

## list.h

```
/******  
File name:      list.h  
Author:         Doug Ryan  
Edited:         Shereen Khoja  
Date:          9/30/12  
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_  
  
#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  
#define ERROR_INVALID_LIST  -9  
  
// user provided a NULL pointer to the function (other than the ListPtr)  
#define ERROR_NULL_PTR      -10  
  
// User-defined datatypes for easier reading
```

## list.h

```
typedef short int BOOLEAN;
typedef short int ERRORCODE;

// The user of this data structure is only concerned with
// two data types: List and DATATYPE. ListElement is an internal
// data structure not to be directly used by the user.
// If the List implementation changes (to dynamic memory, a tree, etc)
// ListElement will change.

#define CHARACTER_VALUE 0
#define INTEGER_VALUE 1
#define FLOAT_VALUE 2

typedef struct DATATYPE
{
    int intValue;
    int whichOne;
} DATATYPE;

typedef struct ListElement* ListElementPtr;
typedef struct ListElement
{
    DATATYPE data;
    ListElementPtr next;
    ListElementPtr prev;
} ListElement;

// A list is an array of ListElements where the current pointer and number
// of elements are maintained at all times

typedef struct List* ListPtr;

typedef struct List
{
    ListElementPtr head;
    ListElementPtr last;
    ListElementPtr current;
    int numElements;
} List;

/*****
*                               Allocation and Deallocation
*****/
ERRORCODE lstCreate (ListPtr );
// results: If list L can be created, then L exists and
// is empty returning NO_ERROR; otherwise,
// NO_LIST_CREATE is returned
```

## list.h

```
ERRORCODE lstDispose (ListPtr );
// results: List no longer exists

/*****
*
*           Checking number of elements in list
*****/
ERRORCODE lstSize (ListPtr, int *);
// results: Returns the number of elements in the list

ERRORCODE lstIsFull (ListPtr, BOOLEAN *);
// results: If list is full, return true;
// otherwise, return false

ERRORCODE lstIsEmpty (ListPtr, BOOLEAN *);
// results: If list is empty, return true;
// otherwise, return false

/*****
*
*           Peek Operations
*****/
ERRORCODE lstPeek (ListPtr, DATATYPE*);
// requires: List is not empty
// results: The value of the current element is
// returned through the argument list
// IMPORTANT: Do not change current

ERRORCODE lstPeekPrev (ListPtr, DATATYPE*);
// requires: List contains two or more elements and
// current is not the first element
// results: The data value of current's predecessor is returned
// through the argument list.
// IMPORTANT: Do not change current

ERRORCODE lstPeekNext (ListPtr, DATATYPE*);
// requires: List contains two or more elements and
// current is not the last element
// results: The data value of current's successor is returned
// through the argument list.
// IMPORTANT: Do not change current

/*****
*
*           Retrieving values and updating current
*****/

ERRORCODE lstFirst(ListPtr, DATATYPE*);
// requires: List is not empty
// results: The value of the first element is returned
```

## list.h

```
// IMPORTANT: Current is changed to first
// if it exists

ERRORCODE lstLast(ListPtr, DATATYPE*);
// requires: List is not empty
// results: The value of the last element is returned
// IMPORTANT: Current is changed to
// last if it exists

ERRORCODE lstNext(ListPtr, DATATYPE*);
// requires: List is not empty, and current is not past the end
// of the list
// results: The value of the current element is returned
// IMPORTANT: Current is changed to the successor
// of the current element

ERRORCODE lstPrev(ListPtr, DATATYPE*);
// requires: List is not empty, and current is not past the first
// of the list
// results: The value of the current element is returned
// IMPORTANT: Current is changed to previous
// if it exists

/*****
*                               Insertion, Deletion, and Updating
*****/

ERRORCODE lstDeleteCurrent (ListPtr, DATATYPE*);
// requires: List is not empty
// results: The current element is deleted and its
// successor and predecessor become each
// others successor and predecessor. If the
// deleted element had a predecessor, then
// make it the new current element; otherwise,
// make the first element current if it exists.
// The deleted element is returned through the argument
// list.

ERRORCODE lstInsertAfter (ListPtr, DATATYPE);
// requires: List is not full
// results: if the list is not empty, insert the new
// element as the successor of the current
// element and make the inserted element the
// current element; otherwise, insert element
// and make it current. The new element is inserted into
// the proper place and all other elements are shifted
// down the list.
```

## list.h

```
ERRORCODE lstInsertBefore (ListPtr, DATATYPE);
// requires: List is not full
// results: If the list is not empty, insert the new
// element as the predecessor of the current
// element and make the inserted element the
// current element; otherwise, insert element
// and make it current. The new element is inserted into
// the proper place and all other elements are shifted
// down the list.

ERRORCODE lstUpdateCurrent (ListPtr, DATATYPE);
// requires: List is not empty
// results: The value of ListElement is copied into the
// current element

/*****
*                               List Testing
*****/

ERRORCODE lstHasNext (ListPtr, BOOLEAN *);
// results: Returns true if there are more elements when traversing
// the list in a forward direction; otherwise, false is
// returned.

ERRORCODE lstHasPrev(ListPtr, BOOLEAN *);
// results: Returns true if the current node has a
// predecessor; otherwise, false is returned

#endif /* LIST_H_ */
```