ECS 40 **Program #1 (55 points, my time 2:30)** **Winter 2015**

Due: Wednesday, January 21st at 11:59pm using handin to p1 directory of cs40a.
New concepts: ddd. Name of executable: funix.out
File names: authors.txt, Ins.c, Makefile, funix.out, funix.cpp, funix.h, directory.cpp, directory.h, permissions.cpp, and permissions.h

authors.txt should be in the format:
First line: <e-mail of first partner> <space> <name of first partner, last name first then comma then first name>
Second line (if needed): <e-mail of second partner> <space> <name of second partner in above format>

ddd Tutorial (10 points)

Follow the directions of the DDD tutorial available online at http://heather.cs.ucdavis.edu/~matloff/Debug/Debug.pdf. Each person must do the tutorial individually. The authors.txt for the partner that is only submitting Ins.c should contain only one name. You will find Ins.c in ~ssdavis/40/p1. When done completely debugging Ins.c, handin it.

There are at least four ways to gain access to ddd:

1. Go to the basement of Kemper and select DDD Debugger from the Programming menu.
2. To use ddd at home under Windows, on programs developed at home.
   - 2.1. Install cygwin (available for free from cygwin.com) with g++, openssh, and ddd. The selection of ddd should automatically install the X windowing server for cygwin.
   - 2.2. Once cygwin is installed, type **xinit&** at the cygwin command prompt to open an X window, and then type **ddd** at the prompt.
3. To use ddd at home under Windows, on programs developed in the CSIF.
   - 3.1. Install cygwin with at least the X server (I would still suggest installing g++ and ddd).
   - 3.2. Once cygwin is installed, type **xinit&** at the cygwin command prompt to open an X window.
   - 3.3. Type ssh –X username@CSIF_computername
   - 3.4. Once you have logged into the CSIF computer, change to the appropriate directory, and then type **ddd&**
4. To use ddd at home under MacIntosh OS X, on programs developed in the CSIF
   - 4.1. Open an X term. (See the MacIntosh help to install the X package)
   - 4.2. Type **ssh –X username@CSIF_computername**
   - 4.3. Once you have logged into the CSIF computer, change to the appropriate directory, and then type **ddd&**

FUNIX Project (45 points)

For this assignment and p3 follows, you will be implementing a subset of the UNIX file commands within an artificial directory structure. We will call our operating system FUNIX, though how much fun it will be has yet to be determined. For this assignment, you will only be working with directories. Though the source filenames end in ".cpp," they may be written purely in C code. In Program #3, you will be converting this program to C++. For this assignment you will have three typedeffed structs: Funix, Directory, and Permissions. You will find main.cpp, funix.h, and my funix.out in ~ssdavis/40/p1.

The Funix struct operates as the operating system. It writes the prompt (which includes an absolute path to the current directory), reads a command, and then processes the command. It continues through this cycle until it reads a legitimate exit command. Here is the syntax of the legal commands:

Command syntax	Interpretation
cd directory_name	Change to the named directory. If "..", then change to the directory's parent
exit	Exit from FUNIX
ls [-l]	List contents of the current directory. The -l option displays the permissions, time of modification, and name of all directories. The name of the current directory is listed as "."
mkdir directory_name	Create a directory with the name specified.
umask octal	Set the umask to the octal value given. Initialized to 0.

If the entered command is not found, FUNIX should give an error message. FUNIX should also give an error message when a legal command has the wrong syntax or attempts an illegal operation. The error messages should match that of my funix.out.

1. Funix struct contains a Directory* named currentDirectory, int umask, and int time.
   1.1. The initial Directory is named "/" which has rwx permissions and was created at time 0.
   1.2. The initial umask is 0.
   1.3. Time is based on the number of commands (valid and invalid) entered since the start of the program. With the first command of the session beginning with time one.
2. Directory struct contains
   2.1. A dynamically allocated char* that holds its name.
   2.2. An int that holds the time of its most recent modification.
   2.3. A pointer to a dynamically allocated array of its immediate subdirectories, and an int holding how many subdirectories actually exist.
      2.3.1. There can be at most 3 subdirectories in each directory, though this specification should be easy to change in your program by declaring it as #define at the top of directory.h.
      2.3.2. Initially the pointer is NULL, and is reallocated each time the number of directories is changed.
   2.4. A pointer to its parent Directory.
      2.4.1. You will use this pointer in a recursive function to print the absolute directory path used in the prompt.
   2.5. A Permission struct.
   2.6. Hint: If you declare **typedef struct Dir{..} Directory**, then you can use a "**struct Dir***" within the typedef for your array and parent pointers.
3. Permissions struct contains a short that is simply a record of the read, write, and execute permissions of each directory in octal format.
   3.1. Permissions are maintained, but do not stop the operation of any command.
   3.2. When a Directory is created its default permissions are rwx, which is 7. The umask value is applied to these default permissions to determine its actual permissions.
4. Further specifications:
   4.1. All structs must be passed as pointers, and only one type of struct pointer may be passed to a function. You should place all functions associated with a struct in its corresponding file.
   4.2. You may assume that the user will never enter a command longer than 79 characters.
   4.3. All functions must be implemented in .cpp files.
   4.4. Use const where possible.
   4.5. Enclose all of your header file code in #ifndef .. #endif preprocessor blocks as shown in my funix.h.
   4.6. Your Makefile must use g++ with –ansi, -Wall, and –g on all compiling lines.
   4.7. You may not change main.cpp, nor funix.h.
   4.8. The style of your program must match that of the Programming Standards.
5. Suggestions:
   5.1. To read an entire line all at once use fgets() with stdin. You will have to eliminate the \n that is at the end of the line somehow. To parse the line use the strtok() function of <string.h>
   5.2. To safely use NULL, you should include <stdlib.h>.
   5.3. atoi() and isdigit() are useful functions for dealing with the umask command.
   5.4. Develop your code using stubs, and in a top down fashion. A stub is a function that has no code in its body, except possibly a return statement. A stub should be able to compile without any warnings. When you do provide code for a stub, you may need to write new stubs for functions it calls. After writing the code for a stub, your program should be able to compile without warnings and run!
      5.4.1. Initially write stubs for all the functions mentioned in funix.h. Hint: copy funix.h into funix.cpp, and then just add {} (with end of function comments) and return values as needed. Have processCommand(), and eXit() return 0.
      5.4.2. Here is an ordering for implementing those functions. I did not fill in the bodies of the Directory and Permission stubs until I had completed all of these. When calling other Funix functions, I copied their signatures for the call, and then eliminated the data types.
         5.4.2.1. run: Calls init(), and then calls getCommand() and processCommand() in a loop until a proper exit command.
         5.4.2.2. init: Sets time and umask, creates the root directory, and then initializes the root directory by calling the Directory::createDirectory() function with appropriate parameters, including the umask.

5.4.2.3. getCommand: Calls writePrompt(), and reads in an entire line from the keyboard.
5.4.2.4. writePrompt: Calls Directory::showPath() function, which is recursive, with the currentDirectory, and then appends " # ".
5.4.2.5. processCommand: Parses the command line using strtok(), compares first argument with a static list of legal command strings, uses a switch statement based on the position in the list to call the proper command function, or print error message. This is the work horse function.
5.4.2.6. eXit: Ensures there a no extra arguments, else prints usage and returns 1.
5.4.2.7. ls: Calls the Directory::ls() function with the currentDirectory, argument count, and arguments.
5.4.2.8. mkdir: Calls the Directory::mkdir() function with the currentDirectory and other parameters.
5.4.2.9. umask: Checks the number of arguments, checks the format of the umask, and sets the umask if proper, else prints a usage or error message.
5.4.2.10. cd: Calls the Directory::cd() function with the currentDirectory, and other parameters.

5.4.3. Now write the Directory and Permissions functions one at a time in this order: createDirectory, Permissions::createPermissions, showPath, mkdir, ls, Permissions::printPermissions, cd.

```
[ssdavis@lect1 p1]$ funix.out
/ # mkdir first
/ # ls -l
rwx 1 first
/ # cd first
/first/ # mkdir second
/first/ # ls -l -l
usage: ls [-l]
/first/ # l
l: Command not found.
/first/ # ls -l
rwx 4 second
/first/ # umask 2
/first/ # mkdir third
/first/ # mkdir third
mkdir: cannot create directory 'third': File exists
/first/ # mkdir fourth
/first/ # mkdir fifth
mkdir: first already contains the maximum number of directories
/first/ # ls
second third fourth
/first/ # ls -l
rwx 4 second
r-x 9 third
r-x 11 fourth
/first/ # cd ..
/ # cd too much
usage: cd directoryName
/ # cd ..
/ # exit now
usage: exit
/ # exit
[ssdavis@lect1 p1]$
```

```
[ssdavis@lect1 p1]$ cat main.cpp
// Author: Sean Davis

#include <stdlib.h>
#include "funix.h"

int main()
{
  Funix *funix = (Funix*) malloc(sizeof(Funix));
  run(funix);
  free (funix);
} // main()
[ssdavis@lect1 p1]$


[ssdavis@lect1 p1]$ cat funix.h
// Author: Sean Davis
#ifndef FUNIX_H
  #define FUNIX_H

#include "directory.h"

#define COMMAND_LENGTH 80
#define NUM_COMMANDS 5

typedef struct
{
  Directory *currentDirectory;
  int umask;
  int time;
} Funix;

void cd(Funix *funix, int argCount, const char *arguments[]);
  // calls cd() with currentDirectory as one of its parameters
int eXit(Funix *funix, int argCount, const char *arguments[]);
  // checks "exit" command, returns 0 on proper exit
void getCommand(Funix *funix, char *command); // writes prompt and reads command
void init(Funix *funix); // creates currentDirectory, and sets umask and time
void ls(Funix *funix, int argCount, const char *arguments[]);
  // calls ls() with currentDirectory as one of its parameters
void mkdir(Funix *funix, int argCount, const char *arguments[]);
  // calls mkdir() with currentDirectory as one of its parameters
int processCommand(Funix *funix, char *command); // returns 0 on proper exit
void run(Funix *funix); // reads and processes commands in a loop until proper
exit
void umask(Funix *funix, int argCount, const char *arguments[]);
  // checks "umask" command and executes it if it is proper
void writePrompt(Funix *funix); // shows path and '#'
#endif
[ssdavis@lect1 p1]$
```