

ICS, Fall 2025

C Programming Lab: Assessing Your C Programming Skills

Assigned: Thursday, Sep 18
Due: Tuesday, Sep 30, 11:59 pm

1 Introduction

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 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.

- Explicit memory management, as required in C.
- Creating and manipulating pointer-based data structures.
- Working with strings.
- Enhancing the performance of key operations by storing redundant information in data structures.
- Implementing robust code that operates correctly with invalid arguments, including NULL pointers.

2 Logistics

There are no late days, grace days, or extensions for this assignment. Start early enough to get it done before the due date. Assume things will not go according to plan, and so you must allow extra time for heavily loaded systems, dropped Internet connections, corrupted files, traffic delays, minor health problems, etc.

A well-prepared student can complete the assignment 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. The testing for your code will be done using Autolab. Address of this lab: <http://101.132.146.234/courses/Introduction-to-Computer-Systems/assessments/lab0cprogramminglab>.

Before you can download your lab materials, you will need to update your Autolab account. 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.

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.

3 Downloading the assignment

For all assignments in this class, you should do all your programming and testing on the Linux machines.

To begin working on this assignment, start by creating a directory for this lab in a subdirectory of your home directory that is accessible only by you.

```
linux> mkdir ~/cprogramminglab  
linux> cd ~/cprogramminglab
```

Get the cprogramminglab-handout.tar from lab attachment on the Autolab website, and move it into the directory. Then, the directory will contain one tar file.

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

cprogramminglab-handout.tar contains the files you need. Extract them with these commands:

```
linux> tar --strip=1 -xf cprogramminglab-handout.tar  
linux> rm cprogramminglab-handout.tar  
linux> ls  
check-format driver.py helper.mk queue.c report.c  
console.c harness.c Makefile queue.h report.h  
console.h harness.h qtest.c README traces
```

For this lab, you will only *need* to modify queue.h and queue.c.

Don't modify any other files! If you do, the autograder will ignore those modifications and your code probably won't compile.

4 Overview

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

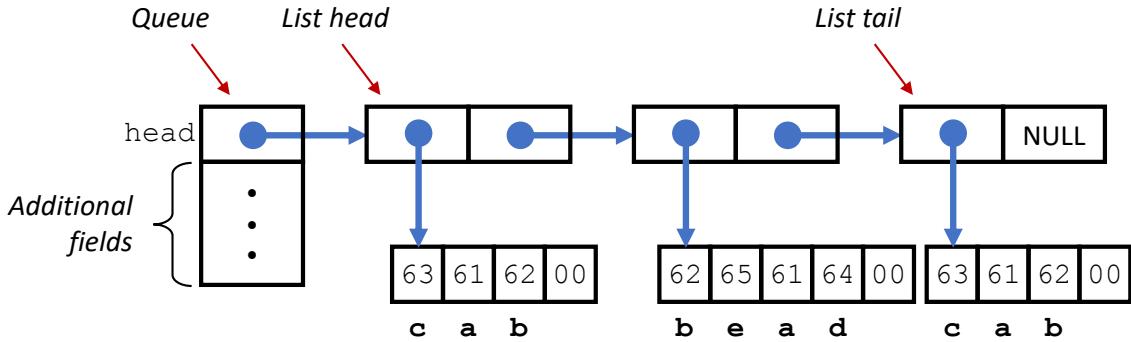


Figure 1: **Linked-list implementation of a queue.** Each list element has a `value` field, pointing to an array of characters (C’s representation of strings), and a `next` field pointing to the next list element. Characters are encoded according to the ASCII encoding (shown in hexadecimal.)

```
/* Linked list element */
typedef struct list_ele {
    char *value;
    struct list_ele *next;
} list_ele_t;

/* Queue structure */
typedef struct {
    list_ele_t *head; /* First element in the queue */
} queue_t;
```

These are combined to implement a queue of strings, 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 pointer to a string and a pointer to the next list element, respectively. The final list element has its `next` pointer set to `NULL`. You may add other fields to the structure `list_ele`, although you need not do so.

Recall that a string is represented in C as an array of values of type `char`. In most machines, data type `char` is represented as a single byte. To store a string of length l , the array has $l + 1$ elements, with the first l storing the codes (typically ASCII¹ format) for the characters and the final one being set to 0. The `value` field of the list element is a pointer to the array of characters. The figure indicates the representation of the list [“cab”, “bead”, “cab”], with characters a–e represented in hexadecimal as ASCII codes 61–65. Observe how the two instances of the string “cab” are represented by separate arrays—each list element should have a separate copy of its string.

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 has value `NULL`. An *empty* queue is one pointing to a valid structure, but the `head` field has value `NULL`. Your code will need to deal properly with both of these cases, as well as queues containing one or more elements.

¹Short for “American Standard Code for Information Interchange,” developed for communicating via teletype machines.

5 Programming Task

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

queue_new: Create a new, empty queue.

queue_free: Free all storage used by a queue.

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

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

queue_remove_head: Attempt to remove the element at the head of the queue.

queue_size: Compute the number of elements in the queue.

queue_reverse: Reorder the list so that the queue elements are reversed in order. This function should not allocate or free any list elements (either directly or via calls to other functions that allocate or free list elements.) Instead, it should rearrange the existing elements.

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.

For functions that provide strings as arguments, you must create and store a copy of the string by calling `malloc` to allocate space (remember to include space for the terminating character) and then copying from the source to the newly allocated space. When it comes time to free a list element, you must also free the space used by the string. You cannot assume any fixed upper bound on the length of a string—you must allocate space for each string based on its length. **Note:** `malloc`, `calloc`, and `realloc` are the only supported functions in this lab for memory allocation, any other functions that allocate memory on the heap may cause you to lose points.

Two of the functions, `queue_insert_tail` and `queue_size`, will require some effort on your part to meet the required performance standards. 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 operation will require only a fixed number of steps, regardless 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. Please work on finding a solution better than the $O(n^2)$ solution for all of the functions.

Your program will be tested on queues with over 1,000,000 elements. You will find that you cannot operate on such long lists using recursive functions, since that would require too much stack space. Instead, you need to use a loop to traverse the elements in a list.

6 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, which you can type into the `qtest` program:

```
# Demonstration of queue testing framework
# Initial queue is NULL.
show
# Create empty queue
new
# Fill it with some values. First at the head
ih dolphin
ih bear
ih gerbil
# Now at the tail
it meerkat
it bear
# Reverse it
reverse
# See how long it is
size
# Delete queue. Goes back to a NULL queue.
free
# Exit program
quit
```

You can also run `qtest` on an entire trace file all at once, as follows:

```
linux> ./qtest -f traces/trace-eg.cmd      # This is the example trace.
linux> ./qtest -f traces/trace-01-ops.cmd  # This is the first real trace.
```

If you try to test the starter code, you will see that it does not implement many of the operations properly.

The `traces` directory contains 15 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.

7 Evaluation

7.1 Autograder

Your program will be evaluated using the fifteen traces described above. You will be given credit (either 6 or 7 points, depending on the trace) for each one that executes correctly, summing to a maximum score of 100. This will be your score for the assignment—the grading is completely automated.

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 the driver directly with the command:

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

or with the command:

```
linux> make test
```

7.2 Style

This lab will not be style graded. The score you receive on Autolab will be your final score. However, your code for all labs **must be formatted correctly to receive points on Autolab**. For formatting your code, we require that you use the `clang-format` tool. To invoke it, run `make format`.

7.3 Memory safety

The `qtest` program will be compiled with AddressSanitizer. This is an instrumentation tool similar to Valgrind that detects common memory issues such as accessing invalid memory, calling `free()` multiple times on the same address, or leaking memory. Programs are instrumented with AddressSanitizer at compile time, so you do not need to do anything extra to take advantage of it. However, be aware that Valgrind and AddressSanitizer cannot be used at the same time.

When AddressSanitizer detects an error, it will print an error similar to the following, then exit the program:

```
==4662==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x6020
000000318 at pc 0x00000004e37e2 bp 0x7ffdbf037f30 sp 0x7ffdbf0376e0
READ of size 1025 at 0x602000000318 thread T0

#0 0x4e37e1 in __asan_memcpy (<...>/qtest+0x4e37e1)
#1 0x52e251 in queue_remove_head <...>/queue.c:160:9
#2 0x5276f9 in do_remove_head <...>/qtest.c:273:16
#3 0x52c0ae in interpret_cmnda <...>/console.c:224:14
#4 0x52be0a in interpret_cmd <...>/console.c:251:15
#5 0x52ca4a in cmd_select <...>/console.c:635:9
#6 0x52da21 in run_console <...>/console.c:722:9
#7 0x52876a in main <...>/qtest.c:527:16
#8 0x7fc4eb07eb96 in __libc_start_main /build/glibc-0TsEL5/glibc-2.
27/csu/../csu/libc-start.c:310
#9 0x41a829 in _start (<...>/qtest+0x41a829)
```

This list of function names is called a **stack trace**, representing the execution state when the error occurred. The lowest numbered frame #0 indicates the innermost function call: for instance, we can see that the error was caused by `__asan_memcpy`. However, the first function that is actually in our code is the frame #1. Therefore, we should look at the referenced location `queue.c:160:9` to find the error, which is referring to line 160 of `queue.c`.

Memory safety issues are a serious correctness issue, so you should make sure to fix any errors that are detected by AddressSanitizer. If you need help interpreting the output, feel free to ask for help on Piazza.

Note: If you find that Valgrind fails to run with your tests due to errors relating to "DWARF", include `-gdwarf-4` to the `CFLAGS` variable in the Makefile (if it does not already exist).

8 Handin

To receive a score, you should upload your code to Autolab, and your report to Elearning.

The Autolab servers will run the same driver program that is provided to you, and record the score that you receive. You may handin as often as you like until the due date.

Running the make command will generate a tar file, cprogramminglab-handin.tar. You can upload this file to the Autolab website.

IMPORTANT: Do not assume your submission will succeed! You should ALWAYS check that you received the expected score on Autolab. You can also check if there were any problems in the autograder output, which you can see by clicking on your autograded score in blue.

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