```
/*****************************************************************************
 File name:     list.h
 Author:        CS, Pacific University
 Date:          10.4.17
 Class:         CS300
 Assignment:    List Interface
 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>
#include <stdlib.h>
#include <string.h>

//*****************************************************************************
// Constants
//*****************************************************************************
#define MAX_ERROR_LIST_CHARS 64

enum {NO_ERROR = 0,
      ERROR_NO_LIST_CREATE,
      ERROR_NO_LIST_TERMINATE,
      ERROR_INVALID_LIST,
      ERROR_FULL_LIST,
      ERROR_NO_NEXT,
      ERROR_NO_CURRENT,
      ERROR_NULL_PTR,
      ERROR_EMPTY_LIST}; // If this error name changes, change stmt below
#define NUMBER_OF_LIST_ERRORS ERROR_EMPTY_LIST - NO_ERROR + 1


//*****************************************************************************
// Error Messages
//*****************************************************************************
#define LOAD_LIST_ERRORS strcpy (gszListErrors[NO_ERROR], "No Error.");\
strcpy (gszListErrors[ERROR_NO_LIST_CREATE], "Error: No List Create.");\
strcpy (gszListErrors[ERROR_NO_LIST_TERMINATE], "Error: No List Terminate.");\
strcpy (gszListErrors[ERROR_INVALID_LIST], "Error: Invalid List.");\
strcpy (gszListErrors[ERROR_FULL_LIST], "Error: Full List.");\
strcpy (gszListErrors[ERROR_NO_NEXT], "Error: No List Next Node.");\
strcpy (gszListErrors[ERROR_NO_CURRENT], "Error: No List Current Node.");\
strcpy (gszListErrors[ERROR_NULL_PTR], "Error: NULL Pointer.");\
strcpy (gszListErrors[ERROR_EMPTY_LIST], "Error: Empty List.");

//*****************************************************************************
```

```
50 // User-defined types
51 //*************************************************************************
52 typedef struct ListElement *ListElementPtr;
53 typedef struct ListElement
54 {
55   // Note: The memory allocated for data cannot contain any pointers to
56   //       additional data or terminate will not work correctly; therefore,
57   //       data must point to a contiguous block of data and contain no
58   //       additional pointers.
59   void *pData;
60   ListElementPtr psNext;
61 } ListElement;
62
63
64 // A list is a dynamic data structure where the current pointer and number
65 // of elements are maintained at all times
66
67 typedef struct List *ListPtr;
68 typedef struct List
69 {
70   ListElementPtr psFirst;
71   ListElementPtr psLast;
72   ListElementPtr psCurrent;
73   int numElements;
74 } List;
75
76 //*************************************************************************
77 // Allocation and Deallocation
78 //*************************************************************************
79 extern void lstCreate (ListPtr psList);
80 // results: If the list can be created, then the list exists and is empty;
81 //          otherwise, ERROR_NO_LIST_CREATE if psList is NULL
82
83 extern void lstTerminate (ListPtr psList);
84 // results: If the list can be terminated, then the list no longer exists
85 //          and is empty; otherwise, ERROR_NO_LIST_TERMINATE
86
87 extern void lstLoadErrorMessages ();
88 // results: Loads the error message strings for the error handler to use
89 //          No error conditions
90
91 //*************************************************************************
92 // Checking number of elements in list
93 //*************************************************************************
94 extern int lstSize (const ListPtr psList);
95 // results: Returns the number of elements in the list
96 //          error code priority: ERROR_INVALID_LIST
97
98 extern bool lstIsEmpty (const ListPtr psList);
```

```
 99 // results: If list is empty, return true; otherwise, return false
100 //          error code priority: ERROR_INVALID_LIST
101
102 //*************************************************************************
103 //                          List Testing
104 //*************************************************************************
105 extern bool lstHasCurrent (const ListPtr psList);
106 // results: Returns true if the current node is not NULL; otherwise, false
107 //          is returned
108 //          error code priority: ERROR_INVALID_LIST
109
110 extern bool lstHasNext (const ListPtr psList);
111  // results: Returns true if the current node has a successor; otherwise,
112  //          false is returned
113  //          error code priority: ERROR_INVALID_LIST
114
115 //*************************************************************************
116 //                            Peek Operations
117 //*************************************************************************
118 extern void *lstPeek (const ListPtr psList, void *pBuffer, int size);
119 // requires:  List is not empty
120 // results:   The value of the current element is returned
121 // IMPORTANT: Do not change current
122 //            error code priority: ERROR_INVALID_LIST, ERROR_NULL_PTR,
123 //                                 ERROR_EMPTY_LIST, ERROR_NO_CURRENT
124
125 extern void *lstPeekNext (const ListPtr psList, void *pBuffer, int size);
126 // requires:  List contains two or more elements and current is not last
127 // results:   The data value of current's successor is returned
128 // IMPORTANT: Do not change current
129 //            error code priority: ERROR_INVALID_LIST, ERROR_NULL_PTR,
130 //                                 ERROR_EMPTY_LIST, ERROR_NO_CURRENT,
131 //                                 ERROR_NO_NEXT
132
133 //*************************************************************************
134 //              Updating current
135 //*************************************************************************
136 extern void lstFirst (ListPtr psList);
137 // requires:  List is not empty
138 // results:   If the list is not empty, current is changed to the first element
139 //            error code priority: ERROR_INVALID_LIST,
140 //                                 ERROR_EMPTY_LIST
141
142 extern void lstNext (ListPtr psList);
143 // requires:  List is not empty
144 // results:   If the list is not empty, current is changed to the
145 //            successor of the current element
146 //            error code priority: ERROR_INVALID_LIST,
147 //                                 ERROR_EMPTY_LIST, ERROR_NO_CURRENT
```

```
148
149 extern void lstLast (ListPtr psList);
150 // requires:  List is not empty
151 // results:   If the list is not empty, current is changed to last
152 //            if it exists
153 //            error code priority: ERROR_INVALID_LIST,
154 //                                 ERROR_EMPTY_LIST
155
156 //*************************************************************************
157 //                  Insertion, Deletion, and Updating
158 //*************************************************************************
159
160 extern void lstInsertAfter (ListPtr psList, const void *pBuffer, int size);
161 // requires: List is not full
162 // results:  If the list is not empty, insert the new element as the
163 //           successor of the current element and make the inserted element
164 //           the current element; otherwise, insert element and make it
165 //           current.
166 //           error code priority: ERROR_INVALID_LIST, ERROR_NULL_PTR,
167 //                                ERROR_NO_CURRENT
168
169 extern void *lstDeleteCurrent (ListPtr psList, void *pBuffer, int size);
170 // requires: List is not empty
171 // results: The current element is deleted and its successor and
172 //          predecessor become each others successor and predecessor. If
173 //          the deleted element had a predecessor, then make it the new
174 //          current element; otherwise, make the first element current if
175 //          it exists. The value of the deleted element is returned.
176 //          error code priority: ERROR_INVALID_LIST, ERROR_NULL_PTR,
177 //                               ERROR_EMPTY_LIST, ERROR_NO_CURRENT
178
179 extern void lstInsertBefore (ListPtr psList, const void *pBuffer,
180                              int size);
181 // requires: List is not full
182 // results:  If the list is not empty, insert the new element as the
183 //           predecessor of the current element and make the inserted
184 //           element the current element; otherwise, insert element
185 //           and make it current.
186 //           error code priority: ERROR_INVALID_LIST, ERROR_NULL_PTR,
187 //                                ERROR_NO_CURRENT
188
189 extern void lstUpdateCurrent (ListPtr psList, const void *pBuffer,
190                               int size);
191 // requires: List is not empty
192 // results:  The value of pBuffer is copied into the current element
193 //            error code priority: ERROR_INVALID_LIST, ERROR_NULL_PTR,
194 //                                 ERROR_EMPTY_LIST, ERROR_NO_CURRENT
195 // IMPORTANT: user could update with smaller, larger, or the same size data
196 //            so free data, then reallocate based on size before updating
```

```
197
198 extern void lstReverse(ListPtr psList);
199
200 #endif /* LIST_H_ */
201
202
```