C Programming

For this assignment, you will implement a simple purely generic queue 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 on the Assignments page of the course website. You should make sure you understand those notes before trying to implement your queue.

Because your queue implementation will be compiled with a test harness, there will be a mandatory interface. You will find a header file for the queue on the course website, and repeated below. You must not make any changes to the specified interface of the queue.

Because the nodes in the generic queue do not store user data, testing might seem to be impossible. However, it is fairly simple to write a function that will traverse a queue 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 queue.

Memory management requirements

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

What to Submit

You will submit your queue implementation file: `Queue.c`, and nothing else. This assignment will be graded automatically. You will be allowed up to ten submissions for this assignment, so use them wisely. Test your functions thoroughly before submitting them. Make sure that your functions produce correct results for every logically valid test case you can think of.

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:

```c
// 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>
```
Generic Queue Interface:

Queue.h:

/* Our queues 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 queue).

An empty list looks like this:

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

A queue 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 queue
processing.

(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 queue does not require itself use dynamically
allocated memory. Instead, each structure that is a potential list
element must embed a QNode member. All of the list functions operate on
these QNodes.

The QList_Entry macro allows conversion from a QNode back to a structure
object that contains it.

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

struct Widget {
    QNode node;
    int bar;
    ...other members...
};

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

Queue Widget_Q;
Queue_Init(&Widget_Q);
Retrieval is a typical situation where it is necessary to convert from a QNode back to its enclosing structure. Here's an example using Widget_Q:

```c
QNode *e = Queue_Pop(Widget_Q);
struct Widget *f = Queue_Entry(e, struct Widget, node);
// now, do something with f...
```

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

Glossary of Queue terms:

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

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

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

- "end": The element figuratively just after the last interior element of a queue; i.e., the rear guard. Well-defined even in an empty queue. Returned by Queue_End().

```c
// Queue node:
struct _QNode {
    struct _QNode *prev;  /* Previous list element. */
    struct _QNode *next;  /* Next list element. */
};
typedef struct _QNode QNode;

// Queue object:
struct _Queue {
    QNode fGuard;  // sentinel node at the front of the queue
    QNode rGuard;  // sentinel node at the tail of the queue
};
typedef struct _Queue Queue;
```
// Queue_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 queue 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 Queue_Entry(NODE, STRUCT, MEMBER)                              
    ((STRUCT *) ((uint8_t *) (NODE) - offsetof (STRUCT, MEMBER)))

// Initialize QNode pointers to NULL.
//
// Pre: pN points to a QNode object
// Post: pN->prev and pN->next are NULL
// void QNode_Init(QNode* const pN);

// Initialize Queue to empty state.
//
// Pre: pQ points to a Queue object
// Post: *pQ has been set to an empty state (see preceding comment
// void Queue_Init(Queue* const pQ);

// Return whether Queue is empty.
//
// Pre: pQ points to a Queue object
// Returns true if *pQ is empty, false otherwise
// bool Queue_Empty(const Queue* const pQ);

// Insert *pNode as last interior element of Queue.
//
// Pre: pQ points to a Queue object
// pNode points to a QNode object
// Post: *pNode has been inserted at the rear of *pQ
// void Queue_Push(Queue* const pQ, QNode* const pNode);

// Remove first interior element of Queue and return it.
//
// Pre: pQ points to a Queue object
// Post: the interior QNode that was at the front of *pQ has been removed
// Returns pointer to the QNode that was removed, NULL if *pQ was empty
// QNode* const Queue_Pop(Queue* const pQ);

// Return pointer to the first interior element, if any.
//
// Pre: pQ points to a Queue object
// Returns pointer first interior QNode in *pQ, NULL if *pQ is empty
// QNode* const Queue_Front(const Queue* const pQ);
// Return pointer to the last interior element, if any.
//
// Pre:  pQ points to a Queue object
// Returns pointer last interior QNode in *pQ, NULL if *pQ is empty
// QNode* const Queue_Back(const Queue* const pQ);

// Return pointer to the rear guard; useful for traversal logic.
//
// Pre:  pQ points to a Queue object
// Returns pointer rear guard element.
// const QNode* const Queue_End(const Queue* const pQ);

#endif
Checking the Structure of a Generic Queue:

The output shown below was produced by a function that takes a pointer `pQ` to a `Queue` 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 queue is correctly structured.

```
&pQ->fGuard: 28CCB4
&pQ->rGuard: 28CCBC
pQ->fGuard.prev: 0
pQ->rGuard.next: 0
Queue 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 queue.