```
#ifndef CSCI180_LIST_H
#define CSCI180_LIST_H

namespace csci180 {
  /** A full list implementation in the spirit of the std::list class.
   *  A similar high-level idea is implemented in Chapter 5.1 of our text.
   */
  template <typename Object>
  class list {

  protected:
    struct Node {                                  // a node in the list
      Object element;                              // element
      Node*  prev;                                 // prev pointer
      Node*  next;                                 // next pointer
      Node(const Object& e = Object(), Node* p = NULL, Node* n = NULL)
        : element(e), prev(p), next(n) { }         // constructor
    };

  private:
    Node sentinel;      // single sentinel will mark both ends of the list
    int sz;             // number of user's items in list (sentinels not included)

  public:
    /** Standard constructor creates an empty list. */
    list() : sz(0) {
      sentinel.prev = sentinel.next = &sentinel;
    }

    /** Returns the number of objects in the list.
     *  @return number of elements
     */
    int size() const {  return sz;  }

    /** Determines if the list is currently empty.
     *  @return true if empty, false otherwise.
     */
    bool empty() const { return sz == 0; }

    /** Returns a const reference to the front object in the list.
     * @return reference to front element
     */
    const Object& front() const { return sentinel.next->element; }

    /** Returns a live reference to the front object in the list.
     * @return reference to front element
     */
    Object& front() { return sentinel.next->element; }

    /** Returns a const reference to the last object in the list.
     * @return reference to last element
     */
    const Object& back() const { return sentinel.prev->element;  }

    /** Returns a live reference to the last object in the list.
     * @return reference to last element
     */
    Object& back() { return sentinel.prev->element; }
```

```
59:     /** Inserts an object at the front of the list.
60:      *  @param the new element
61:      */
62:     void push_front(const Object& elem) {
63:       Node* t = new Node(elem, &sentinel, sentinel.next);
64:       sentinel.next = t;       // header has new node after it
65:       t->next->prev = t;       // old front has new node before it
66:       sz++;
67:     }
68:
69:     /** Inserts an object at the back of the list.
70:      *  @param the new element
71:      */
72:     void push_back(const Object& elem) {
73:       Node* t = new Node(elem, sentinel.prev, &sentinel);
74:       sentinel.prev = t;       // trailer has new node before it
75:       t->prev->next = t;       // old back has new node after it
76:       sz++;
77:     }
78:
79:     /** Removes the front object from the list. */
80:     void pop_front()  {
81:       Node* old = sentinel.next;   // node to remove
82:       sentinel.next = old->next;   // bypass old in forward direction
83:       old->next->prev = &sentinel; // bypass old in reverse direction
84:       sz--;
85:       delete old;
86:     }
87:
88:     /** Removes the back object from the list. */
89:     void pop_back()  {
90:       Node* old = sentinel.prev;   // node to remove
91:       sentinel.prev = old->prev;   // bypass old in reverse direction
92:       old->prev->next = &sentinel; // bypass old in forward direction
93:       sz--;
94:       delete old;
95:     }
96:
97:     // --------------  Nested iterator class ------------------
98:     class iterator {
99:       friend class list<Object>;   // give list class access
100:
101:    private:
102:      typename list<Object>::Node* node;
103:      iterator(Node* n) : node(n) { }
104:
105:    public:
106:      /** Default constructor gives invalid iterator */
107:      iterator() : node(NULL) { }
108:
109:      /** Copy constructor */
110:      iterator(const iterator& other) : node(other.node) {}
111:
112:      /** Return live reference to element */
113:      Object& operator*() const {
114:        return node->element;
115:      }
116:
117:      /** Return live pointer to element */
118:      Object* operator->() const {
119:        return &(node->element);
120:      }
```

```
121:       /** This is the "prefix" increment operator */
122:       iterator& operator++() {
123:         node = node->next;   // mutate the iterator
124:         return *this;
125:       }
126:
127:       /** This is the "postfix" increment operator */
128:       iterator& operator++(int) {
129:         iterator initial = *this;   // Make copy of initial value
130:         ++(*this);                  // Advance (using pre-increment)
131:         return initial;             // Return old value
132:       }
133:
134:       /** This is the "prefix" decrement operator */
135:       iterator& operator--() {
136:         node = node->prev;   // mutate the iterator
137:         return *this;
138:       }
139:
140:       /** This is the "postfix" decrement operator */
141:       iterator& operator--(int) {
142:         iterator initial = *this;   // Make copy of initial value
143:         --(*this);                  // Move backward (using pre-decrement)
144:         return initial;             // Return old value
145:       }
146:
147:       bool operator==(const iterator& other) {
148:         return node == other.node;
149:       }
150:
151:       bool operator!=(const iterator& other) {
152:         return node != other.node;
153:       }
154:     };  // end iterator class
155:
156:     // --------------  Nested const_iterator class -------------------
157:     class const_iterator {
158:       friend class list<Object>;   // give list class access
159:
160:     private:
161:       const typename list<Object>::Node* node;
162:       const_iterator(const Node* n) : node(n) { }
163:
164:     public:
165:       /** Default constructor gives invalid iterator */
166:       const_iterator() : node(NULL) { }
167:
168:       /** Copy constructor */
169:       const_iterator(const const_iterator& other) : node(other.node) {}
170:
171:       /** Return const reference to element */
172:       const Object& operator*() const {
173:         return node->element;
174:       }
175:
176:       /** Return const pointer to element */
177:       const Object* operator->() const {
178:         return &(node->element);
179:       }
```

```
180:       /** This is the "prefix" increment operator */
181:       const_iterator& operator++() {
182:         node = node->next;   // mutate the iterator
183:         return *this;
184:       }
185:
186:       /** This is the "postfix" increment operator */
187:       const_iterator& operator++(int) {
188:         const_iterator initial = *this;   // Make copy of initial value
189:         ++(*this);                        // Advance (using pre-increment)
190:         return initial;                   // Return old value
191:       }
192:
193:       /** This is the "prefix" decrement operator */
194:       const_iterator& operator--() {
195:         node = node->prev;                // mutate the iterator
196:         return *this;
197:       }
198:
199:       /** This is the "postfix" decrement operator */
200:       const_iterator& operator--(int) {
201:         const_iterator initial = *this;   // Make copy of initial value
202:         --(*this);                        // Move backward (using pre-decrement)
203:         return initial;                   // Return old value
204:       }
205:
206:       bool operator==(const const_iterator& other) {
207:         return node == other.node;
208:       }
209:
210:       bool operator!=(const const_iterator& other) {
211:         return node != other.node;
212:       }
213:     };  // end const_iterator class
214:
215:     friend class iterator;         // Give iterator access to list internals
216:     friend class const_iterator;   // Give const_iterator access to list internals
217:
218:     iterator begin() {
219:       return iterator(sentinel.next);
220:     }
221:
222:     const_iterator begin() const {
223:       return const_iterator(sentinel.next);
224:     }
225:
226:     iterator end() {
227:       return iterator(&sentinel);        // sentinel serves as end position
228:     }
229:
230:     const_iterator end() const {
231:       return const_iterator(&sentinel);  // sentinel serves as end position
232:     }
```

```
233:     /** Insert an object immediately before the position indicated by
234:         the iterator.  Note that it must be an iterator (as opposed to
235:         a const_iterator).  It returns an iterator to the newly
236:         inserted item.
237:     */
238:     iterator insert(iterator p, const Object& element) {
239:       Node* after = p.node;
240:       Node* v = new Node(element, after->prev, after);
241:       after->prev->next = v;
242:       after->prev = v;
243:       sz++;
244:       return iterator(v);
245:     }
246:
247:     /** erase the item at the given iterator.  Returns an iterator to
248:         the position that follows the deleted item.
249:     */
250:     iterator erase(iterator p) {
251:       Node* old = p.node;
252:       Node* after = old->next;
253:       after->prev = old->prev;
254:       old->prev->next = after;
255:       delete old;
256:       sz--;
257:       return iterator(after);
258:     }
259:
260:   protected:
261:     void removeAll() {                  // remove list contents
262:       while (!empty()) pop_front();
263:     }
264:
265:     void copyFrom(const list& other) {     // copy from other
266:       // assumes that this list is properly empty
267:       for (const_iterator ci = other.begin(); ci != other.end(); ++ci)
268:         push_back(*ci);
269:     }
270:
271:   public:
272:     /** Copy constructor */
273:     list(const list& other) : sz(0) {
274:       sentinel.next = sentinel.prev = &sentinel;
275:       copyFrom(other);
276:     }
277:
278:     /** Destructor */
279:     ~list() {
280:       removeAll();     // get rid of contents between sentinels
281:     }
282:
283:     /** Assignment operator */
284:     list& operator=(const list& other) {
285:       if (this != &other) {                       // avoid self copy (x = x)
286:         removeAll();                              // remove old contents
287:         copyFrom(other);                          // copy new contents
288:       }
289:       return *this;
290:     }
291:   };  // end of list class
292: } // end of csci180 namespace
293: #endif
```