Project 0: C Programming Warm-Up

Out: Wednesday, September 24
Due: Wednesday, October 1 at 11:59 p.m.

Background

Despite incredible advances in programming languages over the last 30 years, most serious systems programming is still done in C.

Why is this? Because C gives the programmer more control and power over the code's execution than do other, higher-level languages like Java or even C++. Also, C typically has less runtime overhead than higher-level languages, which can translate into increased performance. Suppose you have a function that takes an integer and returns a double. In a strongly typed language, all you can do with this function is call it while passing an integer and treat the result as a double. Of course, you can do this in C. But you can also call it with no parameters, call it with 5 parameters, take the result and store it in an integer. Even better, you could treat the function as an array and read each instruction as an integer if you like. Or, you could call not the first instruction in the function, but maybe the second, or the third, or ... there is a reason why C is sometimes referred to as a "high-level assembly language".

What's bad about this freedom? Bugs. Forgot a parameter? Maybe you did it on purpose. Or maybe (and probably) not. In Java, the compiler shows you your mistake. In C, the compiler is very easy to please, but when you run the program, it fails, generally in a very cryptic way–"segmentation violation," and "core dumped" are the principle error messages. Both mean you made a mistake.

All of this means that you need to program carefully and deliberately in C. If you do this, you can write programs that are as well structured and clear as you can in languages like Java. But, if you don't, you'll quickly have a big mess on your hands. Hard to debug. Hard to read. Hard to modify.

There are lots of good references for programming in C. The primary one is "C Programming Language (2nd Edition)" by Kernighan and Ritchie.

Assignment

You should do this assignment on a Unix machine, which will provide you not only with the compiler (`cc`, or `gcc`, depending on the installation) but also a debugger (`gdb`). You can use any editor you like, but I recommend that you check out `emacs`, which has support for C programming and debugging.

Part 1. The Basic Queue

The queue is one of the most important data structures you'll be dealing with in this class. Consequently, it's a good one to start working with early.

For this part of the assignment, I will be providing you with a complete interface and mostly complete implementation for a queue. Starting with this, you will:

1. Find and fix two critical bugs in the implementation (`queue.c`). (I've put these bugs in after getting the code working). You will need to extend the test infrastructure (`main.c`) significantly and ensure that it functions correctly. You should include comments that make it clear any problems you fixed (either in `queue.c` or in `main.c`).

2. Implement the two declared, but not implemented, methods: `queue_reverse()` and `queue_sort()`. Both methods must work in-place: they can't create a new queue and move or copy elements from the original queue to build the result. The time efficiency of the sorting algorithm is not important, as long as it's something reasonable (not worse than $O(n^2)$, not better than $O(n \log n)$. `queue_reverse()` should execute in $O(n)$ time.

You must follow the coding style (indentation, naming, etc) that you find inside `queue.c` and `queue.h`.

Grading Criteria

2 points:	Perfect. Two critical bugs in `queue.c` found and fixed. New methods of queue implemented cleanly and correctly. Bug in `main.c` found and fixed.
1 point:	Slightly less than perfect.
0 points:	Quite flawed.

Part 2. The Hash Table

Following the same style for the queue, write the code that defines (.h) and implements (.c) a hash table.

The hash table allows you to store and retrieve values of any kind (pointers) based on a key value.

You're free to use any hash table implementation you like. You learned several variations, such as linear probing or separate chaining, in your data structures course.

The operations (in English) you need to define are:

- `Create`. Creates and returns a new hash table.
- `Set` hash function. Establishes the function to be used when computing a hash value based on a key.
- `Add` [key, item] pair. Inserts the given pair into the table.
- `Lookup`. Given a key, returns the associated item. Indicates failure in the event that the key is not present.
- `IsPresent`. Given a key, returns a boolean indicating that the key is present.
- `Remove`. Given a key, deletes the associated [key, item] pair from the hash table.

Grading Criteria

2 points:	Clean code. Works. Clearly demonstrates an understanding of the coding style laid out in Part 1.
1 point:	Code not so clean. Or, maybe doesn't work so well.
0 points:	Problems abound.