Computer Science 381
Programming Unix in C
The College of Saint Rose
Winter Immersion 2016

# Lab 9: Data Structures

**Due: Monday, January 11, 2016**

This lab is a follow on to the previous two. You will write two more C data structures in the object-oriented style.

---

## Unix Utilities

Before we get into this lab's C programming tasks, we take a look at some of the most useful Unix utilities. The extensive set of simple but useful utilities, and the ability to string them together with pipes and scripts are the real power of a Unix environment. Experienced Unix users faced with a task often find that they can quickly put together a script, or even a single command that performs the task, avoiding the need to find (or worse yet, write) a single program to perform it. This is especially true when the task involves processing text files or files with lots of numbers.

Back in the first lab, you looked into the functionality of a list of Unix commands. We will look more closely at some of those, plus a few more, in this lab and in the next.

### Pipes

Input and output redirection, which you've already been doing this semester, is one of the great sources of the power of the Unix command line. You have seen that you can have a program that is expecting input from the keyboard (standard I/O functions like `getchar` and `scanf` that read from `stdin`) get its input instead from the contents of a file. For example, in the `inputadder` program from earlier this semester, you could have a set of numbers to be added in a file `mynumbers.txt` and without changing your program to know that the input is going to be in a file rather than be typed in at its prompts by issuing the command:

```
./inputadder < mynumbers.txt
```

You have been using output redirection all semester, at least for the "output capture" questions in the labs.

```
ls -laR > ls.out
```

Now if you wanted to take the output of one program and use it as the input to another, you could use a temporary file as a way to store that output from the first program and provide it as input to the second. For example, if you have one program that generates a list of numbers (for whatever purpose), and you want to add those numbers up with your `inputadder` program:

```
./gennumbers > tempfile
./inputadder < tempfile
```

While this would work, it has some problems. We need to pick a name for a file that doesn't already exist. We need space in the filesystem to store the file. We will want to remember to remove the file afterward.

Unix provides the ability to attach the output of one program directly to the input of another using a *pipe*. For the example above, your command line would be:

```
./gennumbers | ./inputadder
```

In addition to avoiding the need for the temporary file, this can be done much more efficiently behind the scenes. The first program can still be running while the second starts its work.

There's no reason to limit this to just two commands in a pipeline. For example, suppose we have an input file `namelist.txt` that contains an unsorted list of names, one per line. We want to consider only those names that contain the word "john" anywhere in the name, and we want to print out the last three alphabetically from that group. This pipelined command line would do it:

```
grep -i john namelist.txt | sort | tail -3
```

**? Question 1:**
Explain what's happening in each component of the above command pipeline and how they combine to work as described. (3 points)

For the following lab questions, describe the effect of the given command pipeline.

**? Question 2:**
`ls -1 | wc -l` (1 point) Note: the parameter to `ls` is the number '1' while the parameter to `wc` is the letter 'l'.

**? Question 3:**
`head -10 myfile | tail -1` (1 point) The parameter to `tail` is the number '1'. We assume that the file `myfile` contains at least 10 lines.

For the following lab questions, give a single Unix command pipeline that would accomplish the task described.

**? Question 4:**
Print the number of files in the current directory. (1 point)

**? Question 5:**
List all of the files in a directory that were last modified on Halloween. (2 points) Hint: start with `ls -la`.

**? Question 6:**
Given a file with a list of several hundred words, one per line, print the single word that occurs between lines 100 and 200 of the file which is last alphabetically. (2 points)

---

## Programming Assignment: A Queue of Ratios

Create a queue structure and corresponding functions to operate on queues in C that holds `ratio` values. You may use the `ratio` structure from the `ratios` examples. Again, include an appropriate header file, implementation file and a file containing a `main` function that tests your implementation. Also include a `Makefile` that compiles your queue implementation and your testing code.

If you have a working queue of `int` values from the previous lab, you will be able to do this one by changing the field that stores the value in each of your queue elements to store a "`ratio *`" instead of an `int`, and the appropriate parameters and return values of the queue functions to use "`ratio *`" as needed. Be sure that you are using *pointers to* `ratio` not just `ratio` throughout, because C will not let us use assignment statments on structures, but will with pointers to structures. Your queue code should never allocate or free any `ratio`, just manage `ratio`s that are allocated and freed by whoever calls your queue's functions.

---

## Programming Assignment: A Priority Queue of Ratios

Next, create a priority queue structure and corresponding functions to operate on priority queues in C that holds `ratio` values. Your priority queue should remove the smallest ratio when an item is removed. Again, include an appropriate header file, implementation file and a file containing a `main` function that tests your implementation. Also include a `Makefile` that compiles your priority queue implementation and your testing code.

Some thoughts about this program:

- Your priority queue should be based on a linked list again. You might find it useful to use your queue program from the previous part as a starting point.
- There are many ways to implement a priority queue. When using a singly-linked list implementation, we can either take care of the "priority" part when adding to the structure or when removing. If we do a simple add, then the remove would require a search for the smallest value to be removed from the list and returned. If we do the work in add, that is, we make sure the list is sorted after every add operation, the remove becomes simple. I recommend the latter.
- Assuming you are taking my advice, the only significant change will be in the add (enqueue) function. Think of it in cases, and handle each separately. The following cases are likely to arise:

    1. You are adding to an empty priority queue. Great - we just add at the head of the queue.

2. You are adding a value smaller than any in the queue. OK, so not too bad. We make a new node with this new value become the new head, and it's next is the old head.
3. You are adding a value that needs to end up somewhere in the middle. This entails a search. Once you find the node that your new value should follow, it's a matter of hooking up the pointers to that node to your new one, and your new one to that node's old successor.
4. You are adding a value that should be at the end of the queue. Depending on your implementation, the case above might handle this, but be sure you're checking that you never run off the end of the list, else you will encounter the dreaded `Segmentation fault.`

- Draw lots of pictures. Work through easier cases first. If you can't add the first element to the structure, you certainly can't add the second.
- Remember your debugging techniques. Insert debugging printouts to figure out where your program is failing. Use `fprintf(stderr,...)` to ensure you don't get confused by buffered output not appearing before the operating system kills your process for causing a segmentation fault. Remember `gdb` also might be able to help you track down a bug quickly.

---

## Summary

These programs and their `Makefiles` are worth a combined 40 points as broken down below.

Reference solutions to all programs are available on mogul in `/home/cs381/labs/ooc2`.

---

## Submission

Please submit all required files as email attachments to *terescoj@strose.edu* by Monday, January 11, 2016. Be sure to check that you have used the correct file names and that your submission matches all of the submission guidelines listed on the course home page. In order to email your files, you will need to transfer them from mogul to the computer from which you wish to send the email. There are a number of options, including the `sftp` command from the Mac command line.

---

## Grading

| Grading Breakdown | |
|---|---|
| Lab questions | 10 points |
| `ratio` queue correctness | 12 points |
| `ratio` priortity queue correctness | 12 points |
| Program error checking | 3 points |
| Program memory management | 4 points |
| Program documentation | 5 points |
| Program style | 3 points |
| `Makefiles` | 1 point |