1 Instructions

You must work **alone** on this project. Make sure to put your name in the **source code** files and **name** the **tarball** following the instructions of **Section 5**. **Section 4** describes **what** to submit and by **when**; **read it now**.

2 Lab Project Objectives

1. Use the **GCC C compiler** and a small subset of compiler options (-ansi, -c, -D -g, -o, -O0) to compile a C program.
2. Use the Linux **make** command and a **make file** to automate the building of a project.
3. Write a complete C program involving **multiple functions** in **multiple source code files**.
4. Learn how to write and include **header files**. Understand how to use **#ifndef ... #endif** to prevent multiple header file inclusion.
5. Properly use the **extern** and **static** reserved words. Understand what a **static function** is.
6. Define **struct** data types and use struct variables. Use **dot operator .** and **arrow operator ->** to access struct members.
7. Define and use **pointer variables**. Pass pointers as parameters to functions and return pointers from functions.
8. Use **malloc()** and **free()** to dynamically allocate and deallocate memory.
9. Implement a **doubly linked list** using pointers and dynamic memory allocation.
10. Open **text files** for reading and writing and use **fscanf()** and **fprintf()** to read from and write to text files.
11. Perform **formal testing** to document program correctness.

3 Doubly-Linked List

Download the **Project 3 Tarball** from the course website and extract it to a working directory (it will extract to a directory named **cse220-e02** with a subdirectory under **cse220-e02** named **slist**). The **slist** directory contains the following files,

Makefile A make file to build the project. To build the project, type **make** at the command line. The make file script will be executed by the **make** command, and if you have no syntax or linker errors, it will produce a binary named **ListTest**. Section 12.4 of Chapter 12 in the *GNU/Linux Lab Manual* discusses the Unix **make** command and **make files**. Read it now.

Test.sh A Bash shell script which tests the code by performing several test cases (in the **slist/testcases** subdirectory).

ListMan.c Contains definitions for a "list manager" data structure that is used during testing.

ListMan.h Contains declarations for the list manager.

SList.c Contains function definitions for the singly-linked list data structure.

SList.h Contains declarations for the singly-linked list data structure.

SListNode.c Contains function definitions for a singly-linked list node.

SListNode.h Contains declarations for a singly-linked list node.

ListTest.c Contains **main()** and a test driver to test the singly-linked list implementation.

This code can be written in different ways. I have attempted to make the code somewhat object-oriented by writing accessor/ mutator functions for the data members of the **SListNode** and **SList** structures. These functions are named: **SListNodeGet Data()**, **SListNodeGetNext()**, **SListNodeSetData()**, **SListNodeSetNext()**, **SListGetHead()**, **SListGetSize()**, **SListGetTail()**, **SListSetHead()**, **SListSetSize()**, and **SListSetTail()**.

Remember, in a **doubly-linked list** each node has a pointer to the **previous** and **next** nodes. We will maintain a pointer, named **mHead**, which points to the first node in the list, and a pointer named **mTail** which points to the last node. In addition to those two pointers, the **DList** struct will maintain an **int** variable named **mSize** which will store the number of nodes in the list; when **mSize** equals 0, the list is empty (an empty list also has **mHead** and **mTail** set to NULL). The data stored in each node will be a simple integer, but of course, it could be a datum of any data type, including a pointer.

After extracting the tarball, change directory to **slist** and type **make clean** to clean the source code file directory. Then type **make** to build the project; this will create a binary named **ListTest** which you can run by typing **./ListTest** at the command line. The binary expects two command line arguments: the first one is the name of a file containing test case input data, and the second is the name of the file to which the test case output is to be written. The test case input file is to contain various commands,

Linked List Testing Commands Table

append *list data* Appends a new node containing the data member *data* to list *list*. Returns *list*.

create *list* Creates a new, empty linked list named *list*. Returns the new list *list*.

free *list* Deletes all of the nodes in *list* and then deletes *list* itself.

find *list data* Finds and returns a pointer to the first node containing the data member *data* in *list*.
insert *list index data* Inserts a new node containing data member *data* into *list* at index *index*. Returns *list* on success, NULL on failure.
print *list* Prints the data stored in each node of *list* in order from head to tail.
remove *list data* Removes the first occurrence of a node containing data member *data* from *list*. Returns *list* on success, NULL on failure.

I suggest you study the **test1.in**, **test2.in**, ..., **test13.in** test case input files for examples. To perform one of these test cases, you would type the command: **./ListTest test1.in test1.out**. Then examine **test1.out** and see if it matches **test1.correct**. To compare two files for a byte-for-byte match, you can use the Unix **diff** command: **diff test1.out test1.correct > test1.diff**. Examine **test1.diff**. If it does not exist, then the **.out** and **.correct** files were identical. This means the test case **passed**. If the **.diff** file is nonempty, then you can examine the **.out** and **.correct** files to determine why the test case **failed**. To automate this process, the tarball contains a Bash shell script named **Test.sh**. This shell script can be executed by typing: **./Test.sh**. It will switch to the **testcases** subdirectory, run the **./ListTest** binary on each of the **.in** files to produce a **.out** file, then compare the **.out** file to the **.correct** file to determine the expected output matches the correct output. Study this Bash shell script. It would be helpful to read **Chapter 11** in the *GNU/Linux Lab Manual* which discusses how to write simple Bash shell scripts.

Your project is to use this code as a starting point to produce a doubly-linked list data structure. The code, as given to you, supports operations to **append** an integer to the list, **find** an integer in the list, **insert** an integer into the list (at a certain **index**) and **remove** an integer from the list. You shall extend this code to change the list to be a doubly linked list and to implement the following functions.

The **cse220-e02/dlist** subdirectory contains a template for the doubly-linked list implementation. The code must implement the following functions. The places in the code which you must complete are indicated by ??? symbols.

DList *DListAlloc ()
Allocates a new, empty doubly linked list. Returns a pointer to the new list. Returns NULL if malloc() fails.

DList *DListAppend (DList *pList, int pData)
Appends a new node to the list *pList* with the data member set to *pData*. Returns the list *pList*. Assertion error if *pList* is NULL.

DList *DListCopy (DList *pSrcList)
Creates and returns an exact copy of the list *pSrcList*. On input, *pSrcList* may or may not be the empty list. If *pSrcList* is the empty list, then this function shall return a new list which is also empty. Note: this is not the same thing as returning NULL. Assertion error if *pSrcList* is NULL.

void DListDebugPrint (FILE *pStream, DList *pList)
Prints the data members in each node of the linked list in order to the file output stream *pStream*. The format is **[** data_0 data_1 data_2 **...** data_{size-1} **]**. Note that there is a space following the left bracket, a space between each data member, and a space following the final data member and before the right bracket. Returns nothing.

void DListDebugPrintRev (FILE *pStream, DList *pList)
Prints the data members in each node of the linked list in *reverse* order to the file output stream *pStream*. The format is **[** data_{size-1} data_{size-2} **...** data_1 data_0 **]**. Note that there is a space following the left bracket, a space between each data member, and a space following the final data member and before the right bracket. Returns nothing.

DNode *DListFindData (Dlist *pList, int pData)
Returns a pointer to the node in the list *pList* which is the first occurrence of a node with data member set to *pData*. If such a node is not found, returns NULL. Note that indices are numbered starting at 0. Assertion error if *pList* is NULL.

DNode *DListFindIndex (DList *pList, int pIndex)
On success, returns a pointer to the node in *pList* at index *pIndex*. Fails if *pIndex* < 0 or *pIndex* ≥ *pList->mSize* and returns NULL. Assertion error if *pList* is NULL.

DList *DListFree (DList *pList)
Frees each of the nodes in the list *pList* and then frees *pList* itself. After returning, *pList* does not exist and should not be accessed. Returns NULL. Returns without doing anything if *pList* is NULL.

DListNode *DListGetHead (DList *pList)
Returns the head pointer of *pList*. Assertion errof if *pList* is NULL.

int DListGetIndex (DList *pList, int pData)
Searches *pList* for a node containing data member set to *pData*. Returns the index if the node if it is found, or -1 if *pData* is not in *pList*. Assertion error if *pList* is NULL.

int DListGetSize (DList *pList)
Accessor function for the *mSize* data member of *pList*. Assertion error if *pList* is NULL.

DListNode *DListGetTail (DList *pList)
Accessor function for the *mTail* data member of *pList*. Assertion error if *pList* is NULL.

DList *DListInsertBefore (DList *pList, int pBefore, int pData)
Inserts a new node containing data member set to *pData* into list *pList* before the first occurrence of the node with data member set to *pBefore*. On success, returns the list *pList*. Fails if a node containing data member *pBefore* is not found and returns NULL. Assertion error if *pList* is NULL.

DList *DListInsertIndex (DList *pList, int pIndex, int pData)
Inserts a new node containing *pData* into the list *pList* at index *pIndex*. The nodes in the list are numbered starting at 0. Fails and returns NULL if: (1) *pList* is empty; or (2) *pIndex* < 0; or (3) *pIndex* >= *pList->mSize*. Note: if you want to insert a new node at the end of the list then you should call the DListAppend() function. Assertion error if *pList* is NULL.

bool DListIsEmpty (DList *pList)
Returns true if *pList* is empty, false otherwise. Assertion error if *pList* is NULL.

DList *DListRemoveData (DList *pList, int pData)
Finds and removes the first occurrence of a node containing data member set to *pData* in list *pList*. On success, returns the list *pList*. Fails if a node containing data member set to *pData* is not found and returns NULL. Assertion error if *pList* is NULL.

DList *DListRemoveIndex (DList *pList, int pIndex)
Finds and removes the node at index *pIndex* in list *pList*. On success, returns the list *pList*. Fails if *pIndex* < 0 or *pIndex* ≥ *pList->mSize* and returns NULL. Assertion error if *pList* is NULL.

static DList *DListRemoveNode (DList *pList, DListNode *pNode)
Removes the node *pNode* from the list *pList*. Returns the list *pList* on success. Fails if *pNode* is NULL and returns NULL. Assertion error if *pList* is NULL.

DList *DListSetHead (DList *pList, DListNode *pHead)
Mutator function for the *mHead* data member of *pList*. Assertion error if *pList* is NULL.

DList *DListSetSize (DList *pList, int pSize)
Mutator function for the *mSie* data member of *pList*. Assertion error if *pList* is NULL.

DList *DListSetTail (DList *pList, DListNode *pTail)
Mutator function for the *mTail* data member of *pList*. Assertion error if *pList* is NULL.

If you create additional source code files, or remove any source code files, then modify **Makefile** appropriately. Your code should build cleanly with no syntax errors or warning messages when you type **make**. For testing, modify **ListTest.c** so it will support the additional testing commands,

Linked List Testing Commands Table - Additional Commands

copy *dst src*	Copies the list *src* to a new list *dst*. Returns a pointer to *dst*.
findat *list index*	Finds and returns a pointer to the node at index *index*. Returns NULL if *list* NULL, *index* < 0, or *index* ≥ *list->mSize*.
insert *list before data*	Inserts a new node containing data member *data* into *list* before the first node containing the data member *before*.
insertat *list index data*	Inserts a new node containing data member *data* into *list* at index *index*. Returns *list* on success, or NULL on failure.
printr *list*	Prints the data stored in each node of *list* in reverse order from tail to head.
removeat *list index*	Removes the node at index *index* from *list*. Returns *list* on success, or NULL on failure.

Example Testing Input File (test_dlist.in)

create L1	*L1 created*
append L1 100	*L1 = [100]*
append L1 200	*L1 = [100 200]*
append L1 300	*L1 = [100 200 300]*
append L1 400	*L1 = [100 200 300 400]*
append L1 500	*L1 = [100 200 300 400 500]*
print L1	*prints "L1 = [100 200 300 400 500]"*
copy L2 L1	*L1 is copied to L2*
print L2	*prints "L2 = [100 200 300 400 500]"*
free L1	*frees L1*
insert L2 300 250	*inserts 250 before 300 in L2; L2 = [100 200 250 300 400 500]*
insert L2 999 1	*fails to insert 1 before 999*
insert L2 100 50	*inserts 50 before 100 in L2; L2 = [50 100 200 250 300 400 500]*

insert L2 500 450	*inserts 450 before 500 in L2; L2 = [50 100 200 250 300 400 450 500]*
printr L2	*reverse prints "L2 = [500 450 400 300 250 200 100 50]"*
remove L2 250	*removes 250 from L2; L2 = [50 100 200 300 400 450 500]*
print L2	*prints "L2 = [50 100 200 300 400 450 500]"*
remove L2 50	*removes 50 from L2; L2 = [100 200 300 400 450 500]*
print L2	*prints "L2 = [100 200 300 400 450 500]"*
remove L2 500	*removes 500 from L2; L2 = [100 200 300 400 450]*
print L2	*prints "L2 = [100 200 300 400 450]"*
remove L2 999	*fails to remove 999 from L2*
removeat L2 3	*removes node at index 3 from L2; L2 = [100 200 300 450]*
print L2	*prints "L2 = [100 200 300 450]"*
removeat L2 0	*removes node at index 0 from L2; L2 = [200 300 450]*
print L2	*prints "L2 = [200 300 450]"*
removeat L2 2	*removes node at index 2 from L2; L2 = [200 300]*
print L2	*prints "L2 = [200 300]"*
removeat L2 -1	*fails to remove ndoe at index -1*
removeat L2 2	*fails to remove node at index 2*
insertat L2 0 100	*inserts new node with data 100 at 0 of L2; L2 = [100 200 300]*
print L2	*prints "L2 = [100 200 300]"*
insertat L2 1 150	*inserts new node with data 150 at 1 of L2; L2 = [100 150 200 300]*
print L2	*prints "L2 = [100 150 200 300]"*
create L1	*L1 created*
append L1 100	*appends 100 to L1; L1 = [100]*
append L1 200	*appends 200 to L1; L1 = [100 200]*
append L1 300	*appends 300 to L1; L1 = [100 200 300]*
append L1 400	*appends 400 to L1; L1 = [100 200 300 400]*
append L1 500	*appends 500 to L1; L1 = [100 200 300 400 500]*
print L1	*prints "L1 = [100 200 300 400 500]"*
free L2	*frees L2*
copy L2 L1	*copies L1 to L2; L2 = [100 200 300 400 500]*
print L2	*prints "L2 = [100 200 300 400 500]"*
free L1	*frees L1*
insert L2 300 250	*inserts 250 before 300 in L2; L2 = [100 200 250 300 400 500]*
insert L2 999 1	*fails to insert 1 before 999 in L2*
insert L2 100 50	*inserts 50 before 100 in L2; L2 = [50 100 200 250 300 400 500]*
insert L2 500 450	*inserts 450 before 500 in L2; L2 = [50 100 200 250 300 400 450 500]*
printr L2	*prints "L2 = [500 450 400 300 250 200 100 50]"*
remove L2 250	*removes 250 from L2; L2 = [50 100 200 300 400 450 500]*
print L2	*prints "L2 = [50 100 200 300 400 450 500]"*
remove L2 50	*removes 50 from L2; L2 = [100 200 300 400 450 500]*
print L2	*prints "L2 = [100 200 300 400 450 500]"*
remove L2 500	*removes 500 from L2; L2 = [100 200 300 400 450]*
print L2	*prints "L2 = [100 200 300 400 450]"*
remove L2 999	*fails to remove 999 from L2*
removeat L2 3	*removes node at index 3 from L2; L2 = [100 200 300 450]*
print L2	*prints "L2 = [100 200 300 450]"*
removeat L2 0	*removes node at index 0 from L2; L2 = [200 300 450]*
print L2	*prints "L2 = [200 300 450]"*
removeat L2 2	*removes node at index 2 from L2; L2 = [200 300]*
print L2	*prints "L2 = [200 300]"*
removeat L2 -1	*fails to remove node at index -1 from L2*
removeat L2 2	*fails to remove node at index 2 from L2*
insertat L2 0 100	*inserts new node with data 100 at 0 of L2; L2 = [100 200 300]*
print L2	*prints "L2 = [100 200 300]"*
insertat L2 1 150	*inserts new node with data 150 at 1 of L2; L2 = [100 150 200 300]*
print L2	*prints "L2 = [100 150 200 300]"*
insertat L2 3 250	*inserts new node with data 250 at 3 of L2; L2 = [100 150 200 250 300]*
insertat L2 -1 100	*fails to insert new node at index -1 of L2*
insertat L2 5 100	*fails to insert new node at index 5 of L2*
find L2 100	*finds 100 in L2*
find L2 300	*finds 300 in L2*
find L2 999	*fails to find 999 in L2*

create L3	*L3 created*
print L3	*prints L3 = [ ]*
printr L3	*prints L3 = [ ]*
append L3 100	*appends 100 to L3; L3 = [100]*
append L3 200	*appends 200 to L3; L3 = [100 200]*
append L3 100	*appends 100 to L3; L3 = [100 200 100]*
append L3 200	*appends 200 to L3; L3 = [100 200 100 200]*
print L3	*prints "L3 = [100 200 100 200]"*
find L3 100	*finds 100 in L3*
insert L3 100 50	*inserts 50 before100 in L3; L3 = [50 100 200 100 200]*
print L3	*prints [50 100 200 100 200]*
remove L3 100	*removes 100 from L3; L3 = [50 200 100 200]*
print L3	*prints "L3 = [50 200 100 200]"*
findat L3 0	*finds 50 at index 0 of L3*
findat L3 1	*finds 200 at index 1 of L3*
findat L3 3	*finds 200 at index 3 of L3*
findat L3 4	*fails to find node at index 4 of L3*
findat L3 -1	*fails to find node at index -1 of L3*

Example Testing Output File (test_dlist.out)

```
L1 created
appended 100 to L1
appended 200 to L1
appended 300 to L1
appended 400 to L1
appended 500 to L1
L1 = [ 100 200 300 400 500 ]
copied L1 to L2
L2 = [ 100 200 300 400 500 ]
freed L1
inserted 250 before 300 in L2
failed to insert 1 before 999 in L2
inserted 50 before 100 in L2
inserted 450 before 500 in L2
L2 = [ 500 450 400 300 250 200 100 50 ]
removed 250 from L2
L2 = [ 50 100 200 300 400 450 500 ]
removed 50 from L2
L2 = [ 100 200 300 400 450 500 ]
removed 500 from L2
L2 = [ 100 200 300 400 450 ]
failed to remove 999 from L2
removed node at 3 from L2
L2 = [ 100 200 300 450 ]
removed node at 0 from L2
L2 = [ 200 300 450 ]
removed node at 2 from L2
L2 = [ 200 300 ]
failed to remove node at -1
failed to remove node at 2
inserted 100 at 0 in L2
L2 = [ 100 200 300 ]
inserted 150 at 1 in L2
L2 = [ 100 150 200 300 ]
L1 created
appended 100 to L1
appended 200 to L1
appended 300 to L1
appended 400 to L1
appended 500 to L1
L1 = [ 100 200 300 400 500 ]
freed L2
```

```
copied L1 to L2
L2 = [ 100 200 300 400 500 ]
freed L1
inserted 250 before 300 in L2
failed to insert 1 before 999 in L2
inserted 50 before 100 in L2
inserted 450 before 500 in L2
L2 = [ 500 450 400 300 250 200 100 50 ]
removed 250 from L2
L2 = [ 50 100 200 300 400 450 500 ]
removed 50 from L2
L2 = [ 100 200 300 400 450 500 ]
removed 500 from L2
L2 = [ 100 200 300 400 450 ]
failed to remove 999 from L2
removed node at 3 from L2
L2 = [ 100 200 300 450 ]
removed node at 0 from L2
L2 = [ 200 300 450 ]
removed node at 2 from L2
L2 = [ 200 300 ]
failed to remove node at -1
failed to remove node at 2
inserted 100 at 0 in L2
L2 = [ 100 200 300 ]
inserted 150 at 1 in L2
L2 = [ 100 150 200 300 ]
inserted 250 at 3 in L2
failed to insert 100 at -1 in L2
failed to insert 100 at 5 in L2
found 100 in L2
found 300 in L2
failed to find 999 in L2
L3 created
L3 = [ ]
L3 = [ ]
appended 100 to L3
appended 200 to L3
appended 100 to L3
appended 200 to L3
L3 = [ 100 200 100 200 ]
found 100 in L3
inserted 50 before 100 in L3
L3 = [ 50 100 200 100 200 ]
removed 100 from L3
L3 = [ 50 200 100 200 ]
found 50 in L3 at 0
found 200 in L3 at 1
found 200 in L3 at 3
failed to findat 4 in L3
failed to findat -1 in L3
```

4 Testing

For grading purposes, we will run your binary against several test case input files, comparing the output to the correct output files. The test case input files are found in the **cse220-e03/dlist/testcases** directory. I have omitted the "correct" files; you must figure out what those will contain.

5 What to Submit for Grading and by When

When your programs are complete, type the following commands to "clean" the source code directories and create a bzipped tarball named **cse220-e02-*lastname*.tar.bz2**,

```
~ $ cd cse220-e02/dlist
```

```
~/cse220-p03/dlist $ make clean
~/cse220-p03/dlist $ cd ../..
~/cse220-p03 $ tar cvjf cse220-e02-lastname.tar.bz2 cse220-e02
```

Where *lastname* is your surname (or your first name if you do not have a surname). Be sure to put your name in the source code files (in the AUTHORS section of the header comment blocks) and name your tarball **cse220-e02-*last name*.tar.bz2**. Submit this tarball to Blackboard using the lab project submission link before the deadline. The **deadline is 11:59pm Fri 2 May 2014**. **Consult the online syllabus for the academic integrity policies. There is no additional extra credit for turning in this assignment early and no late assignments will be accepted.**