C Programming Simple Generic Data Structure

For this assignment, you will implement a simple purely generic *doubly-linked list* in C.

The best approach to implementing a generic list in C is discussed in CS 2505. Rather than repeat that discussion here, you will find links to the relevant notes from CS 2505 in the Suggested Resources section later in this specification. You should make sure you understand those notes before trying to implement your list.

Because your list implementation will be compiled with a test harness, there will be a mandatory interface, shown later in this specification. You will find a header file for the list on the course website, and repeated below. You must not make any changes to the specified interface of the list, although you may add helper functions as you see fit. Those helper functions should be declared as `static`.

Because the nodes in the generic list do not store user data, testing might seem to be impossible. When testing your solution, we will employ a wrapper type with integer payload, as described in the CS 2505 notes. In addition, it is fairly simple to write a function that will traverse a list and print detailed information about the nodes and how they are linked. The last page of this specification shows detailed information for a nonempty generic list.

Memory management requirements

The list itself does not perform any dynamic allocation or deallocation of memory. If it is not clear to you why the previous sentence is true, you do not yet understand the CS 2505 notes referenced above. Go over them again, and see a course TA or instructor.

Testing

Although your solution will be subjected to automated testing and grading, you are expected to test your solution thoroughly on your own Linux system or on the rlogin cluster. In order to make it easier for you to understand how the automated testing is performed, some test files will be provided:

`driver.c`	driver code to manage the testing process
`DList.h`	header file for the `DList` implementation (shown below)
`DListTestHarness.h`	header file for the test harness we will use
`DListTestHarness32.o`	32-bit Linux object file for the test harness we will use
`DListTestHarness64.o`	64-bit Linux object file for the test harness we will use
`IntegrityChecker.h`	header file for the list integrity checker we will use (see below)
`IntegrityChecker32.o`	32-bit Linux object file for the integrity checker
`IntegrityChecker64.o`	64-bit Linux object file for the integrity checker

You should download the supplied tar file and extract its contents into a directory. Object files are supplied for the test harness and integrity checker because the C source code contains elements of the solution for the deque itself. Two versions of each object file are supplied so you can build either a 32- or 64-bit executable, according to the capabilities of your Linux system. You should use the appropriate versions. Place your solution file, `DList.c`, into the same directory. The following command will build a 32-bit executable named `driver`:

```
gcc -o driver -m32 -std=c99 driver.c DList.c DListTestHarness32.o IntegrityChecker32.o
```

The following command will build a 64-bit executable named `driver`:

```
gcc -o driver -m64 -std=c99 driver.c DList.c DListTestHarness64.o IntegrityChecker64.o
```

You can execute the tests in two modes. If you execute the command: `./driver -prof`

the test code will generate a pseudo-random test sequence. This will yield a collection of test files, whose names indicate what was being tested. Examine those files to see if any errors were reported.

If you execute the command: `./driver`

(after running the driver with the `-prof` switch) the test code will repeat the last test sequence that was performed. This will also yield a collection of test files, whose names indicate what was being tested. Examine those files to see if any errors were reported.

This is essentially how the automated testing will be performed, except that the `-prof` switch will be used when the tests are run on a reference solution, and the two sets of test results will be compared automatically.

Since you will be given a limited number of submissions for automated testing and grading, you should be serious about your own testing.

Suggested resources

From the CS 2505 course website at:

http://courses.cs.vt.edu/~cs2505/summer2015/

you should consider the following notes:

Intro to Pointers	http://courses.cs.vt.edu/cs2505/summer2015/Notes/T14_IntroPointers.pdf
C Pointer Finale	http://courses.cs.vt.edu/cs2505/summer2015/Notes/T17_CPointerFinale.pdf
C struct Types	http://courses.cs.vt.edu/cs2505/summer2015/Notes/T24_CstructTypes.pdf
DT List	http://courses.cs.vt.edu/cs2505/summer2015/Notes/T25_CListDT.pdf
gdb List Example	http://courses.cs.vt.edu/cs2505/summer2015/Notes/T26_gdbListExample.pdf

Some of the other notes on the basics of C and separate compilation may also be useful.

Generic Doubly-linked List Interface:

DList.h (also posted on the course website):

```
#ifndef DLIST_H
#define DLIST_H
#include <stddef.h>
#include <stdbool.h>

/* DList objects have two Guard elements: fGuard just before the first
   interior element (if any) and rGuard just after the last interior
   element (if any).

   The 'prev' link of the front guard is NULL, as is the 'next' link of the
   rear guard.  Their other two links point toward each other (directly, or
   via the interior elements of the list).

   An empty list looks like this:

                    +------------+     +-----------+
               NULL-| fGuard     |<--->| rGuard    |-NULL
                    +------------+     +-----------+

   A list with two elements in it looks like this:

         +------------+     +-------+     +-------+     +-----------+
   NULL-| fGuard     |<--->|   1   |<--->|   2   |<--->| rGuard    |-NULL
         +------------+     +-------+     +-------+     +-----------+

   The symmetry of this arrangement eliminates lots of special cases in list
   processing.

   We say a DList object is proper if it conforms to the structural description
   above.  In particular, a proper DList object has the following properties:

     - fGuard.prev == NULL and rGuard.next == NULL
     - if the DList is empty, fGuard.next == &rGuard and
       rGuard.prev == &fGuard
     - if the DList is nonempty, fGuard.next points to the first interior
       node, and rGuard.prev points to the last interior node
     - for each interior node, its prev pointer points to the preceding node,
       and its next pointer points to the succeeding node

   (Because only one of the pointers in each guard element is used, we could
   in fact combine them into a single header element without sacrificing this
   simplicity.  But using two separate elements allows us to do a little bit
   of checking on some operations, which can be valuable.)

   This implementation of a list does not require use of dynamically allocated
   memory.  Instead, each structure that is a potential list element must embed
   a DNode member.  All of the list functions operate on these DNode objects.

   The DList_Entry macro allows conversion from a DNode back to a structure object
   that contains it.

   For example, suppose there is a need for a list of 'struct Widget'.
   'struct Widget' should contain a 'DNode' member, like so:

      struct Widget {
         DNode node;
         int bar;
         ...other members...
      };
```

```
   Then a list of 'struct Widget' can be be declared and initialized like so:

      DList Widget_L;

      DList_Init(&Widget_L);

   Retrieval is a typical situation where it is necessary to convert from a
   DNode back to its enclosing structure.  Here's an example using Widget_L:

      DNode *e = DList_Pop(Widget_L);

      struct Widget *f = DList_Entry(e, struct Widget, node);
      // now, do something with f...

   The interface for this list is inspired by the list<> template in the C++
   STL.  If you're familiar with list<>, you should find this easy to use.
   However, it should be emphasized that these lists do *no* type checking and
   can't do much other correctness checking.  If you screw up, it will bite you.

   Glossary of DList terms:

     - "interior element": An element that is not the head or tail, that is, a
               real list element.  An empty list does not have any interior elements.

     - "front": The first interior element in a list.  Undefined in an empty list.
                Returned by DList_Front().

     - "back": The last interior element in a list.  Undefined in an empty list.
               Returned by DList_Back().

     - "end":  The element figuratively just after the last interior element of a list;
               i.e., the rear guard.
               Well-defined even in an empty list.
*/

// DList node:
struct _DNode {

   struct _DNode *prev;     // Previous list element (toward fGuard)
   struct _DNode *next;     // Next list element (toward rGuard)
};

typedef struct _DNode DNode;

// DList object:
struct _DList {
   DNode fGuard;           // sentinel node at the front of the list
   DNode rGuard;           // sentinel node at the tail of the list
};

typedef struct _DList DList;

// DList_Entry() is a useful macro; there is a full discussion of a similar
// macro for a generic doubly-linked list implementation in the CS 2505 notes.
// Converts pointer to list element NODE into a pointer to the structure that
// NODE is embedded inside.  Supply the name of the outer structure STRUCT and
// the member name MEMBER of the NODE.  See the big comment at the top of the
// file for an example.

#define DList_Entry(NODE, STRUCT, MEMBER)                                   \
        ((STRUCT *) ((uint8_t *) (NODE) - offsetof (STRUCT, MEMBER)))
```

```
// Initialize DNode pointers to NULL.
//
// Pre:  pN points to a presumably raw DNode object
// Post: pN->prev and pN->next are NULL
//
void DNode_Init(DNode* const pN);

// Initialize DList to empty state.
//
// Pre:  pL points to a presumably raw DList object
// Post: *pL has been set to an empty state (see header comments)
//
void DList_Init(DList* const pL);

// Return whether DList is empty.
//
// Pre:  pL points to a proper DList object
// Returns true if *pL is empty, false otherwise
//
bool DList_Empty(const DList* const pL);

// Insert *pNode as predecessor of *pBefore.
//
// Pre:  pBefore points to an interior node or the rear guard of a
//         proper DList object
//       pNode points to a proper DNode object
// Post: *pNode has been inserted in front of *pBefore
//
void DList_PushBefore(DNode* const pBefore, DNode* const pNode);

// Insert *pNode as first interior element of *pL.
//
// Pre:  pL points to a proper DList object
//       pNode points to a proper DNode object
// Post: *pNode has been inserted at the front of *pL
//
void DList_PushFront(DList* const pL, DNode* const pNode);

// Insert *pNode as last interior element of *pL.
//
// Pre:  pL points to a proper DList object
//       pNode points to a proper DNode object
// Post: *pNode has been inserted at the rear of *pL
//
void DList_PushRear(DList* const pL, DNode* const pNode);

// Remove interior node preceding *pBefore and return it.
//
// Pre:  pBefore points to an interior node or rear guard of a
//         proper DList object
// Post: the interior DNode that preceded *pBefore has been removed
// Returns pointer to the DNode that was removed, NULL otherwise
//
DNode* DList_PopBefore(DNode* const pBefore);

// Remove first interior element of *pL and return it.
//
// Pre:  pL points to a proper DList object
// Post: the interior DNode that was at the front of *pL has been removed
// Returns pointer to the DNode that was removed, NULL if *pL was empty
//
DNode* DList_PopFront(DList* const pL);
```

```
// Remove last interior element of *pL and return it.
//
// Pre:  pL points to a proper DList object
// Post: the interior DNode that was at the rear of *pL has been removed
// Returns pointer to the DNode that was removed, NULL otherwise
//
DNode* DList_PopRear(DList* const pL);

// Return pointer to the first interior element, if any; does not remove
// the element.
//
// Pre:  pL points to a proper DList object
// Returns pointer first interior DNode in *pL, NULL if *pL is empty
//
const DNode* DList_Front(const DList* const pL);

// Return pointer to the last interior element, if any; useful for client-
// side traversal code.
//
// Pre:  pL points to a proper DList object
// Returns pointer last interior DNode in *pL, NULL if *pL is empty
//
const DNode* DList_Back(const DList* const pL);

#endif
```

Checking the Structure of a Generic List:

The output shown below was produced by a function that takes a pointer `pL` to a `DList` object and traverses the nodes, beginning with the front guard node and stopping at the rear guard node, printing address information (pointer values).

By examining the addresses, we can see whether the list is correctly structured.

```
&pL->fGuard:      28CCB4
&pL->rGuard:      28CCBC
 pL->fGuard.prev: 0
 pL->rGuard.next: 0
List appears to be nonempty.
Considering nodes 0 and 1
 & of node 0:     28CCB4
 & of node 1:     A08618
 node(0).next:    A08618
 node(1).prev:    28CCB4
 OK
Considering nodes 1 and 2
 & of node 1:     A08618
 & of node 2:     A317F0
 node(1).next:    A317F0
 node(2).prev:    A08618
 OK
Considering nodes 2 and 3
 & of node 2:     A317F0
 & of node 3:     A31800
 node(2).next:    A31800
 node(3).prev:    A317F0
 OK
Considering nodes 3 and 4
 & of node 3:     A31800
 & of node 4:     A31810
 node(3).next:    A31810
 node(4).prev:    A31800
 OK
Considering nodes 4 and 5
 & of node 4:     A31810
 & of node 5:     A31820
 node(4).next:    A31820
 node(5).prev:    A31810
 OK
Considering nodes 5 and 6
 & of node 5:     A31820
 & of node 6:     28CCBC
 node(5).next:    28CCBC
 node(6).prev:    A31820
```

Note how the addresses match up:

`node(0).next == address of node(1)`
`node(1).prev == address of node(0)`

That is exactly as it should be.

Implementing a function to produce the kind of output shown above is **not** a requirement for this assignment. **But**, doing so will improve your C programming skills, and provide you with a convenient way to examine the structure of your list.

What to Submit

You will submit your list implementation file, `DList.c`, and nothing else.

This assignment will be graded automatically, using the posted testing code (so you should not receive any unpleasant surprises). Your submission will be compiled with the testing code using the command: `gcc -std=c99 -m32 -Wall ...`

Test your solution thoroughly before submitting it.

The *Student Guide* and other pertinent information, such as the link to the proper submit page, can be found at:

http://www.cs.vt.edu/curator/

Pledge:

Each of your program submissions must be pledged to conform to the Honor Code requirements for this course. Specifically, you **must** include the following pledge statement in the submitted file:

```
//    On my honor:
//
//    - I have not discussed the C language code in my program with
//      anyone other than my instructor or the teaching assistants
//      assigned to this course.
//
//    - I have not used C language code obtained from another student,
//      or any other unauthorized source, either modified or unmodified.
//
//    - If any C language code or documentation used in my program
//      was obtained from another source, such as a text book or course
//      notes, that has been clearly noted with a proper citation in
//      the comments of my program.
//
//    <Student Name>
```