Lab 5

25 September 2014

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

Before you get started, copy `Node.h` out of `/home/shared/162` into your working directory for this lab. This is the class file for the `Node` class from the book, slightly adapted for our use.[1]

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 Node Node.h
```

you'll get a listing of every line on which the word `Node` shows up in the file `Node.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 'void set' Node.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 'ItemType.*item' Node.h
  ```

- To match the beginning or end of the line, use caret and dollar-sign respectively.

[1] Specifically, I put it all into a single file for our convenience.

```
grep '^Node' Node.h
```

will give just those lines that *start* with `Node`.

Vim FOTD: searching (and replacing)

From command mode, if you hit the forward slash key,[2] 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.

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 `Node.h`: Let's say that instead of `ItemType` as the stand-in name of the type this object contains, we prefer `Thing`. Press `/` and type `ItemType` (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 `Node.h` file for this lab.

Planning `linkSearch`

Continuing from class, recall that we were planning a function with header

[2] 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: `\`

```
bool linkSearch (Node<char>* node, char value)
```

that determined whether the group of nodes starting at the given `Node` included the given `value`. (Note that this is a function, not a method of `Node` or any other class.)

You might want to look at the board photos from yesterday's class to remind you what your group did (and what the other group was up to). Remember that both of those implementations were still in the planning stages (i.e. still buggy), though!

There are three things we need to account for in the repetition of this search:

- We don't find it
  (the current node is `nullptr`)
- We find it
  (the current node has it as its value)
- We have to keep looking
  (continue on to the current node's `next`)

Mark up the above list with, on the one hand, how these parts map out to the different elements of iteration (init loop variable, keep-going condition, etc), and on the other hand how they map out to the elements of recursion (base case, etc). Consult your notes (or the left board photo from 12 September) to make sure you haven't missed anything.

Stub functions

The code for `linkSearch` will be separate from the `Node` class, so we'll need a separate `.h` and `.cpp` for it. Inside `nodefunctions.h`, you'll write the function prototype; and since you'll be doing two implementations, I'm going to have you write two nearly-identical prototypes (the only difference is their name):

```
bool linkSearchRec (Node<char>* node, char value);
bool linkSearchIter (Node<char>* node, char value);
```

(This file will also need to have the `#include` for `Node.h`.) Then, in the file `nodefunctions.cpp`, you'll write—for now—stub functions:

```
bool linkSearchRec (Node<char>* node, char value)
{
  return false;
}

bool linkSearchIter (Node<char>* node, char value)
{
  return false;
}
```

Examples and testing

Edit a file named `test_nodefunctions.u` or something similar. Start a test suite—refer back to Labs 2 and 3 for instructions on how the `.u` file is laid out—and in the test fixture, type in the two examples we wrote in class, or similar ones of your own devising. Add an example that refers to a link chain with a single element, and name it appropriately.

You should now have at least three examples: one that represents no node at all; one with a dead end node that contains just a single element; and one that (indirectly) contains at least three elements (that is, it contains one and points to further `Node`s with the other two elements).

A note about deleting

Strictly speaking, using `new` without somewhere making plans for a matching `delete` is a bit dodgy, as I said in class yesterday. Testing software such as unci provides a mechanism to "tear down" a test fixture, which is where calls to `delete` will normally go, but I choose to defer this until a bit later. So: file away in your brain that we will have to worry about `delete` at some point, but for now let's move on.

Testing `linkSearch`

Now that you've made the examples (and checked that they compiled, right?), you can encode the test cases for a `linkSearch` function. Make two test blocks, one called `emptyRec` and one called `emptyIter`, and fill them in:

```
test emptyRec
{
  check (linkSearchRec(emptyChain, 'K')) expect false;
}
```

This statement is almost identical to one of the ones you wrote on the board yesterday, except the call to `linkSearch` is now `linkSearchRec` (also, the letter being searched for is different, not that it matters). Your `emptyIter` test block should be the same except testing `linkSearchIter` instead, because both functions have the same interface and are supposed to behave the same.[3]

Write additional test blocks to encode the other test cases you wrote on the board. (You can have multiple test cases per block, but group them logically.) Add some test cases that make use of the one-element chain you created earlier.

Compile and run your test suite to make sure you haven't made any typoes or other mistakes. Of course, since we still only have stub functions for `linkSearchRec` and `linkSearchIter`, some of the tests should report failures (if not, you need more or better test cases!).

Writing `linkSearch`

In your `nodefunctions.cpp` file, you should now actually write the implementations you planned out earlier. If you get stuck on one, make sure that it at least compiles and then work on the other one for a while.

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.

[3] In fact, if you make use of the Vim features from last week (yanking many lines at once and pasting them) and this week (search and replace), you should be able to add the test for `linkSearchIter` with almost no *new* typing at all!