```
/*****************************************************************************
 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.");

//****************************************************************************
// User-defined types
//****************************************************************************
typedef struct ListElement *ListElementPtr;
typedef struct ListElement
{
  // Note: The memory allocated for data cannot contain any pointers to
  //       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 lstIsFull (const ListPtr psList);
99 // results: If list is full, return true; otherwise, return false
100 //          error code priority: ERROR_INVALID_LIST
101
102 extern bool lstIsEmpty (const ListPtr psList);
103 // results: If list is empty, return true; otherwise, return false
104 //          error code priority: ERROR_INVALID_LIST
105
106 //*************************************************************************
107 //                         List Testing
108 //*************************************************************************
109 extern bool lstHasCurrent (const ListPtr psList);
110 // results: Returns true if the current node is not NULL; otherwise, false
111 //          is returned
112 //          error code priority: ERROR_INVALID_LIST
```

```
113
114 extern bool lstHasNext (const ListPtr psList);
115  // results: Returns true if the current node has a successor; otherwise,
116  //          false is returned
117  //          error code priority: ERROR_INVALID_LIST
118
119 //*************************************************************************
120 //                          Peek Operations
121 //*************************************************************************
122 extern void *lstPeek (const ListPtr psList, void *pBuffer, int size);
123 // requires:  List is not empty
124 // results:   The value of the current element is returned
125 // IMPORTANT: Do not change current
126 //            error code priority: ERROR_INVALID_LIST, ERROR_NULL_PTR,
127 //                                 ERROR_EMPTY_LIST, ERROR_NO_CURRENT
128
129 extern void *lstPeekNext (const ListPtr psList, void *pBuffer, int size);
130 // requires:  List contains two or more elements and current is not last
131 // results:   The data value of current's successor is returned
132 // IMPORTANT: Do not change current
133 //            error code priority: ERROR_INVALID_LIST, ERROR_NULL_PTR,
134 //                                 ERROR_EMPTY_LIST, ERROR_NO_CURRENT,
135 //                                 ERROR_NO_NEXT
136
137 //*************************************************************************
138 //              Updating current
139 //*************************************************************************
140 extern void lstFirst (ListPtr psList);
141 // requires:  List is not empty
142 // results:   If the list is not empty, current is changed to the first element
143 //            error code priority: ERROR_INVALID_LIST,
144 //                                 ERROR_EMPTY_LIST
145
146 extern void lstNext (ListPtr psList);
147 // requires:  List is not empty
148 // results:   If the list is not empty, current is changed to the
149 //            successor of the current element
150 //            error code priority: ERROR_INVALID_LIST,
151 //                                 ERROR_EMPTY_LIST, ERROR_NO_CURRENT
152
153 extern void lstLast (ListPtr psList);
154 // requires:  List is not empty
155 // results:   If the list is not empty, current is changed to last
156 //            if it exists
157 //            error code priority: ERROR_INVALID_LIST,
158 //                                 ERROR_EMPTY_LIST
159
160 //*************************************************************************
161 //                  Insertion, Deletion, and Updating
162 //*************************************************************************
163
164 extern void lstInsertAfter (ListPtr psList, const void *pBuffer, int size);
165 // requires: List is not full
166 // results:  If the list is not empty, insert the new element as the
167 //           successor of the current element and make the inserted element
168 //           the current element; otherwise, insert element and make it
```

```
169 //           current.
170 //           error code priority: ERROR_INVALID_LIST, ERROR_NULL_PTR,
171 //                                ERROR_NO_CURRENT
172
173 extern void *lstDeleteCurrent (ListPtr psList, void *pBuffer, int size);
174 // requires: List is not empty
175 // results: The current element is deleted and its successor and
176 //          predecessor become each others successor and predecessor. If
177 //          the deleted element had a predecessor, then make it the new
178 //          current element; otherwise, make the first element current if
179 //          it exists. The value of the deleted element is returned.
180 //          error code priority: ERROR_INVALID_LIST, ERROR_NULL_PTR,
181 //                               ERROR_EMPTY_LIST, ERROR_NO_CURRENT
182
183 extern void lstInsertBefore (ListPtr psList, const void *pBuffer,
184                              int size);
185 // requires: List is not full
186 // results:  If the list is not empty, insert the new element as the
187 //           predecessor of the current element and make the inserted
188 //           element the current element; otherwise, insert element
189 //           and make it current.
190 //           error code priority: ERROR_INVALID_LIST, ERROR_NULL_PTR,
191 //                                ERROR_NO_CURRENT
192
193 extern void lstUpdateCurrent (ListPtr psList, const void *pBuffer,
194                               int size);
195 // requires: List is not empty
196 // results:  The value of pBuffer is copied into the current element
197 //            error code priority: ERROR_INVALID_LIST, ERROR_NULL_PTR,
198 //                                 ERROR_EMPTY_LIST, ERROR_NO_CURRENT
199 // IMPORTANT: user could update with smaller, larger, or the same size data
200 //            so free data, then reallocate based on size before updating
201
202
203 #endif /* LIST_H_ */
204
205
```