Warmup Exercise III: C programming

October 2, 2019

Introduction

Hopefully after the previous warmup exercises, you are all more comfortable with Linux command line, gdb, text editors, and Makefiles. The final exercise is designed to familiarize you to programming in C. Specifically, it's designed so that you practice writing C code using pointers and managing your own memory — you will be implementing a linked-list data structure in C.

First, obtain the starter code for this exercise by going to this GitHub link:
`https://classroom.github.com/a/7OQb4bFo`

Once you accept the assignment and clone the code base, you should find a list of files in the exe3 repository:

- `Makefile`
- `test-list.c`: the driver code for using list data type
- `list.h`: what the user of the list data type sees
- `list.c`: the implementation of list data type
- `sol`: the sample solution in the form of a binary executable; when in doubt, check how the `sol` executable behaves, and your program should implement the same behavior.

Makefile

A few notes about `Makefile` — if you don't know the basics of a `Makefile`, please work through exercise 2. Assuming you have done so, did you notice something different in this `Makefile`? In this `Makefile`, sometimes we use funny symbols in the command, such `$<`, `$^`, and `$@`. These are ***automatic variables*** that get expanded automatically depending on the pattern that matched to cause the rule to trigger. The `$@` expands to the name of the first prerequisite of the rule, where as the `$@` expands to the name of the target.

For instance, the rule

```
%.o: %.c
	gcc $(CFLAGS) -c $< -o $@
```

is used to compile from any `*.c` file into `*.o`, so the `*` symbol expands to whatever filename it happens to match. If the rule got triggered because we are compiling from `list.c` (the prerequisite) to `list.o` (the target), then `$<` is expanded to `list.c`, where as `$@` is expanded to `list.o`. Sometimes, a rule

may have multiple prerequisites (such as the rule with target `test-list`), in which case `$^` is used to expand to all the prerequisites with space between them.

You can find out more about automatic variables for `Makefile` here: `www.gnu.org/software/make/manual/html_node/Automatic-Variables.html`.

Notes on C programming

Before you begin, if you are not familiar with the concept of pointers, structures, and arrays, do read through the C tutorials posted on Piazza under resource page: `https://www.cprogramming.com/tutorial/c-tutorial.html` (especially the links under Pointers, Structures, and Arrays).

In this exercise, you will also need to manage your own memory. Whenever you want to create a new linked list node to insert into the list, you need to obtain memory to do that. In C, you invoke `malloc` to obtain a chunk of continuous memory, where the argument specifies how many bytes you need:

```
node_t *node = (node_t *) malloc(sizeof(node_t));
```

The call to `sizeof(node_t)` returns how many bytes one needs for a piece of data that's of type `node_t`. The `malloc` then allocates a piece of memory of that size, and return the *address* of that memory. Hence why the return value is of a pointer type (and in this case, a pointer to `node_t`).

It's always a good practice to free resource that you don't need anymore. Thus, whenever you delete a node from your linked list, you should free the memory that used to hold the list node value:

```
free(node);
```

Note that, node should be a pointer returned by `malloc` at some point earlier in the execution.

Notes on the codebase

When coding in C, typically the ***header*** file (i.e., `list.h`) contain information that is exposed to the end user — in this case, the person using your linked-list library implementation. On the other hand, the C file (`list.c`) should contain the actual implementation of the linked list, which will be hidden from the end user.

If you take a look at the content in `list.h`, it shows a list of function declarations supported by the list-link data structure (that you will implement). It also shows the `list_t` data type, which includes the usual fields that a linked-list ought to have — a `head` pointer pointing to the first element in the list, a `tail` pointer pointing to the last element in the list, and a `size` field indicating the number of elements in the list. Note that, however, the `struct node_t` type is opaque — it does not tell the end user how the linked list node is maintained and what fields it include. This is a good practice, because if the library implementor (i.e., you) ever want to change how the underlying linked-list is implemented, say, by changing the fields of the `struct node_t` type, it will not affect the end user as long as the interface specified in the header file stays the same. The end user, in order to use the linked-list, then simply needs to include `list.h` without knowing how it's implemented, such as done in the driver code `test-list.c`.

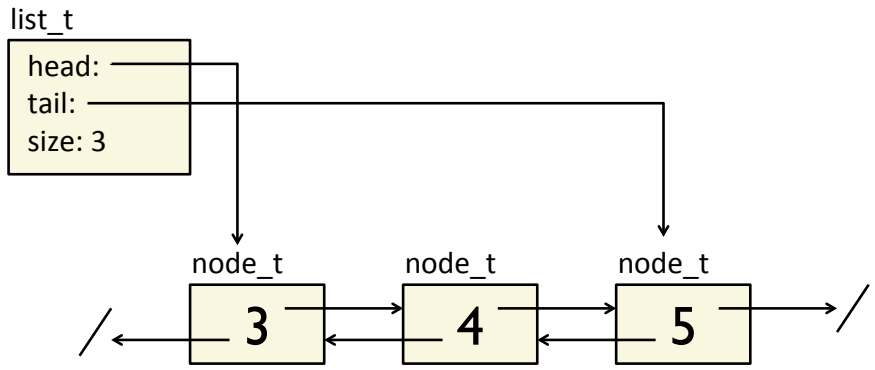

Figure 1: An illustration of a doubly linked list containing elements with values `3`, `4`, and `5`. The arrows are pointers, and the `/` character indicates a `NULL` pointer.

The `list.c` already includes a set of functions that you should implement, including the type declaration of `struct node_t`, although the implementation of the function bodies are mostly missing. Your job is to implement them. Currently the `struct node_t` type declaration assumes that the list implementation will be a doubly linked-list, such as ones shown in Figure 1, which allows for $O(1)$ list append and list prepend.

If you are unsure as to how to start, check out the linked list section in the C tutorial: `https://www.cprogramming.com/tutorial/c/lesson15.html`. It walks through an implementation that traverses a singly-linked list, which can help you get started.

Valgrind for testing your code

Since you manage your own memory in C, sometimes it's easy to make mistakes with respect to memory — you may accidentally free the same pointer twice; you can accidentally write to memory location that you are not supposed to; or you may forget to free memory that you no longer need (called memory leak).

The good news is, there are tools out there built for this purpose. `valgrind` is such a tool (and installed on Linuxlabs machines). Whenever you are done writing your code, you should invoke your program with valgrind to check for things like memory corruption and memory leak.

To see what `valgrind` does, try:

```
> valgrind ./sol
```

Do a few operations and then quit using option 7. Invoke `sol` with `valgrind` again, but this time, instead of quitting using option 7, quit by typing in `Ctrl-C` (hit the `Ctrl` key and then `C` while holding down the `Ctrl` key). `Ctrl-C` will terminate the `sol` program without allowing it to cleanup properly, and thus trigger a memory leak.

You should test your code to ensure that there is no memory leak when the program quits properly.