

# Programming Assignment 3

---

**Due** Oct 26 by 11:59pm      **Points** 100

---

## Overview

In the previous assignment you were asked to write a program that interacted with an emulated filesystem. One of the goals was to emphasize that directories provide structure, but are otherwise just files with a particular format (and role).

This assignment requires that you write a program that interacts with the system's actual filesystem (via system calls), that traverses a portion of the directory hierarchy, and that manages scarce resources. For this assignment you will write a program that recursively prints directory contents in a tree-like format.

## Course Learning Objectives

This assignment addresses the following course learning objectives.

### Direct


- Read and write complex programs in the C programming language.
- Transition to upper-division coursework where programming is a tool used in the exploration of course-specific content.
- Transition from following stepwise, prescribed decompositions to independent problem solving.
- Discuss the architecture of the UNIX operating system from a system-programmer's perspective and be able to write programs that use operating system services (system calls) directly.
- Distinguish language features from operating system features.
- Gain experience with low-level programming in the UNIX environment.
- Consider basic resource management in the development of programs.

### Indirect

- Use standard UNIX user-level commands and the UNIX software development environment.

## Setup

Clone the repository.

- Accept the [GitHub Classroom Assignment 3 invitation](https://classroom.github.com/a/SqolCyyB)    
(<https://classroom.github.com/a/SqolCyyB>).

## Requirements

There is, in effect, only a single task, so the following outlines requirements for your program.

## Functional Overview

Your program, named `{tt tree}`, must take as command-line arguments optional switches and any number of file/directory names. For each given file name, your program must only print that name. For each directory name, your program must print the contents of that directory (and, recursively, each subdirectory) in a tree-like format. If no file names or directory names are given, then list the contents of the current directory.

As an example, your program output should look as follows:

```
% ./tree tmp
tmp
|-- alpha
|-- bob
|-- tmp
|   |-- alpha
|   |-- bob
|   |-- burrito
|   |-- tmp
|       |-- bob
|       |-- zebra
|-- zebra
```

The format of your program's output should be very similar to that shown above. This tree shows the contents of the `tmp` directory and each subdirectory below it. The contents of each directory must be sorted alphabetically.

**Note:** When processing a directory's contents, do not recursively follow the two special files `.` and `..` (consider why not). Also, do not follow symbolic links (but do print the link name itself).

## Command-line Arguments

Your program must support the following two command-line switches. If given, these switches will immediately follow the program name (and precede any specified files). Your program must allow the switches to be provided in any order.

- Hidden Files [-a]: By default, your program should not include "hidden" files (those beginning with a `.`) in its listing. The `-a` switch causes "hidden" files to be processed (except for the two special files `.` and `..`).
- Size [-s]: The `-s` switch causes the size for each file to be printed. For example,

```
./tree tmp
tmp [size: 192]
|-- alpha [size: 0]
|-- bob [size: 0]
|-- tmp [size: 224]
|   |-- alpha [size: 11]
|   |-- bob [size: 23]
|   |-- burrito [size: 0]
|   |-- tmp [size: 96]
|   |   |-- bob [size: 13]
|   |-- zebra [size: 13]
|-- zebra [size: 17]
```

## Resource Management

All programs must properly manage the resources they use. In this respect, systems programs differ only in that the resources they must manage are often more scarce than those used by general applications. To address this issue, one will often use more readily available resources (e.g., memory) to avoid exhausting other resources.

If you are not careful in your implementation, then your `tree` program will likely run out of file descriptors and be unable to continue. Do not allow this to happen. Your program will be tested with a very deep, very contrived directory structure (i.e., it will be much deeper than the number of files that a process may open) or with the limit for the number of open files set artificially low (you should do this for your testing).

## Useful Functions

You might find the following functions to be of use (this is not to say that you will need them all or that you will not use others)

```
chdir, fchdir, opendir, qsort, readdir, stat, lstat
```

## Deliverables

- Source Code (and Makefile) - Push all relevant source code and an appropriate Makefile to your repository.

- **Code Review Monitoring** - This is not really a deliverable, but be sure to monitor the github messages for comments on your code and then address them as appropriate (certainly meeting with me if anything is unclear or in dispute).

<b>tree</b>			
<b>Criteria</b>	<b>Ratings</b>		<b>Pts</b>
Basic Functionality	<b>50 pts</b> <b>Full Marks</b>	<b>0 pts</b> <b>No Marks</b>	50 pts
Size Switch	<b>10 pts</b> <b>Full Marks</b>	<b>0 pts</b> <b>No Marks</b>	10 pts
Hidden Files Switch	<b>10 pts</b> <b>Full Marks</b>	<b>0 pts</b> <b>No Marks</b>	10 pts
Resource Management memory, file descriptors	<b>20 pts</b> <b>Full Marks</b>	<b>0 pts</b> <b>No Marks</b>	20 pts
Error Handling	<b>10 pts</b> <b>Full Marks</b>	<b>0 pts</b> <b>No Marks</b>	10 pts
			<b>Total Points: 100</b>