```c
/* $Id: List.c,v 1.4 2011-10-08 19:09:41-07 - - $ */

/*
 * List.c
 * A doubly linked list ADT for integers.
 * PA2
 * By B Stewart Bracken
 * bbracken@ucsc.edu
 * ID# 1187817
*/

#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include "List.h"

/* Private inner Node struct, corresponding reference type, and
      * * constructor-destructor pair. Not exported.  */
typedef struct Node{
   int data;
   struct Node* next;
   struct Node* prev;
} Node;

typedef Node* NodeRef;

/*Node constructor*/
NodeRef newNode(int node_data) {
   NodeRef N = malloc( sizeof(Node) );
   assert ( N != NULL );
   N->data = node_data;
   N->next = NULL;
   return (N);
}

/*Node deconstructor*/
void freeNode(NodeRef* pN) {
   if (pN != NULL && *pN != NULL){
      free(*pN);
      *pN = NULL;
   }
}


/* Public List struct, constructor-destructor */
```

```c
typedef struct List{
   NodeRef front, back, current;
   int length, index;
} List;

/* List constructor */
ListRef newList(void){
   ListRef L;
   L = malloc(sizeof(List));
   assert ( L!= NULL );
   L->front = L->back = L->current = NULL;
   L->length = 0;
   L->index = -1;
   return(L);
}

/* List deconstructor */
void freeList(ListRef* pL) {
   if ( pL != NULL && *pL != NULL ){
      if (!isEmpty(*pL) ) {
         /*free all the things!*/
         makeEmpty(*pL);
      }
      free(*pL);
      *pL = NULL;
   }
}


/*Access functions ****************************************************
***********************************************************************/

/* getLength() - Returns length of list. */
int getLength(ListRef L) {
   if ( L == NULL ) killProgram("Calling getLength() on NULL ListRef.");
   return (L->length);
}

/* isEmpty() - Returns true if this List is empty, false otherwise. */
int isEmpty(ListRef L) {
   if ( L == NULL ) killProgram("Calling isEmpty() on NULL ListRef.");
   return ( L->length == 0 );
}

/* offEnd() - Returns true if current is undefined. */
```

```c
int offEnd(ListRef L) {
   if ( L == NULL ) killProgram("Calling offEnd() on NULL ListRef.");
   return (L->current == NULL);
}

/* getIndex() - Returns the current index position from 0 to
   length-1, or -1 if current is undefined. */
int getIndex(ListRef L) {
   if ( L == NULL ) killProgram("Calling getIndex() on NULL ListRef.");
   return (L->index);
}

/* getFront() - Returns front element.
   Pre: !isEmpty() */
int getFront(ListRef L) {
   if ( L == NULL ) killProgram("Calling getFront() on NULL ListRef.");
   if (isEmpty(L)) {
      killProgram("Method getFront() failed to pass pre !isEmpty() check.");
   } else {
      return (L->front->data);
   }
   return -1111;
}

/* getBack() - Returns back element.
   Pre: !isEmpty() */
int getBack(ListRef L) {
   if ( L == NULL ) killProgram("Calling getBack() on NULL ListRef.");
   if (isEmpty(L)) {
      killProgram("Method getBack() failed to pass pre !isEmpty() check.");
   } else {
      return (L->back->data);
   }
   return -1111;
}

/* getCurrent() - Returns current element.
   Pre: !isEmpty(), !offEnd() */
int getCurrent(ListRef L) {
   if ( L == NULL ) killProgram("Calling getCurrent() on NULL ListRef.");
    if (isEmpty(L) || offEnd(L)) {
       killProgram("Method getCurrent() failed to pass pre !isEmpty() && !offEnd()
check.");
    } else {
       return L->current->data;
```

```c
    }
   return -1111;
}

/* equals() - Returns true if this List and L are the same integer
   sequence. Ignores the current element in both Lists. */
int equals(ListRef L, ListRef M) {
   if ( L == NULL || M == NULL ) killProgram("Calling equals() on NULL ListRef.");
   NodeRef currL = L->front;
   NodeRef currM = M->front;
   if ( getLength(L) != getLength(M) ) return FALSE;
   while ( currL != NULL && currM != NULL ) {
      if ( currL->data != currM->data ) return FALSE;
      currL = currL->next;
      currM = currM->next;
   }
   return TRUE;
}

/* Manipulation procedures *******************************************
************************************************************************/

/* makeEmpty() - Sets this List to the empty state.
   Post: isEmpty(). */
void makeEmpty(ListRef L){
   if ( L == NULL ) killProgram("Calling makeEmpty() on NULL ListRef.");
   if (!isEmpty(L) ) {
      /*free all the things!*/
      while(!isEmpty(L)){
         deleteFront(L);
      }
      L->index = -1;
      L->front = L->back = L->current = NULL;
      L->length = 0;
   }
}

/* moveTo() - Moves current element marker to position i in
   this List. */
void moveTo(ListRef L, int i){
   if ( L == NULL ) killProgram("Calling moveTo() on NULL ListRef.");
   if ( i < 0 || i >= getLength(L) )
      killProgram("Bad index pass in moveTo() method.");
   else {
      NodeRef N;
```

```
        int k;
        int distFromCurrent = abs(L->index-i);
        if ( i == 0 ) {
           N = L->front;
        }else if ( i == getLength(L)-1 ) {
           N = L->back;
        } else if (i == L->index) {
           N = L->current;
        }else if ( (L->index > 0) && (distFromCurrent < i)
                    && (distFromCurrent < (getLength(L)-1-i)) ) {
           if ( L->index - i > 0 ) {
              for ( N = L->current, k = L->index; k > i; k--, N = N->prev);
           } else {
              for ( N = L->current, k = L->index; k < i; k++, N = N->next);
           }
        } else if ( (getLength(L)-1-i) <= i) {
           for ( N = L->back, k = getLength(L)-1; k>i; k--, N = N->prev);
        } else {
           for ( N = L->front, k = 0; k < i; k++, N = N->next);
        }
        L->current = N;
        L->index = i;
     }
   }

   /* movePrev() - Moves current one step toward front element.
      Pre: !isEmpty(), !offEnd(). */
   void movePrev(ListRef L) {
      if ( L == NULL ) killProgram("Calling movePrev() on NULL ListRef.");
       if (isEmpty(L) || offEnd(L)) {
           killProgram("Method movePrev() failed to pass pre !isEmpty() && !offEnd()
   check.");
       } else {
           L->current = L->current->prev;
           L->index--;
       }
   }

   /* moveNext() - Moves current one step toward back element.
      Pre: !isEmpty(), !offEnd(). */
   void moveNext(ListRef L) {
      if ( L == NULL ) killProgram("Calling moveNext() on NULL ListRef.");
       if (isEmpty(L) || offEnd(L)) {
           killProgram("Method moveNext() failed to pass pre !isEmpty() && !offEnd()
   check.");
```

```
       } else {
           L->current = L->current->next;
           L->index++;
       }
   }

   /* insertFront() - Inserts new element at the front position.
      Post: !isEmpty(). */
   void insertFront(ListRef L, int data) {
      if ( L == NULL ) killProgram("Calling insertFront() on NULL ListRef.");
       NodeRef N = newNode(data);
       if (L->front == NULL) {
           L->front = N;
           L->back = N;
       } else {
           L->front->prev = N;
           N->next = L->front;
       }
       L->front = N;
       L->length++;

       if (isEmpty(L)) {
           killProgram("Method insertFront() failed to pass post isEmpty() check.");
       }
   }

   /* insertBack() - Inserts new element in the back position.
      Post: !isEmpty(). */
   void insertBack(ListRef L, int data) {
      if ( L == NULL ) killProgram("Calling insertBack() on NULL ListRef.");
       NodeRef N = newNode(data);
       if (L->back == NULL) {
           L->front = N;
           L->back = N;
       } else {
           L->back->next = N;
           N->prev = L->back;
       }
       L->back = N;
       L->length++;

       if (isEmpty(L)) {
           killProgram("Method insertBack() failed to pass post isEmpty() check.");
       }
   }
```

```c
/* insertBeforeCurrent() - Inserts new element before current element.
    increments index by 1.
    Pre: !isEmpty(), !offEnd() */
void insertBeforeCurrent(ListRef L, int data) {
   if ( L == NULL ) killProgram("Calling insertBeforeCurrent() on NULL ListRef.");
    NodeRef N = newNode(data);
    if (isEmpty(L)) {
        killProgram("Method insertBeforeCurrent() failed to pass pre isEmpty()
check.");
    } else if (offEnd(L) ){
        killProgram("Method insertBeforeCurrent() failed to pass pre offEnd()
check.");
    } else{
        N->prev = L->current->prev;
        N->next = L->current;
        if (L->current->prev == NULL) {
            L->front = N;
        } else {
            L->current->prev->next = N;
        }
        L->current->prev = N;
        L->length++;
      L->index++;
    }
}

/* insertAfterCurrent() - Inserts new element after current element.
   Pre: !isEmpty(), !offEnd(). */
void insertAfterCurrent(ListRef L, int data) {
   if ( L == NULL ) killProgram("Calling insertAfterCurrent() on NULL ListRef.");
    NodeRef N = newNode(data);
    if (isEmpty(L) || offEnd(L)) {
        killProgram("Method insertBeforeCurrent() failed to pass ppre isEmpty() &
offEnd() check.");
    } else {
        N->prev = L->current;
        N->next = L->current->next;
        if (N->next == NULL) {
            L->back = N;
        } else {
            N->next->prev = N;
        }
        L->current->next = N;
        L->length++;
```

```c
    }
}

/* deleteFront() - Deletes front element.
   Pre: !isEmpty(). */
void deleteFront(ListRef L) {
   if ( L == NULL ) killProgram("Calling deleteFront() on NULL ListRef.");
   if (isEmpty(L)) {
       killProgram("Method deleteFront() cannot operate on an empty list.");
    } else {
      NodeRef temp = L->front;
        L->front = L->front->next;
        if (L->front == NULL)
            L->back = NULL;
        else
            L->front->prev = NULL;
      freeNode(&temp);
        L->length--;
    }
}

/* deleteBack() - Deletes back element.
   Pre: !isEmpty(). */
void deleteBack(ListRef L) {
   if ( L == NULL ) killProgram("Calling deleteBack() on NULL ListRef.");
   if ( isEmpty(L) )
      killProgram("Method deleteBack() cannot operate on an empty list.");
   else {
      NodeRef temp = L->back;
      L->back = L->back->prev;
      if ( L->back == NULL )
         L->front = NULL;
      else
         L->back->next = NULL;
      freeNode(&temp);
      L->length--;
   }
}

/* deleteCurrent() - Deletes current element.
   Pre: !isEmpty(), !offEnd()
   Post: offEnd() */
void deleteCurrent(ListRef L) {
   if ( L == NULL ) killProgram("Calling deleteCurrent() on NULL ListRef.");
    if ( isEmpty(L) || offEnd(L) ) {
         killProgram("Method deleteCurrent() cannot delete current node when list is
   empty or current is null.");
       }
      NodeRef temp = L->current;
       if ( L->current->prev == NULL ) {
           if ( L->current->next != NULL ) L->current->next->prev = NULL;
           L->front = L->current->next;
           if ( L->front == NULL ) L->back = NULL;
       } else if ( L->current->next == NULL ) {
           L->current->prev->next = NULL;
           L->back = L->current->prev;
       } else {
           L->current->prev->next = L->current->next;
           L->current->next->prev = L->current->prev;
       }
       L->current = NULL;
       L->index = -1;
       L->length--;
      freeNode(&temp);

       if (!offEnd(L)) {
           killProgram("Method deleteBack() failed to pass pre isEmpty() check.");
       }
   }


   /* Other functions *****************************************************
   **********************************************************************/

   /* copyList() - Returns a new list which is identical to this list. */
   ListRef copyList(ListRef L) {
      if ( L == NULL ) killProgram("Calling copyList() on NULL ListRef.");
      ListRef M = newList();
      NodeRef N;
      for ( N = L->front; N != NULL; N=N->next ){
         insertBack(M,N->data);
      }
      return M;
   }

   /* printList() prints current list to stdout */
   void printList(FILE* out, ListRef L) {
      if ( L == NULL ) killProgram("Calling printList() on NULL ListRef.");
      if ( isEmpty(L) ) printf("Nothing in List\n");
      else {
```

```
         NodeRef N = NULL;
         for ( N = L->front; N != NULL; N = N->next ){
            fprintf( out, "%d", N->data );
            if (N!=L->back) fprintf(out, " ");
         }
         /*fprintf(out, "\n");*/
      }
   }

   /* killProgram() - Utility method to report an error to user then exit. */
   void killProgram(char* error){
      printf("List.c: %s\n",error);
      exit(1);
   }
```