Anil Neerukonda Institute of Technology & Sciences
Department of Computer Science & Engineering (AI &ML, DS)

CSE117 Problem Solving with C

Date: October 20, 2022 **Time: 12:40p.m to 3:20p.m.**

Handout - Lab Session 1

Linux will be used for most of the courses in this program. It is in general, very easy to use, and also probably the most programmer friendly operating systems. Ubuntu Linux is highly recommended.

As a CS[AI & ML, DS] student, you will be spending a lot of time on the computers for coding, assignments, and other stuff. You must be able to type quickly and without looking at the keyboard. Practice typing, there are a number of tools available online.

Most of the problems selected for this lab can be solved using a simple program and you will appreciate the power of programming.

Objective :

- Describe how a Unix file system is organized.
- Develop familiarity with basic Unix commands.
- Be comfortable working in a Unix environment.

Pre Lab : Read the UNIX concepts and get familiar with them.

During Lab : Run the Unix commands given in the handout. Once you get comfortable with the UNIX environment, solve the exercise problems.

Post Lab : Take the quiz.

UNIX Concepts

Directory
The UNIX concept of a directory is just like the Windows/Mac concept of a folder. Directories hold files and other directories. The UNIX system contains lots of directories in a nested structure.

Subdirectory
A subdirectory is simply a directory that is inside another directory. Since UNIX only has one top level directory, called the root directory, every other directory can technically be called a subdirectory.

Root directory
The top of the directory hierarchy on any UNIX system. It is represented by the directory named with just a slash, as in /.

Current or Present working directory
The current or present working directory is the default directory for UNIX commands. For eg., using ls by itself lists files in the current directory. If you want to access files other than in the current directory, you have to use either a relative pathname or a full(absolute) pathname.
When you login, your current directory is your home directory.

Home directory
Your home directory is the directory allotted for all your stuff. It, and the files and subdirectories under it, are all yours.

Path** or **Pathname
A *path* or *pathname* is simply the set of directories that have to be traversed to get to a file. In other words, it is a way of giving an address for the location of a file, like a mailing address for a letter.

An example path would be:

cs2110/lab_1/p1.c

Typically you'd use a path to tell a command what file to operate on. For example,

ls -l cs2110/lab_1/p1.c

uses ls to list information about the file p1.c. Since, in this case, p1.c is not in your current directory. you must give a path with the file name.

Full** or **Absolute pathname
A *full pathname* is just a path that starts at the root directory, i.e., ***begins with*** the slash (/) character.

Any example full path is:

/usr/bin/man

which is the location of the man command.

Command Flag** or **Option** or **Switch
Command flags are typed at the prompt as part of a command. eg., ls -l, in which case the flag -l gives a longer set of information. Flags alter the default behavior of commands. For eg., the command ls by default, just lists the names of the files. With the -l flag, it gives you more information about each file(or directory).
Flags normally start with a minus sign(-) and consist of a single letter, like 'l' or a whole word, as in -debug.

Text file/Binary file
A *text file* is just like any other file, but it mainly contains letters, digits and punctuation characters. So, a textual e-mail message or the source code for a program is often stored in a text file. Since the coding we use for letters, etc. (in most computers) is the *ASCII* (pronounced *ask-key*) code, *text files* are also sometimes called ASCII files.

Types of files that are not text files are executables and word processor documents and are called *binary files*. When you try to view binary files with text file commands, they look like garbage. Again, a *text file* only differs from a binary file in that information is stored in it using the typical symbols that you'd see on a typewriter.

Source Code
A *source code* file is a file that contains program code in C (or some other language). For C, these files usually have names ending in **.c** (and sometimes in **.h**). These files cannot be *run* on the computer. In order to get an executable, which can be run, you must compile and link the source code.

Executable
An *executable* is a file that contains a program that is ready to run. When using ls to view files, these executables might have an asterisk (*****) after their names (though the ***** is not part of the file's name).
To run an executable, you just type its name.

Compiling and Linking
Compiling and linking means to turn source code into an executable that can be run on the computer.

Compiling and linking are actually separate procedures, though often you'll use one program to both compile and link. Technically, *compiling* puts all the source code into an intermediate form and *linking* links all the intermediate pieces and special libraries into a single executable.

Often we'll use the term *compiling* for both compiling and linking.
We'll compile and link our C programs by using the gcc compiler or by using the make utility.

Debugger
A *debugger* is a program that allows you to examine what an executable is doing while it is running. It is useful in finding errors in programs that compile fine, but that don't do what you want them to.

Our debugger is gdb.

Returning to your home directory:
Finally, since we are done with the files in the directory for now, let's go back up to our home directory. Since you are one directory below your home directory, you can just type:
cd **..**

to go up one level. Dot dot (..) is a way to refer to the directory above.
We could also use:
cd ~

since tilde (~) is shorthand for your home directory at the UNIX prompt (and in some applications, like Emacs). This works even if our home directory is not right above where we are.
Finally, we could also just use:
cd

by itself, which also goes to your home directory.
You can confirm where you are by using **pwd**.

UNIX Commands

Below are brief descriptions of the UNIX commands you should know. Remember that you can get more information on a command via the UNIX Manual Pages.

The commands in the examples below are always the first word, while the rest of the words are arguments to the commands.

Task: Make a directory

Example:

mkdir hw1

mkdir makes a directory that can hold files and other subdirectories. Above, we issued a command to make a directory called *hw1* under the current directory.

Task: Change (to another) directory

Example:

cd hw1
OR
cd ..
OR
cd foo/bar

cd allows you to go to another directory, and thus, change your current directory. Above, we first go into the directory named *hw1*, which must exist under the current directory. Then, we go to the directory above us (dot dot (**..**) is shorthand for *the directory above*). Finally, we go to a directory named *bar*, under a directory named *foo* under our current directory.

Task: Find out what is your current directory

Example:

pwd

pwd always reports the full path of your current directory.

Task: List files, directories or their information

Example:

ls
OR

ls hw1
OR
ls -l prog1.c

ls is used to list files (not their contents). For example, ls above list the files (and subdirectories) in the current directory. Similarly, ls hw1 lists what files are in the subdirectory *hw1*. In a slightly different usage, ls -l prog1.c lists information about the file prog1.c, such as how big it is etc. The **-l** (minus ell, not one) part of the command is a flag for the *ls* command.

Task: Look at the contents of a text file

Example:

cat hw1.c

Spits the contents of a text file on the screen.

Task: Look at the contents of a text file page by page

Example:

less hw1.c

To view a longer text file, use the **less** program followed by the name of the file you want to view, as in the example.

Less will allow you to page forward through the file using the **<SPACE>** bar and page backward using the **b** key.

When you are done looking at the file, type **q** to quit less.

Task: Copy a file

Example:

cp hw1.c hw1.c.bak
OR
cp testfile1 testfile2 ~/cse117

cp makes a copy of a file. This last argument specifies the destination name. The arguments preceding it are the source file(s).

When the destination is a directory, files with the same name as the source files are placed in that directory. In this case, any number of source files (to be copied) may precede the destination directory.

In both cases, since a *copy* is made, the original source file is left untouched.

When the command would cause a file to be copied over a file that already exists, you may be asked to confirm the copying command. To enforce this confirmation behavior, you may want to use the -i flag with the copy command:

cp **-i** source destination

In our first example, a backup copy of the file hw1.c is made--the copy's name is hw1.c.bak and will reside in the current directory.

In the second example, copies of the files testfile1 and testfile2 are made in the directory cse117, which resides under our home directory (i.e., referred to by the ~).

Task: Move (or rename) a file

Example:

mv writeup.bak writeup
OR
mv hw3.c hw3.h ~/homeworks

mv moves or renames files. The last argument specifies the destination. When the destination includes a path or is the name of a directory, files with the same name as the source files are placed in that path/directory. In this case, any number of source files (to be moved) may precede the destination.

In both cases, the original source file won't exist any more (except with their new names and possibly new locations).

When the command would cause a file to overwrite a file that already exists, you may be asked to confirm the move command. To ensure this confirmation behavior, You may want to use the -i flag with the move command:

mv **-i** source destination

In our first example, the file writeup is restored from a backup copy (which we could have made with cp) named writeup.bak. After, the file writeup.bak no longer exists. The new file writeup will end up in the current directory.

In the second example, the files hw3.c and hw3.h are moved into the directory homeworks, which resides under our home directory (i.e., referred to by the ~). There will no longer be files named hw3.c and hw3.h in the current directory.

Task: Remove a file

Example:

rm file1 file2

Removes the files. There is no easy way to *undelete* these files. They can be retrieved from a backup tape sometimes.

Task: Show the Manual Page (documentation) for a command (or function)

Example:

man ls
OR
man -s3m pow
OR
man -k pow

man gives documentation for a command, such as what it does, what flags it takes, etc. Man pages are a bit technical, but you get used to using them over time.

You can also look up C(++) library functions. While looking up a library function, as in:

man printf

Task: Compile with the *make* utility

Example:

make
OR
make -f hw1.mak

The *make* utility makes compiling more complicated programs easier, more consistent, and more efficient. You run the command *make* in the directory where your source code for a program exists.

By default, the make utility expects a file called a *make file* to be present in the same directory. This *make file* usually specifies how to compile your program. The *make* utility looks for a *make file* in a UNIX file named either Makefile or makefile in the directory where *make* is run. The -f option can be used to tell *make* to look for a different *make file* as in the second example.

Note the difference between a *make file*, which contains instructions on how to compile a program VS. the *make utility*, which is the program that does all the work (using a make file).

Make will display what commands it is using to compile, etc. as it goes along.

Task: Compile with the *gcc* compiler

Example:

gcc file1.c file2.c
OR
gcc -g -*o hw1-bank* hw1-bank.c -lX

gcc is the compiler we are using for this course. For more complicated programs, you may want to use the make utility for compiling.

The easiest way to compile a program (although not always the best way) is to type **gcc** followed by the names of all **.c** files that make up the program (as in the first example). By default, gcc creates an executable named **a.out**.

Caution: This may not work always work.

Flags
With gcc, you can use the following flags (others are listed in the Manual Pages for gcc).

For example, **-o** (that's oh, not zero) followed by the name for the executable. In the second example, **-o hw1-bank** has been used to call the executable **hw1-bank** instead of **a.out**.

The **-g** flag must be used when compiling code that you want to be able to debug with gdb.

The **-lX** flag at the end of the command tells gcc to link a library with your executable (in the example, the X Windows library). You'll normally be told when you must use the -l (minus ell, not one) option to link in a library.

Remember that compilers give *Warning*, which you should attend to, but that may not prevent your program from running, and *Errors*, which will prevent the compiler from generating an executable. Messages often give you the *file*, *line number*, and a *description* of the error as below.
sum.c:39: unterminated string or character constant
sum.c:32: possible real start of unterminated constant

Resolve the first errors in the code first since they often cause later errors.

Task: Debug executable compiled with *gcc*

Example:

gdb a.out

Gdb is the debugger that goes with the gcc compiler. You always run the debugger on the executable for a program.

To be able to debug an executable, its source code must have been compiled using the -g flag of gcc.

The debugger will allow you to run the program, stop it at certain points, and examine/change variables. It also tells you what the offending line is when a program crashes in the debugger.

When you run the debugger on an executable, make sure that the source code files for the program are in the same directory.

Task: Edit or create files with Emacs

Example:

emacs hw1.c &
OR
emacs &

Emacs is an editor for text files. Thus, you'll probably want to use it to create and edit your source code files.

If you give a file name with the *emacs* command, as in the first example, it will load that file if it exists, or start a new one using that name if it does not. Since *emacs* brings up a window, you should run it with the ampersand (**&**).

When you are editing in *emacs*, you are editing a **copy** of the file, so remember to save edits to disk before you exit *emacs*.

Task: Use **<** (input redirection)

Example:

a.out **<** data1

The *less than (<)* character makes an executable take input from a file instead of waiting for something to be typed at the keyboard. This is called input redirection.

For example, if data1 above contained the following:

2
3.4
4.5

Then, the program a.out would behave as if we typed **2**, then hit **<RETURN>**, typed **3.4**, then hit **<RETURN>**, and finally, typed **4.5** and hit **<RETURN>**.

Task: Use **>** (output redirection)

Example:

a.out **>** output1
OR
gcc file.c **>&** output_and_errors
OR
a.out **>>** output1

The *greater than (>)* character sends the output of an executable to a file instead of putting it on the screen. This is called output redirection.

For example, if the executable a.out above normally prints out "Hello, world!" on the screen, then the first example would cause nothing to appear on the screen, but instead, "Hello, world!" would be found in the file output1.

Some output from programs is meant as error messages. Sometimes these will not be redirected by *greater than (>)*. To redirect both regular and error output to a file, use the ampersand after the greater than symbol (i.e., **>&**), as in the second example.

When you use output redirection, the first thing UNIX does is to create (or clear out, if it exists) the output file (e.g., output1 and output_and_errors in our examples). Be careful with this, since if you try to do something like:

cat **a** b c > **a**

You may run into problems since **a** gets cleared out (the "**> a**" part) before it is used (the "**cat a b c**" part).

When you don't want the output file cleared out first, but rather, want the output of the command to be appended to the end of the file, use 2 greater thans (**>>**), as in the third example. Using the ampersand to redirect both regular and error output also works with appending (i.e., **>>&**).

Task: Use | (piping)

Example:

a.out | sort
OR
gcc file.c **|&** less
OR
a.out | sort | less

The *pipe symbol (|)* sends the output of an executable as the input of the command that follows it, instead of putting the output on the screen. This is called piping.

For example, if the executable a.out above normally prints out a bunch of names on the screen, then the first example would cause those names to go as input to the sort command, and then sort would print out the names in sorted order on the screen.

Some output from programs is meant as error messages. Sometimes these will not be piped by |. To pipe both regular and error output to another program, use the ampersand after the pipe symbol (i.e., **|&**), as in the second example, where the output of the compiler gcc is paged with less.

You can string along several commands with pipes. In the third example, the output of the program a.out is first sorted and then paged with the program less.

Task: Use *Up* and *Down* arrows (command history)

On our system, you can use the *up and down arrows* to scroll through the commands that you typed during the same login session.

This is an easy way to retype or edit commands you've already typed in.

(Please Note: This tutorial has been taken from CSE, IITM.)

Exercise Problems

1. Make a directory ***CSML***. Make another directory inside CSML with your roll_no (eg 12345)

2. What do you think the following commands would list ?
   % ls ~
   % ls ~/..

3. Create a text file ***hw.txt***. Create a backup of ***hw.tx*t** file by copying it to a file called ***hw.bak***.

4. Create a directory called ***tempdir*** using *mkdir* , then remove it using the *rmdir* command.

5. Create a text file *sample.txt* with some content in it. What would the following commands display?
   % cat sample.txt
   % less sample.txt
   % head sample.txt
   % head -5 sample.txt (What difference did the -5 do to the head command?)
   % tail sample.txt

6. What did the following commands print?
   % grep the sample.txt
   % grep -i the sample.txt
   % grep -i 'in the' sample.txt