15-213/18-213/15-513, Fall 2017
C Programming Lab: Assessing Your C Programming Skills

Assigned:	Tues., Aug. 29, 2017
Due:	Thurs., Sept. 7, 11:59 pm
Last possible hand in:	Tues., Sept. 7, 11:59 pm

1 Logistics

There are no late days, grace days, or extensions for this assignment. Plan to start early enough to get it done by the due date. A well-prepared student can complete it in 1–2 hours, but it may require longer if you have not done much C programming. If you are not yet registered for the course, you can request an Autolab account and complete the assignment on schedule.

This is an individual project. All handins are electronic. You can do this assignment on any machine you choose. It has been tested on machines running several versions of Linux and OS X. The testing for your code will be done using Autolab, using OS and compiler configurations similar to those on the Andrew Linux machines. We advise you test your code on an Andrew Linux machine before submitting it.

2 Overview

This lab will give you practice in the style of programming you will need to be able to do proficiently, especially for the later assignments in the class. The material covered *should* all be review for you. Some of the skills tested are:

- Explicit memory management, as required in C.
- Creating and manipulating pointer-based data structures.
- Implementing robust code that operates correctly with invalid arguments, including NULL pointers.

The lab involves implementing a queue, supporting both last-in, first-out (LIFO) and first-in-first-out (FIFO) queueing disciplines. The underlying data structure is a singly-linked list, enhanced to make some of the operations more efficient.

3 Logging in to Autolab

All 213/513 labs are being offered this term through a Web service developed by CMU students and faculty called *Autolab*. Before you can download your lab materials, you will need to update your Autolab account. Point your browser at the Autolab front page

```
https://autolab.andrew.cmu.edu
```

You will be asked to authenticate via Shibboleth. After you authenticate the first time, Autolab will prompt you to update your account information with a *nickname*. Your nickname is the external name that identifies you on the public scoreboards that Autolab maintains for each assignment, so pick something interesting! You can change your nickname as often as you like. Once you have updated your account information, click on "Save Changes" button, and then select the "Home" link to proceed to the main Autolab page.

You must be enrolled to receive an Autolab account. If you added the class late, you might not be included in Autolab's list of valid students. In this case, you won't see the 213 course listed on your Autolab home page. If this happens, contact the staff and ask for an account.

If you are still on the waitlist for the course, then download a copy of the archive file (described below) from the course schedule web page. You can get working on the lab and then get an Autolab account once you are enrolled.

4 Downloading the assignment

Your lab materials are contained in a Linux archive file called `cprogramminglab-handout.tar`, which you can download from Autolab or from the class web page. Start by copying the file to a protected directory in Andrew in which you plan to do your work. Then login to a Linux machine and give the command

```
linux> tar xvf cprogramminglab-handout.tar
```

This will create a directory called `cprogramminglab-handout` that contains a number of files. Consult the file `README` for descriptions of the files.

5 Overview

The file `queue.h` contains declarations of the following structures:

```
/* Linked list element */
typedef struct ELE {
    int value;
    struct ELE *next;
} list_ele_t;
```

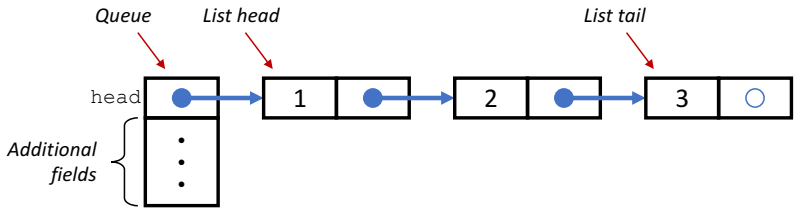

Figure 1: Linked-list implementation of a queue

```
/* Queue structure */
typedef struct {
    list_ele_t *head;  /* Linked list of elements */
} queue_t;
```

These are combined to implement a queue, as illustrated in Figure 1. The top-level representation of a queue is a structure of type `queue_t`. In the starter code, this structure contains only a single field "`head`," but you will want to add other fields. The queue contents are represented as a singly-linked list, with each element represented by a structure of type `list_ele_t`, having fields "`value`" and "`next`," storing a queue value and a pointer to the next list element, respectively. You may add other fields to this structure, although you need not do so.

In our C code, a queue is a pointer of type `queue_t *`. We distinguish two special cases: a *NULL* queue is one for which the pointer is set to `NULL`. An *empty* queue is one pointing to a valid `queue_t` structure with a `head` field set to `NULL`. Your code will need to deal properly with both of these cases, as well as queues containing one or more elements.

6 Programming Task

Your task is to modify the code in `queue.h` and `queue.c` to fully implement the following functions.

`q_new`: Create a new, empty queue.

`q_free`: Free all storage used by a queue.

`q_insert_head`: Attempt to insert a new element at the head of the queue (LIFO discipline).

`q_insert_tail`: Attempt to insert a new element at the tail of the queue (FIFO discipline).

`q_remove_head`: Attempt to remove the element at the head of the queue.

`q_size`: Compute the number of elements in the queue.

`q_reverse`: Reorder the list so that the queue elements are reversed in order.

More details can be found in the comments in these two files, including how to handle invalid operations (e.g., removing from an empty or NULL queue), and what side effects and return values the functions should have.

The following are some important notes about how you must implement these functions.

- Two of the functions: `q_insert_tail` and `q_size` will require some effort on your part to meet the required performance standards. The naive implementations would require $O(n)$ steps for a queue with n elements. We require that your implementations operate in time $O(1)$, i.e., that the time required is independent of the queue size. You can do this by including other fields in the `queue_t` data structure and managing these values properly as list elements are inserted, removed and reversed.
- You must implement `q_reverse` in a way that does not require allocating any additional memory. Instead, your code should modify the pointers in the existing list. Implementations that require allocating new list elements will fail the performance tests, due to the way our testing code monitors calls to `free` and `malloc`.
- Your program will be tested on queues with over 1,000,000 elements. You will find that you cannot traverse such long lists using recursive functions, since that would require too much stack space.

7 Testing

You can compile your code using the command:

```
linux> make
```

If there are no errors, the compiler will generate an executable program `qtest`, providing a command interface with which you can create, modify, and examine queues. Documentation on the available commands can be found by starting this program and running the `help` command:

```
linux> ./qtest
cmd>help
```

The following file (`traces/trace-eg.cmd`) illustrates an example command sequence:

```
# Demonstration of queue testing framework
# Initial queue is NULL.
show
# Create empty queue
new
# Fill it with some values.  First at the head
ih 2
ih 1
ih 3
# Now at the tail
it 5
```

```
it 1
# Reverse it
reverse
# See how long it is
size
# Delete queue.  Goes back to a NULL queue.
free
# Exit program
quit
```

You can see the effect of these commands by operating `qtest` in batch mode:

```
linux> ./qtest -f traces/trace-eg.cmd
```

With the starter code, you will see that many of these operations are not implemented properly.

The `traces` directory contains 14 trace files, with names of the form `trace-`k`-`*cat*`.txt`, where k is the trace number, and *cat* specifies the category of properties being tested. Each trace consists of a sequence of commands, similar to those shown above. They test different aspects of the correctness, robustness, and performance of your program. You can use these, your own trace files, and direct interactions with `qtest` to test and debug your program.

8 Evaluation

Your program will be evaluated using the fourteen traces described above. You will given credit (either 7 or 8 points, depending on the trace) for each one that executes correctly, summing to a maximum score of 100.

The driver program `driver.py` runs `qtest` on the traces and computes the score. This is the same program that will be used to compute your score with Autolab. You can invoke it directly with the command:

```
linux> ./driver.py
```

or with the command:

```
linux> make test
```

9 Handin

Using `make` to generate `qtest` also has the effect of generating a file `handin.tar`. You should do this on a Linux machine. You can upload this file (and only this file!) to Autolab, which will autograde your submission and record your scores. You may handin as often as you like until the due date.

IMPORTANT: Do not upload files in other archive formats, such as those with extensions `.zip`, `.gzip`, or `.tgz`.

10 Reflection

This should be a straightforward assignment for students who are fully prepared to take this course. If you found you had trouble writing this code or getting it to work properly, this may be an indication that you need to upgrade your C programming skills over the next month. Starting with Lab 4 in early October, you will need to write programs that require a mastery of the skills tested in this assignment.

A good place to start is to carefully study *The C Programming Language*, by Kernighan and Ritchie. This book documents the features of the language and also includes a number of examples illustrating good programming style. The book is a bit dated, and so it doesn't contain some more modern features of the language, such at the `bool` data type, but it is still considered one of the best books on how to program in C.