```
/*****************************************************************************
 File name:     list.h
 Author:        CS, Pacific University
 Date:          9/30/13
 Class:         CS300
 Assignment:    List Implementation
 Purpose:       This file defines the constants, data structures, and function
                prototypes for implementing a list data structure. In essence,
                the list API is defined for other modules.

*****************************************************************************/

#ifndef LIST_H_
#define LIST_H_

#include <stdbool.h>

#define TRUE    1
#define FALSE   0

// List error codes for each function to use

#define NO_ERROR                0

// list create failed
#define ERROR_NO_LIST_CREATE   -1

// user tried to operate on an empty list
#define ERROR_EMPTY_LIST       -2

// user tried to add data to a full list
#define ERROR_FULL_LIST        -3

// user tried to peekNext when no next existed
#define ERROR_NO_NEXT          -6

// user tried to peekPrev when no next existed
#define ERROR_NO_PREV          -7

// user tried to use current when current was not defined
#define ERROR_NO_CURRENT       -8

// user tried to operate on an invalid list. An invalid
// list may be a NULL ListPtr or contain an invalid value for numElements
```

```
44 #define ERROR_INVALID_LIST     -9
45
46 // user provided a NULL pointer to the function (other than the ListPtr)
47 #define ERROR_NULL_PTR         -10
48
49
50 #define NO_CURRENT -100
51 #define EMPTY_LIST 0
52 // Possible datatypes for later use
53
54 #define CHARACTER_VALUE 0
55 #define INTEGER_VALUE   1
56 #define FLOAT_VALUE     2
57
58 #define NO_CURRENT -100
59 #define EMPTY_LIST 0
60
61 // User-defined datatypes for easier reading
62
63
64 typedef short int ERRORCODE;
65
66 // The user of this data structure is only concerned with
67 // two data types: List and DATATYPE.  ListElement is an internal
68 // data structure not to be directly used by the user.
69 // If the List implementation changes (to static memory, a tree, etc)
70 // ListElement will change.
71
72 typedef struct DATATYPE
73 {
74   union
75   {
76     char charValue;
77     unsigned int intValue;
78     float floatValue;
79   };
80   unsigned short whichOne;
81 } DATATYPE;
82
83 typedef struct ListElement* ListElementPtr;
84 typedef struct ListElement
85 {
86   DATATYPE data;
87   ListElementPtr psNext;
88   ListElementPtr psPrev;
89 } ListElement;
90
91
92 // A list is a dynamic data structure where the current pointer and number
```

```
 93 // of elements are maintained at all times
 94
 95 typedef struct List* ListPtr;
 96 typedef struct List
 97 {
 98   ListElementPtr psHead;
 99   ListElementPtr psLast;
100   ListElementPtr psCurrent;
101   int numElements;
102 } List;
103
104 /*************************************************************************
    *
105 *                    Allocation and Deallocation
106 *************************************************************************
    */
107  ERRORCODE lstCreate (ListPtr );
108 // results: If list L can be created, then L exists and
109 // is empty returning NO_ERROR; otherwise,
110 // NO_LIST_CREATE is returned
111
112  ERRORCODE lstDispose (ListPtr );
113 // results: List no longer exists
114
115 /*************************************************************************
    *
116 *                  Checking number of elements in list
117 *************************************************************************
    */
118  ERRORCODE lstSize (ListPtr, int *);
119 // results: Returns the number of elements in the list
120
121  ERRORCODE lstIsFull (ListPtr, bool *);
122 // results: If list is full, return true;
123 // otherwise, return false
124
125  ERRORCODE lstIsEmpty (ListPtr, bool *);
126  // results: If list is empty, return true;
127  // otherwise, return false
128
129 /*************************************************************************
    *
130 *                          Peek Operations
131 *************************************************************************
    */
132  ERRORCODE lstPeek (ListPtr, DATATYPE *);
```

```
133 // requires: List is not empty
134 // results: The value of the current element is
135 // returned through the argument list
136 // IMPORTANT: Do not change current
137
138  ERRORCODE lstPeekPrev (ListPtr, DATATYPE *);
139  // requires: List contains two or more elements and
140  // current is not the first element
141  // results: The data value of current's predecessor is returned
142  // through the argument list.
143  // IMPORTANT: Do not change current
144
145  ERRORCODE lstPeekNext (ListPtr, DATATYPE *);
146 // requires: List contains two or more elements and
147 // current is not the last element
148 // results: The data value of current's successor is returned
149 // through the argument list.
150 // IMPORTANT: Do not change current
151
152 /
    ****************************************************************************
    *
153 *             Retrieving values and updating current
154 ****************************************************************************
    */
155
156  ERRORCODE lstFirst(ListPtr, DATATYPE *);
157 // requires: List is not empty
158 // results: The value of the first element is returned
159 // IMPORTANT: Current is changed to first if it exists
160
161  ERRORCODE lstLast(ListPtr, DATATYPE *);
162 // requires: List is not empty
163 // results: The value of the last element is returned
164 // IMPORTANT: Current is changed to last if it exists
165
166  ERRORCODE lstNext(ListPtr, DATATYPE *);
167 // requires: List is not empty, and current is not past the end of the list
168 // results: The value of the current element is returned
169 // IMPORTANT: Current is changed to the successor of the current element
170
171  ERRORCODE lstPrev(ListPtr, DATATYPE *);
172  // requires: List is not empty, and current is not past the first of the
    list
173  // results: The value of the current element is returned
174  // IMPORTANT: Current is changed to previous if it exists
175
176 /
    ****************************************************************************
```

```
   *
177 *                  Insertion, Deletion, and Updating
178 ************************************************************************
   */
179
180  ERRORCODE lstDeleteCurrent (ListPtr, DATATYPE *);
181 // requires: List is not empty
182 // results: The current element is deleted and its
183 // successor and predecessor become each
184 // others successor and predecessor. If the
185 // deleted element had a predecessor, then
186 // make it the new current element; otherwise,
187 // make the first element current if it exists.
188 // The deleted element is returned through the argument list.
189
190  ERRORCODE lstInsertAfter (ListPtr, DATATYPE);
191 // requires: List is not full
192 // results: if the list is not empty, insert the new
193 // element as the successor of the current
194 // element and make the inserted element the
195 // current element; otherwise, insert element
196 // and make it current.
197
198  ERRORCODE lstInsertBefore (ListPtr, DATATYPE);
199 // requires: List is not full
200 // results: If the list is not empty, insert the new
201 // element as the predecessor of the current
202 // element and make the inserted element the
203 // current element; otherwise, insert element
204 // and make it current.
205
206  ERRORCODE lstUpdateCurrent (ListPtr, DATATYPE);
207 // requires: List is not empty
208 // results: The value of ListElement is copied into the current element
209
210 /
   ************************************************************************
   *
211 *                        List Testing
212 ************************************************************************
   */
213
214  ERRORCODE lstHasNext (ListPtr, bool *);
215  // results: Returns true if there are more elements when traversing
216  // the list in a forward direction; otherwise, false is returned.
217
218  ERRORCODE lstHasPrev(ListPtr, bool *);
219  // results: Returns true if the current node has a
220  // predecessor; otherwise, false is returned
```

```
221
222 #endif /* LIST_H_ */
223
224
```