CS2110 Fall 2014
Homework 12

Andrew Wilder

Rules and Regulations

Academic Misconduct

Academic misconduct is taken very seriously in this class. Homework assignments are collaborative. However, each of these assignments should be coded by you and only you. This means you may not copy code from your peers, someone who has already taken this course, or from the Internet. You may work with others **who are enrolled in the course,** but each student should be turning in their own version of the assignment. Be very careful when supplying your work to a classmate that promises just to look at it. If he/she turns it in as his own you will both be charged.

We will be using automated code analysis and comparison tools to enforce these rules. **If you are caught you will receive a zero and will be reported to Dean of Students.**

Submission Guidelines

1. You are responsible for turning in assignments on time. This includes allowing for unforeseen circumstances. If you have an emergency let us know ***IN ADVANCE*** of the due time supplying documentation (i.e. note from the dean, doctor's note, etc). Extensions will only be granted to those who contact us in advance of the deadline and no extensions will be made after the due date.

2. You are also responsible for ensuring that what you turned in is what you meant to turn in. No excuses, what you turn in is what we grade. In addition, your assignment must be turned in via T-Square. When you submit the assignment you should get an email from T-Square telling you that you submitted the assignment. If you do not get this email that means that you did not complete the submission process correctly. Under no circumstances whatsoever we will accept any email submission of an assignment. Note: if you were granted an extension you will still turn in the assignment over T-Square.

3. There is a random grace period added to all assignments and the TA who posts the assignment determines it. The grace period will last **at least one hour** and may be up to 6 hours and can end on a 5 minute interval; therefore, you are guaranteed to be able to submit your assignment before 12:55AM and you may have up to 5:55AM. As stated it can end on a 5 minute interval so valid ending times are 1AM, 1:05AM, 1:10AM, etc. **Do not ask us what the grace period is we will not tell you**. *So what you should take from this is not to start assignments on the last day and depend on this grace period past 12:55AM.* There is no late penalty for submitting within the grace period. If you can not submit your assignment on T-Square due to the grace period ending then you will receive a zero, no exceptions.

General Rules

1. In addition any code you write (if any) must be clearly commented and the comments must be meaningful. You should comment your code in terms of the algorithm you are implementing we all know what the line of code does.
2. Although you may ask TAs for clarification, you are ultimately responsible for what you submit.
3. Please read the assignment in its entirety before asking questions.
4. Please start assignments early, and ask for help early. Do not email us the night the assignment is due with questions.
5. If you find any problems with the assignment it would be greatly appreciated if you reported them to the author (which can be found at the top of the assignment). Announcements will be posted if the assignment changes.

Submission Conventions

1. **Failure to follow these may result in a max of 5 points taken off**
2. All files you submit for assignments in this course should have your name at the top of the file as a comment for any source code file, and somewhere in the file, near the top, for other files unless otherwise noted.
3. When preparing your submission you may either submit the files individually to T-Square or you may submit an archive (zip or tar.gz only please) of the files (preferred). You can create an archive by right clicking on files and selecting the appropriate compress option in on your system.
4. If you choose to submit an archive please don't zip up a folder with the files, only submit an archive of the files we want. (See Deliverables).
5. Do not submit compiled files that is .class files for Java code and .o files for C code. Only submit the files we ask for in the assignment.
6. Do not submit links to files. We will not grade assignments submitted this way as it is easy to change the files after the submission period ends.

Overview

NOTICE: please read the WHOLE assignment before you begin coding.

You will be writing a linked list library. Specifically, you will be writing a **generic doubly-linked list with a head and tail pointer**. **You may not use a “sentinel” node or a dummy node i.e. a node with no data that starts out the list. Each node in the linked list must have valid data.**

DO NOT IMPLEMENT ANY OTHER TYPE OF LINKED LIST or else we will not grade it.

Your list struct will have a head and tail pointer and a size and only these. Your lnode will have a pointer to the next and previous node and data (which can be a pointer to any type, which is a **void***)

Part 1

The first part of this assignment is implementing the library. We have provided you two files: *list.c* and *list.h*. **Please review the definitions in list.h, as you may see these on the final.** You may not change the definitions in this file. We have also provided prototypes for all of the functions you will be implementing.

Next, turn your attention to *list.c*. We've provided you with function headers to all the linked list functions we would like you to implement. Implement these functions!

Make sure you read the comments before asking any questions. The comments tell you exactly how to implement the function.

<u>Function Pointers</u>

If you'll look closely at the files we've provided you, some functions take in pointers to functions as parameters. This is no mistake (so don't change what we've given you!), this is where the concept of function pointers comes into play. You will be using what you've learned about function pointers in class to manage this portion of the homework. To briefly put into perspective what you should be doing, take a look at this line in *list.c*:

int remove_front(list* llist, list_op free_func)

Since you're removing the front node of your list, this function needs to know the list to cut the head off of (represented by our parameter, llist) and it needs to know how to free the memory for the **data** that was allocated to that specific node (parameter free_func). Remember that each node has data that was given to you by the user. This function pointer will point to a function that was written by the person who passed in the data the node contained to tell you how to free that data. You **<u>will not</u>** be calling this function on the node (What will the user do with it?) but the data the node contains.

Again to reiterate you should only call free_func on the node's data and not the node itself. The user of your linked list library is the only person who knows how to free the data. You the library writer however is in charge of freeing the actual node allocated for the user's data. Be sure to do these steps in the right order, because accessing freed memory is a no-no.

<u>Hints</u>

- Read the comments and this assignment before asking a TA for help. If you do not, you will get referred back to the assignment description and comments.

1- Never leave dangling pointers. If you remove a node, remember to fix up the pointers on its neighbors so they point to the right spots.
2- Always keep track of where your pointers are pointing.
3- Be very careful to handle special cases, such as removing the only node in the linked list.
4- We will not be passing NULL to any of your functions.

Part 1.5 – Design Issues

The design of this linked list library is such that the user using your library does not have to deal with the details of the implementation of your library (i.e. the node struct used to implement the linked list). None of these functions return an lnode nor do any of these functions take in an lnode.

It is your responsibility to create the nodes and add them into the linked list yourself. Not the user. For example, to use the linked list library, I can decide that I want a linked list of person structs. I can then define these functions to work for a Person.

If I want to print the persons' data, I would write a print person function that matches list_op. If I wanted to print them all, I would call the traverse function passing in my print person function as a parameter.

If the user has things they need to free, then they will write their own free person function that also matches list_op. When the user is done with the linked list, they will call empty list which removes all of the persons from the linked list.

Part 2

Data structures in C must be tested with all those pointers flying everywhere, and it's hard to get them right the first time. For this reason, we are making it mandatory for you to test your code.

We have provided you with a file called *test.c* with which to test your code. It contains multiple test cases, all of which create, destroy, and/or modify a linked list. You are required to write more test code than we have given you, and you must test each function at least once!

Printing out the contents of your structures can't catch all logical and memory errors, so we also require you run your code through valgrind. If you need help with debugging, there is a C debugger called gdb that will help point out problems. A tutorial with examples of using these debuggers is included in this assignment in the "debugging.tar.gz" archive. We certainly will be checking for memory leaks by using valgrind, so if you learn how to use it, you'll catch any memory errors before we do.

Here is a small tutorial on valgrind: http://cs.ecs.baylor.edu/~donahoo/tools/valgrind/

Your code must not crash, run infinitely, nor generate memory leaks/errors.

Running your code

We have provided a makefile for this assignment that will build your project. However, note that there are TWO makefiles in this directory. If you just type *make* (or *make vba*), you will not be using the correct Makefile (you will be using the one for part 3). Here are the commands you should be using to use the correct Makefile:

1. To run the tests in *test.c*: **make run-test**
2. To run the tests in *test.c* in debug mode: **make run-debug** (be sure to run this command first: **make clean**)
3. To debug your code using gdb: **make run-gdb** (be sure to run the clean target first)
4. To run your code with valgrind: **make run-valgrind**

Debugging

If your code generates a segmentation fault then you should first run gdb on the debug version of your executable before asking questions. (We will not look at your code to find your segmentation fault. This is why gdb was written to help you find your segmentation fault yourself.).

Here are some tutorials on gdb

http://heather.cs.ucdavis.edu/~matloff/UnixAndC/CLanguage/Debug.html

http://heather.cs.ucdavis.edu/~matloff/debug.html

http://www.delorie.com/gnu/docs/gdb/gdb_toc.html

Getting good at debugging will make your life with C that much easier.

In addition to the debugging folder (which you should definitely give a read), I have provided some useful debugging macros for your use:

1) IF_DEBUG(statement)

Runs statement if compiled in debug mode (run-debug / run-gdb / run-valgrind)

Example usage:

```
IF_DEBUG(printf("HELLO 2110"));
IF_DEBUG(
{
        x = 3;
        x++;
        x = 7;
        x++;
});
```

2) DEBUG_PRINT(string)

Prints out string (in red to distinguish it) if compiled in debug mode
This will also print out the file and line the print occurs printing goes to stderr

```
DEBUG_PRINT("Hello 2110");
```

3) DEBUG_ASSERT(expression)

If compiled in debug mode if expression is false
The program terminates with a message and exits it also tells you where the assertion failed.

```
/* This will fail since 8^8 is 0 which is false*/
DEBUG_ASSERT(8 ^ 8);
```

```
/* This will pass since 8^7 is 15 which is true */
DEBUG_ASSERT(8 ^ 7);
```

You may alternatively use assert, but you must #include <assert.h>

Remember

1. Write the contents of all of the functions in *list.c*. Be sure to pay attention to special cases. Your code should never crash, run infinitely, nor should it leak.

2. Write more test cases in *test.c*. The provided test cases are not exhaustive.

3 Do not change any of the function prototypes in *list.h*, because it will cause your submission not to compile when we test it. Non-compiling code is an automatic zero.

Deliverables

Turn in ALL files needed to compile and execute your code. This includes:
-Your *test.c*, *list.c*, and *list.h*.
- The Makefile

You may alternatively submit a zip or tar.gz of your files, but please do not put them in a folder.

Your files MUST compile with the required 2110 flags, which are: -std=c99 -pedantic -Wall -Werror -O2

Remember, non-compiling homeworks receive a zero. Make sure you turn in everything you need! Remember to be safe and redownload your submission into a new directory and try compiling it.