */
37:     int size() const {
38:       return num_items;
39:     }
40:
41:     /** Return true if list is empty. */
42:     bool empty() const {
43:       return num_items == 0;
44:     }
45:
46:     /** Return constant reference to first element of the list. */
47:     const Item_Type& front() const {
48:       return head->data;
49:     }
50:
51:     /** Return reference to first element of the list. */
52:     Item_Type& front() {
53:       return head->data;
54:     }
55:
56:     /** Return constant reference to last element of the list. */
57:     const Item_Type& back() const {
58:       return tail->data;
59:     }
60:
61:     /** Return constant reference to first element of the list. */
62:     Item_Type& back() {
63:       return tail->data;
64:     }
65:
66:     /** Insert an object at the beginning of the list. */
67:     void push_front(const Item_Type& item) {
```

```
68:       head = new DNode(item, NULL, head); // Step 1
69:       if (head->next != NULL)
70:         head->next->prev = head;  // Step 2
71:       if (tail == NULL)    // List was empty.
72:         tail = head;
73:       num_items++;
74:     }
75:
76:     /** Insert an object at the end of the list. */
77:     void push_back(const Item_Type& item) {
78:       if (tail != NULL) {
79:         tail->next = new DNode(item, tail, NULL); // Step 1
80:         tail = tail->next; // Step 2
81:         num_items++;
82:       } else {         // List was empty.
83:         push_front(item);
84:       }
85:     }
86:
87:     /** Remove an object at the beginning of the list. */
88:     void pop_front() {
89:       if (head == NULL)
90:         throw std::invalid_argument("Attempt to call pop_front() on an empty list");
91:       DNode* removed_node = head;
92:       head = head->next;
93:       delete removed_node;
94:       if (head != NULL)
95:         head->prev = NULL;
96:       else
97:         tail = NULL;
98:       num_items--;
99:     }
100:
101:    /** Remove an object at the end of the list. */
102:    void pop_back() {
103:      if (tail == NULL)
104:        throw std::invalid_argument("Attempt to call pop_back() on an empty list");
105:      DNode* removed_node = tail;
106:      tail = tail->prev;
107:      delete removed_node;
108:      if (tail != NULL)
109:        tail->next = NULL;
110:      else
111:        head = NULL;
112:      num_items--;
113:    }
```

```
114:
115:     class iterator {
116:       friend class list<Item_Type>;     // Give the parent class access to this class
117:
118:     private:
119:       list<Item_Type>* parent;                    // A reference to the parent list
120:       typename list<Item_Type>::DNode* current;   // A pointer to the current DNode
121:       iterator(list<Item_Type>* my_parent, DNode* position) :    // constructor
122:         parent(my_parent), current(position) {}
123:
124:     public:
125:       iterator(const iterator& other) :            // copy constructor
126:         parent(other.parent), current(other.current) {}
127:
128:       Item_Type& operator*() const {
129:         if (current == NULL)
130:           throw std::invalid_argument("Attempt to dereference end()");
131:         return current->data;
132:       }
133:
134:       Item_Type* operator->() const {
135:         if (current == NULL)
136:           throw std::invalid_argument("Attempt to dereference end()");
137:         return &(current->data);
138:       }
139:
140:       /** This is the "prefix" increment operator. */
141:       iterator& operator++() {
142:         if (current == NULL)
143:           throw std::invalid_argument("Attempt to advance past end()");
144:         current = current->next;
145:         return *this;
146:       }
147:
148:       /** This is the "postfix" increment operator. */
149:       iterator operator++(int) {
150:         iterator return_value = *this;  // Make a copy of the current value.
151:         ++(*this);                      // Advance self forward (using prefix form)
152:         return return_value;            // Return old value.
153:       }
154:
155:       /** This is the "prefix" decrement operator. */
156:       iterator& operator--() {
157:         if (current == parent->head)
158:           throw std::invalid_argument("Attempt to move before begin()");
159:         if (current == NULL)   // Past last element.
160:           current = parent->tail;
161:         else
162:           current = current->prev;
163:         return *this;
164:       }
165:
166:       /** This is the "postfix" decrement operator. */
167:       iterator operator--(int) {
168:         iterator return_value = *this;       // Make a copy of the current value.
169:         --(*this);                           // Move self backward (using prefix form
170:         return return_value;                 // Return old value.
171:       }
172:
173:       bool operator==(const iterator& other) {     // Compare for equality.
174:         return current == other.current;
175:       }
176:
177:       bool operator!=(const iterator& other) {     // Not equal
178:         return !operator==(other);
179:       }
180:     }; // End iterator
```

```
181:     class const_iterator {
182:       friend class list<Item_Type>;   // Give the parent class access to this class
183:
184:     private:
185:       const list<Item_Type>* parent;              // A pointer to the parent list
186:       typename list<Item_Type>::DNode* current;   // A pointer to the current node
187:       const_iterator(const list<Item_Type>* my_parent, DNode* position) :  // constru
188:         parent(my_parent), current(position) {}
189:
190:     public:
191:       const_iterator(const const_iterator& other) :       // copy constructor
192:         parent(other.parent), current(other.current) {}
193:
194:       const Item_Type& operator*() const {
195:         if (current == NULL)
196:           throw std::invalid_argument("Attempt to dereference end()");
197:         return current->data;
198:       }
199:
200:       const Item_Type* operator->() const {
201:         if (current == NULL)
202:           throw std::invalid_argument("Attempt to dereference end()");
203:         return &(current->data);
204:       }
205:
206:       /** This is the "prefix" increment operator. */
207:       const_iterator& operator++() {
208:         if (current == NULL)
209:           throw std::invalid_argument("Attempt to advance past end()");
210:         current = current->next;
211:         return *this;
212:       }
213:
214:       /** This is the "postfix" increment operator. */
215:       const_iterator operator++(int) {
216:         const_iterator return_value = *this;   // Make a copy of the current value
217:         ++(*this);                             // Advance self forward (using prefix
218:         return return_value;                   // Return old value
219:       }
220:
221:       /** This is the "prefix" decrement operator. */
222:       const_iterator& operator--() {
223:         if (current == parent->head)
224:           throw std::invalid_argument("Attempt to move before begin()");
225:         if (current == NULL)
226:           current = parent->tail;
227:         else
228:           current = current->prev;
229:         return *this;
230:       }
231:
232:       /** This is the "postfix" decrement operator. */
233:       const_iterator operator--(int) {
234:         const_iterator return_value = *this;       // Make a copy of the current valu
235:         --(*this);                                 // Move self back
236:         return return_value;                       // Return old value
237:       }
238:
239:       bool operator==(const const_iterator& other) {      // Compare for equality
240:         return current == other.current;
241:       }
242:
243:       bool operator!=(const const_iterator& other) {      // Not equal
244:         return !operator==(other);
245:       }
246:     }; // End const_iterator
```

```
247:     friend class iterator;         // Give list access to internal values in iterator
248:     friend class const_iterator;   // Give list access to internal values in const_it
249:
250:     /** Return iterator to beginning of the list. */
251:     iterator begin() {
252:       return iterator(this, head);
253:     }
254:
255:     /** Return const_iterator to beginning of the list. */
256:     const_iterator begin() const {
257:       return const_iterator(this, head);
258:     }
259:
260:     /** Return iterator to position at end of the list. */
261:     iterator end() {
262:       return iterator(this, NULL);
263:     }
264:
265:     /** Return const_iterator to position at end of the list. */
266:     const_iterator end() const {
267:       return const_iterator(this, NULL);
268:     }
269:
270:
271:
272:     /** Insert an object before given position.
273:      *  @return An iterator that references the inserted item.
274:      */
275:     iterator insert(iterator pos, const Item_Type& item) {
276:       // Check for special cases
277:       if (pos.current == head) {
278:         push_front(item);
279:         return begin();
280:       } else if (pos.current == NULL) {  // Past the last node.
281:         push_back(item);
282:         return iterator(this, tail);
283:       }
284:
285:       // Create a new node linked before node referenced by pos.
286:       DNode* new_node = new DNode(item,
287:                                   pos.current->prev,
288:                                   pos.current); // Step 1
289:
290:       // Update links
291:       pos.current->prev->next = new_node;     // Step 2
292:       pos.current->prev = new_node;           // Step 3
293:       num_items++;
294:       return iterator(this, new_node);
295:     }
296:
297:     /** erase the item at the given iterator.
298:      *   @return iterator which follows the deleted item
299:      */
300:     iterator erase(iterator pos) {
301:       if (empty())
302:         throw std::invalid_argument("Attempt to call erase on an empty list");
303:       if (pos == end())
304:         throw std::invalid_argument("Attempt to call erase of end()");
305:       /* Create an interator that references the position
306:          following pos.*/
307:       iterator return_value = pos;
308:       ++return_value;
309:       // Check for special cases.
310:       if (pos.current == head) {
311:         pop_front();
312:         return return_value;
313:       } else if (pos.current == tail) {
```

```
314:         pop_back();
315:         return return_value;
316:       } else { // Remove a node in the interior of the list.
317:         // Unlink current node.
318:         num_items--;
319:         DNode* removed_node = pos.current;
320:         removed_node->prev->next = removed_node->next;
321:         removed_node->next->prev = removed_node->prev;
322:         delete removed_node;
323:         return return_value;
324:       }
325:     }
326:
327:     /** Construct a list from a sequence */
328:     template <typename iterator>
329:     list(iterator begin, iterator end) {
330:       head = NULL;
331:       tail = NULL;
332:       while (begin != end) {
333:         push_back(*begin++);
334:       }
335:     }
336:
337:     /** Construct a copy of a list. */
338:     list(const list<Item_Type>& other) {
339:       head = NULL;
340:       tail = NULL;
341:       num_items = 0;
342:       for (const_iterator itr = other.begin(); itr != other.end();
343:            ++itr) {
344:         push_back(*itr);
345:       }
346:     }
347:
348:     /** Destroy a list. */
349:     ~list() {
350:       while (head != NULL) {
351:         DNode* current = head;
352:         head = head->next;
353:         delete current;
354:       }
355:       tail = NULL;
356:       num_items = 0;
357:     }
358:
359:     /** Swap this list contents with another one */
360:     void swap(list<Item_Type>& other) {
361:       std::swap(head, other.head);
362:       std::swap(tail, other.tail);
363:       std::swap(num_items, other.num_items);
364:     }
365:
366:     /** Assign the contents of one list to another. */
367:     list<Item_Type>& operator=(const list<Item_Type>& other) {
368:       if (&other != this) {
369:         list<Item_Type> temp_copy(other);       // Make a copy of the other list.
370:         swap(temp_copy);                  // Swap contents of self with the copy.
371:         return *this;        // Return -- upon return the copy will be destroyed.
372:       }
373:     }
374:
375:   };  // end of list class
376:
377: } // end of KW namespace
378: #endif
```