CMPS 101
Algorithms and Abstract Data Types
Summer 2008

Programming Assignment 2

Due Thursday July 10, 10:00 pm

In this assignment you will create a program with the same functionality as pa1, but now in C. Our purpose is again threefold: to make sure everyone is up to speed with C (especially pointers and structures), to practice modularity and ADTs, and to build an ADT implementation which will be used in future assignments.

Again you are to write a program that shuffles lists of integers. Your executable will be called `Shuffle` and will be invoked at the command line by typing: `Shuffle in_file out_file`. The program `FileIO.c` on the class webpage shows how file input and output can be accomplished in C. The program operation and file formats for this project will be identical to that described in pa1. As before your List ADT will be a double ended queue, with a current-position marker. Read the handout entitled "ADTs and Modules in Java and ANSI C", paying special attention to the section on ANSI C. Also read the handout entitled "Some Additional Remarks on ADTs and Modules in ANSI C". Your List module will export a `ListRef` type, along with the following operations.

```
/*** Constructors-Destructors ***/
ListRef newList(void);
void freeList(ListRef* pL);

/*** Access functions ***/
int isEmpty(ListRef L);
int offEnd(ListRef L);
int atFirst(ListRef L);
int atLast(ListRef L);
int getFirst(ListRef L);
int getLast(ListRef L);
int getCurrent(ListRef L);
int getLength(ListRef L);
int equals(ListRef A, ListRef B);

/*** Manipulation procedures ***/
void makeEmpty(ListRef L);
void moveFirst(ListRef L);
void moveLast(ListRef L);
void movePrev(ListRef L);
void moveNext(ListRef L);
void insertBeforeFirst(ListRef L, int data);
void insertAfterLast(ListRef L, int data);
void insertBeforeCurrent(ListRef L, int data);
void insertAfterCurrent(ListRef L, int data);
void deleteFirst(ListRef L);
void deleteLast(ListRef L);
void deleteCurrent(ListRef L);

/*** Other operations ***/
void printList(FILE* out, ListRef L);
```

```
ListRef copyList(ListRef L);
```

Function `newList` returns a `ListRef` which points to a new empty list. Function `freeList` frees all heap memory associated with its `ListRef*` argument, and sets *`pL` to `NULL`. Function `printList()` prints the List L to the file pointed to by out, formated as a space-separated string. This function plays roughly the same role as the "toString" function in Java. The operation of the other functions, and they're preconditions, are described in the pa1 specifications. Note that the `int` type in C will stand in for `boolean` in java, with 1 being true and 0 false. All of the above functions are required for full credit, but you may add additional operations if you like such as the following, whose operation is described in pa1.

```
ListRef catList(ListRef A, ListRef B);
```

Your program will be structured in three files: a client program `Shuffle.c`, a List implementation file `List.c`, and a List header file `List.h`. You must also turn in three other files: a `Makefile`, a driver program `ListTest.c` whose purpose is to test your List module in isolation, and a `README` file describing the files created for this assignment, their purposes, and relationships. Please note that the above file names are *not* optional. Your `Makefile` must create an executable called `Shuffle` and must include a clean utility that removes all executables and object files. Each file you turn in must begin with your name, user id, and assignment name. Thus for this project, you will turn in 6 files altogether.

A simple Makefile for this assignment might look like:

```
# Makefile for Programming Assignment 2

Shuffle : List.o Shuffle.o
     gcc -o Shuffle Shuffle.o List.o

Shuffle.o : List.h Shuffle.c
     gcc -c -ansi -Wall Shuffle.c

ListTest: List.o ListTest.o
     gcc -o ListTest ListTest.o List.o

ListTest.o : List.h ListTest.c
     gcc -c -ansi -Wall ListTest.c

List.o : List.h List.c
     gcc -c -ansi -Wall List.c

clean :
     rm -f Shuffle ListTest Shuffle.o ListTest.o List.o
```

The first line is a comment, as are all lines starting with "#". The rest of the file is organized into blocks of the form

```
Target : Dependencies
     Operation
```

separated by blank lines. Important note: the white space before the "operation" is a tab, not spaces! *Target* is a file to be created, and the *dependency list* for that target consists of those files on which the target depends. If one of the files in the dependency list changes, the target will be recompiled. *Operation* is the command which creates the target. The targets are listed in "top down" order, since

make occasionally gets confused if they are listed in another order. Once you have a Makefile, your entire program can be compiled (or re-compiled) simply by typing the unix command "gmake". This is efficient since only the changed modules will be re-compiled. The target "clean" is known as a phony target. Nothing is created, but an operation is performed. By typing "gmake clean" you remove all old targets. An excellent on-line manual for makefiles is mentioned on the webpage. You can also go to my CMPS 12B Spring 2005 webpage, follow the link to 12M lab assignments, then read lab assignment 1:

`http://www.soe.ucsc.edu/classes/cmps012b/Spring05/lab.html`

That lab assignment has a section on Makefiles. Note that the compile operations mentioned in the above Makefile call the gcc compiler. It is a requirement of this and all other assignments in C that your program compile without warnings or errors under gcc, and run properly in the IC Solaris computing environment provided by ITS (Information and Technology Services). In particular you should not use the cc compiler. Information on how to turn in your program is posted on the webpage.