Prelab 7: Week of March 15th

For this prelab you are to use linked lists to implement a List ADT that supports the following functions:

```
/* This function returns an empty List object. */
List initList(int)

/* This function inserts the object of the first parameter
   at the head of the list and returns an error code. */
int insertAtHead(void*, List)

/* This function returns the object at the index location
   given by the first parameter. */
void* getAtIndex(int, List)

/* This function returns the size limit of the array, i.e.,
   the value that was specified when the List was created. */
int getSizeLimit(List)

/* This function returns the number of objects in the list. */
int countItems(List)

/* This function clears the list (makes it empty) just like
   after it was created by initList. */
void clearList(List)

/* This function frees all memory allocated for a list. */
void freeList(List)
```

If you don't understand things very well, then the above looks fine and you can't see anything problematic, i.e., you can't see any reason why there might be a problem implementing to these interface prototypes. If you understand things well then you're probably doubtful whether it's possible to implement to these interface prototypes.

Notice that the user will be declaring variables to be of type List, *not pointers to* List. And these List things seem to be passed by value, so how can the user's list be changed by the functions? Does C have super-secret black-magic stuff we haven't covered yet? Good questions!

As I did in the previous prelab, here are a couple of suggested structs:

```
typedef internalStruct {
     Node *head;
     int listLength;
} ListStuff;

typedef struct arrayListStruct {
      ListStuff *p;
} List;
```

Egad, you say, what's going on? Two structs? Why? Why not put the ListStuff members into List? What's gained by this weird pointer-to-a-struct-within-a-struct nonsense? Again, good questions!

The struct ListStuff contains the real information we need to maintain. The struct List is now a "wrapper" around the ListStuff pointer so that the user's List variables will never need to be changed by any function. Can you see why? Functions will only need to access and change members of the ListStuff struct. Note that when you free a list you'll need to free the ListStuff struct you created in initList.

Here are a few details to help you get started. The function initList will have a local variable of type List (no need to malloc it) and its member pointer (its only member) will be assigned to an allocated ListStuff struct. That local List variable is what will be returned to the user, and it will never need to be changed because that pointer to the ListStuff struct will always be the same. All of the changes we'll ever need to make will be to the ListStuff struct.

- JKU

========= OPTIONAL INFO IN CASE YOU'RE INTERESTED =========

You may recall that earlier in the semester I said that C has a problem with the way it treats pointers. Specifically, *data types* and *pointers-to-data-types* are different things, but look at how C treats them in the following declaration:

```
int i, *p, n, *q;
```

Within the same declaration we're declaring variables of completely different types: two are ints and two are *pointers* to int. The way things *should* work is the following:

```
int i, n;
int* p, q;  // we're declaring two int pointers
```

where now the distinction is clear. Not only would this be more conceptually consistent, it would also allow many things to be done more elegantly. For example, we could avoid the convoluted *pointer-to-a-struct-within-a-struct* wrapper simply by using a typedef:

```
typedef ListStuff* List;   // This won't do what we'd like it to do…
```

This would allow the user to declare List variables like the following:

```
List empList, inventoryList, shoppingList;
```

Note that this won't work in C because the above declaration would declare empList to be a pointer while the subsequent two variables are declared as structs. Do you see why? If so then you now see why the C treatment of pointers is problematic. Fortunately, once you understand the notion of a "wrapper" you can effectively circumvent the problem.

If you think about it a little deeper you'll realize that the use of interface functions achieves the same kind of thing: they provide a buffer/barrier between the user and the underlying mechanics of our implementations. It turns out that the concept of wrappers is important in many programming contexts.

- JKU

Bonus Challenge

Instead of the two structures I suggested earlier, I claim we could just use the following:

```
typedef internalStruct {
     Node *head;
     int *listLength;
} List;
```

How can this be made to work? (HINT: it will require use of a dummy node with our linked list.) Why is `listLength` now an `int` pointer? Assuming this works using only one struct, what advantage would there be to using two structs like I described earlier? (HINT: Consider the case in which the List struct has many more members.)