Lab 5

11 February 2021

In this lab, you'll work a little with linked lists, and see how to use the unit testing framework with pointer-based data.

First, start a directory for this lab and copy `Link.h` from the `/home/shared/162/opendsa/ch10/` directory into it. This is the class file for the `Link` class from the book.

Then, create a file `test_LinkFunctions.u` that, for now, just has a fixture that declares and builds at least three linked list structures (consult your notes and the board photos for some ideas how to write these).

Remember as you're doing this that the top part of the fixture should have decl-and-init statements that look like

> typename varname = initvalue;

and any additional setup, if needed, goes in the `setup` block.

Right now, run the handin script on your `.u` file so I can see what you wrote before lab:

```
handin cmsc162 lab5 test_LinkFunctions.u
```

In the rest of this lab, you'll be writing functions that operate on (pointers to) nodes that represent linked lists. Put them in a file called `LinkFunctions.h`, and test them with the `.u` file you've already made.

But first...

Command line FOTD: `grep`

The `grep` command is a general search tool that lets you find occurrences of some pattern in a whole batch of files. For instance, if you type

```
grep Link Link.h
```

you'll get a listing of every line on which the word `Link` shows up in the file `Link.h` (which is a lot!).

But `grep` is more powerful than that. Its first argument is what's called a "regular expression" or "regex", and lets you search for some pretty complicated things. You can get more information on this on your own, but a few quick tricks:

- By enclosing the pattern in (single) quotes, you can search for strings with spaces in them:

  ```
  grep 'T set' Link.h
  ```

- To match any amount of any text, use the "period asterisk" wildcard (note that for filenames on the command line we use asterisk by itself, but within a regular expression we need the period plus the asterisk):

  ```
  grep 'T.*inn' Link.h
  ```

- To match the beginning or end of the line, use caret (`^`) and dollar-sign (`$`) respectively.

  ```
  grep '^Link' Link.h
  ```

  would give just those lines that *start* with `Link`—and you'll notice there aren't any! The ones that "start" with `Link` actually have a few spaces first. Putting together some of what we've seen,

  ```
  grep '^ *Link' Link.h
  ```

  will find the ones that start with any number of spaces and then the word `Link`.

Vim FOTD: searching (and replacing)

From command mode, if you hit the forward slash key,[1] it's a little bit like colon mode: the cursor moves to the bottom of the screen and awaits further input. But what it's waiting for now is a regular expression to search for.

Having just explained regexes in the context of `grep`, there's not much more to explain here; they work essentially the same way. After the initial slash, you type a regex and hit enter, and Vim will find the next place in the file that matches that regex, or if there are none it will tell you that.

[1]Having trouble remembering which is forward slash and which is backslash? If you imagine them walking across the page from left to right, the forward slash is leaning forward: `/` And the backslash is leaning backward: `\`

Also inside command mode, the `n` command will repeat the previous search. So pressing `n` repeatedly will cycle through all matches in a file. Using `N` instead goes through matches in reverse order.

The `n` command together with the period command (which repeats the previous command) is a workhorse combination: first, search for a pattern and do something; then alternate `n.n.n.` until you've done your action every place that pattern occurs. Try it in `Link.h`: Let's say that instead of `T` as the stand-in name of the type this object contains, we prefer `Thing`. Press `/` and type `T` (and hit enter) to find some occurrence of that word. Then, type the command `cw` to "change word" and type `Thing` (and hit Escape) to complete the change. Now press `n` to go to the next occurrence, and press period to make the same change. Keep pressing `n` and `.` until you've made that change throughout the file.

You can either save and quit (if you prefer `Thing`), or save without quitting (using `:q!`), but other than that change, you should not need to further modify the `Link.h` file for this lab.

Finishing sum

In class we got this much on the board before the end of class:

```
int sum (shared_ptr<LinkNode<int>> head)
{
    ...
            sum(head->next)
    ...
}
```

In the file `LinkFunctions.h`, put at least the stub for that function—adjust it to work with the declarations in `Link.h`—and then add test cases for it to your `.u` file. (Remember, you can use your existing examples for this!) Then, go back and finish it off. Right at the end of class, we verbally laid out the following checklist of stuff to keep track of in a recursive function:

- How do we identify the base case? (Hint: usually the empty list)
- What do we return in the base case?

- What does the recursive call look like?
- How do we combine whatever we're doing to the current node with the recursive result?

In this case, the part we actually wrote on the board is exactly right for the recursive call, and we said out loud that the base case would be when `head` was `nullptr`, in which case we'd return zero. In the recursive case, what should you add to the sum of "the rest of the list" in order to make the sum of the entire list?

Implementing countMatch

Now, you'll use what we learned in class yesterday to write your own recursive function, making sure to test it, that behaves as follows:

> `countMatch` counts how many elements in the given linked list are exactly equal to the value represented by the given item.

Plan your way through the function design—header and test cases, and you may wish to create an additional example in your fixture to test this one.

As you write your function, keep all four steps of the checklist in mind and be sure you don't forget to handle one!

Implementing `negateAll`

Once you've got `countMatch` working, try developing a function `negateAll` that works as follows:

> `negateAll` produces a linked list that contains the negations of the numeric values in the given linked list.

Although the result of this function is itself a list (which you'll represent as a node pointer, just as we've done for the parameters of these functions), the function design process is the same and the function you write will have the same basic structure as the others we've written. (Your test cases may have to be a little more indirect: run the function, hold its result in a temp variable, and then check/expect various aspects of the result, e.g.:

```
  check (result) expect != nullptr;
  check (result->element()) expect == -7;
  check (result->next()) expect != nullptr;
  check (result->next()->element()) expect == 42;
```

and so on.

Other functions to think about

If you find yourself with extra time at the end of the lab, try your hand at writing

> `initials` extracts the initials from each string in the given linked list and combines them into a single string.

and

> `keepPositives` produces a linked list containing only those values in the given linked list that are positive.

Whether or not you get to writing these functions during lab, think about and write down test cases for them and bring them to class tomorrow.

Handing in

This is feeding into class Friday and an upcoming homework; do as much as you can and make note of any questions you have, and bring those with you to class, but no need to run `handin` this week.