Tutorial for Basic Unix Commands
Bob Myers
Computer Science Dept., Florida State University

This tutorial is intended to help you become familiar with the basic set of unix commands used for moving around the file system and manipulating files and directories. A simple list of basic commands can be found from the main course web site -- www.cs.fsu.edu/~myers/cop3330

---

1) Your Home Directory

Once you log in to your unix account, you will have a specific home directory, usually identified by your user name. After you log in, you will see a command line prompt. Try the command:

```
pwd
```

This command returns the name of your current working directory (i.e. the directory you are currently in). When you log in, you should be located in your own home directory. For example, my user name is "myers", and my home directory is:

```
/home/faculty/myers
```

2) Getting a Directory Listing

In DOS, the command to show the contents of a directory is "dir". In unix, the equivalent command is "ls". Try it:

```
ls
```

The listing you see is just a plain and simple listing of the contents of the current working directory. This is like opening a folder in Windows and looking inside it. You can get more information by specifying some parameters on the ls command. Try this one:

```
ls -l
```

This command gives a longer listing with more information. The result may look something like this:

```
drwx------  14 myers    gs           1024 May 10 10:41 Mail
drwx------   2 myers    gs            512 Mar  1 17:23 News
-rw-r--r--   1 myers    gs            151 Feb 26 00:27 file1.txt
-rw-------   1 myers    gs            317 Apr 30 02:09 code.cpp
drwx------   5 myers    gs            512 Dec 15 01:34 arch
drwx------   2 myers    gs            512 Apr 19 10:49 archive
drwx------  18 myers    gs            512 May  8 12:01 cgs4406
drwxr-xr-x   7 myers    gs            512 May  4 00:37 public_html
```

In the above sample listing, there are many pieces of information. Each line represents one directory or file. The first character of the line indicates what it is. If the first character is a "d", the item is a directory (equivalent to a "folder" in Windows). If the first character is a dash "-", then the item is simply a file. Note that the item "code.cpp" is a file, while "public_html" is a directory.

The next set of characters indicate file permission bits (which will be discussed later). Other information includes the owner of the file, any group that the file belongs to, the file size, the most recent modification date and time (last time it was changed), and the file or directory name itself. A shortcut for this command that works on

shell.cs.fsu.edu is:

```
ll                    (this is not always set up on some unix systems)
```

3) Creating and removing directories

The "mkdir" command stands for "make directory". This is like creating a new folder in Windows. The "rmdir" command stands for "remove directory". This is like deleting a folder in Windows.

Let's make some directories. Try the following commands:

```
mkdir dog
mkdir temp
mkdir cgs4406
mkdir Dog
mkdir mystuff
```

Now list the contents of the current directory:

```
ls -l
```

Notice that "dog" and "Dog" are two different directories! Unix is case sensitive (unlike DOS/Windows), so upper and lower case words are distinct names.

Now try removing a few directories:

```
rmdir Dog
rmdir temp
```

Now do a directory listing and see what's left.

```
ls -l
```

Conclusion: Creating and deleting directories is EASY!

4) Moving to different directories

To move around to different working directories, you will use the "cd" command. This stands for "change directory", and it is very similar to the DOS cd command.

Try moving into the mystuff directory you just created:

```
cd mystuff
```

Use the pwd command to see the full path name. It now includes the new directory you are in:

```
pwd
```

Now make another directory called "letters" and move into it:

```
mkdir letters
cd letters
pwd
```

Now try the command "cd" just by itself, and then see where you are:

```
cd
pwd
```

You should be back in your home directory! The cd command by itself always takes you home. Now just repeat after me, "There's no place like

already called you home! Now just repeat after me, "There's no place like home!"

You can specify where you want to go by using an entire path name. For example, the following command takes you to where the C library files are stored:

```
cd /usr/include
```

Type cd to go home:

```
cd
```

You can also specify where to go from a starting point in your own home directory by using the ~ as the first directory entry:

```
cd ~/mystuff/letters
```

You will be back in the letters directory we recently created. The ~ is a special symbol that can substitute as your home directory.

The phrase .. means go back a directory. Try this:

```
cd ..
pwd
```

You should now be in the mystuff directory.

You can also mix and match, separating directory names by the forward slash / (as opposed to the backslash \ that DOS uses):

```
cd ../cgs4406
```

This command takes you back a directory and forward into cgs4406
Now try:

```
cd ../mystuff/letters
```

This backs up one and moves forward through mystuff and into letters.

Try making some new directories and moving around in them with the cd command.

5) Creating files with the pico text editor

One useful tool available on many unix systems is the program "pico", which is a simple text editor. If have ever used the mail programs "elm" or "pine", you have probably used pico without realizing it. pico is the standard text editor used with these mail programs.

Go to the letters directory and try creating a new pico document:

```
cd
cd mystuff/letters
pico memo
```

The last command will open up a pico session, using the filename "memo". Type the following text:

```
The quick brown fox jumped over the lazy platypus.
```

Now hold the control key and type 'x' (control-x) for exit. You will be asked if you want to save. Answer yes and get out of pico. Now, take a look at the contents of your directory:

```
ls -l
```

You should now see a new file called memo. Congratulations! You just made a file!

Let's make another:

```
pico hello.cpp
```

This opens up a new pico file. Type in the following program:

```
#include <iostream>
using namespace std;

int main()
{
   cout << "Hello, World!\n";
}
```

Now exit pico and save the file. Look at your directory:

```
ls -l
```

You should see two files listed: "memo" and "hello.cpp"

You can edit an existing file by opening it up in pico again:

```
pico memo
```

The file will open up and you can pick up where you left off. Type something else and then exit pico and save.

6) Copying, Moving, Renaming

The basic formats for the copy command (cp) and the move command (mv) are as follows:

```
cp <source> <destination>
mv <source> <destination>
```

In the above, <source> can refer to the name of the source file alone, or the path and filename of the source file. Either way, this part refers to the original that is being copied or moved. <destination> can refer to either a new destination filename, a new destination location (i.e. a different directory), or a combination of both.

First a word about combining directory paths and filenames. We've already seen that a directory path is a series of directories separated by the slash character, like this:

```
/usr/include
~myers/mystuff/letters
```

A file can be referred to by just its filename, as long as it is in the current working directory:

```
hello.cpp
```

If you want to refer to a file by name, but you are in a different working directory, then simply attach the filename to the end of the path name, again with a slash as a separator:

```
~myers/mystuff/letters/hello.cpp
```

Now, let's try some copying. Go to your new directory mystuff/letters, so that you are in the directory that has the new files "memo" and "hello.cpp". Now, let's make a copy of the memo file, and call it memo2:

```
cp memo memo2
```

Now do a directory listing and see that the new file memo2 has been created. It should be exactly the same size as memo, and it should have a later file modification date. Check the contents of the file by opening it with pico:

```
ls -l
pico memo2
```

Now, let's say you want to make a copy of hello.cpp, but you want the copy in a different directory. First, let's make a new directory called "temp", and then copy hello.cpp to it:

```
mkdir temp
cp hello.cpp temp
```

Notice that in the last command, the <source> was a filename, but the <destination> was a directory name. That's okay. Since the destination already existed as a directory, the copy command simply placed a copy of hello.cpp (with the same filename) into the temp directory. Let's verify this by doing a listing of the temp directory. Instead of moving anywhere, use the following command:

```
ls -l temp
```

This should show you a listing of the contents of the temp directory (another option when using the ls command). You should see the hello.cpp file here.

Now let's make a copy of hello.cpp, but let's change the filename and put the copy into the temp directory:

```
cp hello.cpp temp/prog1.cpp
```

Notice the way that the <destination> is now a combination of path and filename. Check the contents of the temp folder again:

```
ls -l temp
```

You should see hello.cpp and prog1.cpp.

The move (mv) command works just like the copy command in terms of the format. There is one difference. When you use the mv command, the <destination> file is created (just like a copy), but the <source> file is deleted. Let's try some moves:

```
mv memo mymemo.txt
ls -l
```

One important aspect of the mv command is that it also serves as a "rename" command. In the above mv command, we just moved the file "memo" to a new filename, "mymemo.txt", without changing it's location. Now try this one:

```
mv mymemo.txt temp
```

This moves the file's location into the temp folder. This is similar to the copy we did before, but if you do a directory listing of the current directory, you'll see that the mymemo.txt file is gone. However, try listing the contents of the temp directory, and you'll see it is now there:

```
ls -l temp
```

Now try this one:

```
   mv temp/prog1.cpp .
```

Notice the space and the period at the end of the command. The <source> in this command is the path "temp" coupled with the filename "prog1.cpp". The <destination> parameter is simply the single period. In unix, the single period always means "right here, in the current working directory". So, this command should have moved the prog1.cpp file into your current directory. Let's verify:

```
   ls -l
```

Looking at the current directory should reveal that the prog1.cpp file is now here.

```
   ls -l temp
```

Looking into the temp directory should show you that the prog1.cpp file is gone from there, but the mymemo.txt file is now there.

7) Deleting files

The command "rm" stands for remove, and this command will delete files. The format is simply:

```
   rm <filename>
```

Let's try it. First, let's make a copy of the prog1.cpp file:

```
   cp prog1.cpp junk.cpp
   ls -l
```

Doing a listing will show that junk.cpp is now in the current directory. Now, let's remove it:

```
   rm junk.cpp
   ls -l
```

Now it's gone!

8) Wild Cards

In any unix commands that deal with paths and file names, you can use wild cards. The wild card characters are the question mark (?) and the asterisk (*).

The question mark ? can be substituted in a filename, and it will represent any character in that position. Let's set up a situation to illustrate it's use. You should still be in the "mystuff/letters" directory. Let's make some copies of the prog1.cpp file that should be sitting in the current directory. Type the following commands:

```
   cp prog1.cpp prog2.cpp
   cp prog1.cpp prog3.cpp
   cp prog1.cpp prog4.cpp
   cp prog1.cpp prog5.cpp
```

Now, let's move all five of these files into the temp directory, but with one command! We can achieve this quickly with the wild card:

```
   mv prog?.cpp temp
```

The ? in the above command says that the command applies to any filename matching that pattern, in which the ? can represent any character (like a wild card in a poker hand). Let's verify it. List the current directory.

Then list the temp directory:

```
ls -l
ls -l temp
```

Where are the files? They should now be in the temp directory.

The asterisk is a more powerful wild card character. It works like the question mark, but it represents any number of characters in that position. Let's move the files back from temp. Use the following command:

```
mv temp/prog* .
```

Notice again that the command ends with a space and a period. The <source> is "temp/prog*". This means that the command applies to any filename that starts with prog and ends with ANY sequence of characters. Do directory listings again and see where the files are:

```
ls -l
ls -l temp
```

The .cpp files starting with prog should now be in the current working directory.

Now try this:

```
cp * temp
ls -l temp
```

The <source> parameter was now simply just *. This means "all files in the current directory". We just asked the machine to copy all of the current files into the temp directory. Here's a very nice way of periodically making backups of your work. Just copy everything into another directory.

Now try this, but make sure you are still in the letters directory (and not somewhere important):

```
rm *
ls -l
```

When you use * with rm (remove), you are asking the system to delete EVERY file in the current directory. Fortunately, this only removes files, and not directories (remember you need rmdir to remove directories). However, be VERY careful with this! You don't want to accidentally remove important files! Your current directory should now only have the temp directory inside it. However, remember that backup we just made into the temp directory? Whew! All of our files are safe!

9) E-mail from a unix command shell

Many unix systems allow access to e-mail programs from the command line. Two of the most common are "elm" and "pine". To access these, simply type the name of the program:

```
elm

pine
```

Try these. They may ask you if you want to establish default mail folders when you first run them. Go ahead and answer yes, and explore the commands of these programs. Both have good help files (the pico editor does, too). You may find that pine is a little more intuitive and user-friendly.

10) Man pages

This is not a place where men go to strut around. man refers to
"manual", and unix systems provide pages of the manual that can be
accessed from the command prompt. You can get further information on any
unix command or system call with the command:

   man <commandname>

Sometimes the man page has lots of information on it that you really don't
need at the moment, and some of them can be downright confusing in places.
However, they contain all of the necessary information about the usage of
a command, including the available parameters. Some of the better man
pages give examples of a command's usage. Try accessing a few man pages.
Don't worry when you see <more> at the bottom of the screen. Just press
the spacebar for the next screen full, or press the return key for the
next single line.

   man ls
   man cd
   man mv

Give these a try and check out the manual pages that come up.

The commands discussed above comprise the most basic set of commands that
are necessary for getting around the unix file system. The items
discussed below are mostly other useful commands and/or a few more
advanced items. These are not necessary for getting around, but can be
useful to know.

Note to C++ students: For now, make sure you are familiar with the
commands presented above. I may add material to this section later on.

Bob Myers

---

11) File Permissions

You have control over the permissions on your own directories and files.
you can limit permissions to just yourself, or you can allow other people
to access your files and run your programs. You can do this on a file by
file basis, if you like. To change your file permissions, you use the
chmod (change mode) command. You can see a full description of chmod in
the manual pages:
        man chmod

There are several ways to use the chmod command, but the easiest format
is:

        chmod <permission_list> <file_or_directory_name>

The permission list can be given with a numerical code. The basic format
is to use a 3 digit number (4 digits are possible, for more advanced
options).

When you do a long format directory listing, the information about
permissions is on the left side of the screen:

```
drwx------   6 myers    gs           512 Sep 19 21:49 project
drwxr-xr-x   7 myers    gs           512 Oct 13 11:09 public_html
drwx------   2 myers    gs           512 Jan 23  1999 review
-rw-------   1 myers    gs           257 Oct 19 16:01 samp.cpp
```

```
-rwxrw----   1 myers    gs          74240 Mar 27  2000 sem1a.ppt
drwx------   2 myers    gs            512 Jul 25 22:49 sw
drwxr-xr-x   6 myers    gs           1024 Aug  7 22:57 sysadmin
-rw-r--r--   1 myers    gs             81 Oct 23 13:57 t1

 ^^^^^^^^^
 These are the permission bits
```

The permissions are shown using 9 spots. The first position to the left tells what type of item it is. The letter 'd' means directory. A dash means that it is just a regular file.

The next 9 spots, the permission bits, can be grouped into 3 sets of 3. Each set of 3 shows the read, write, and execute permissions. Here is what the letters mean:

- r - read permission is allowed
- w - write permission is allowed
- x - execute permission is allowed

If read is allowed, you can view or copy a file. If write is allowed, you can actually modify a file. Execute permission allows you to run a program (if the file is a program or an executable script) or to open and view a directory (if the file is a directory). A dash '-' in a position means that privelege is NOT allowed.

Each group of 3 does these in order -- read, write, execute.
Examples:

- rwx = permission to read, write, and execute is given
- r-x = permission to read and execute (but not write)
- --- = no permission to this file
- r-- = read only

There are 3 sets of these. The first set applies to the owner of the file. The second set applies to any groups the owner belongs to. The third set applies to everybody else. So, the three groupings, in order, are: owner permissions, group permissions, everybody else's permissions.

```
drwx------   6 myers    gs            512 Sep 19 21:49 project
drwxr-xr-x   7 myers    gs            512 Oct 13 11:09 public_html
drwx------   2 myers    gs            512 Jan 23  1999 review
-rw-------   1 myers    gs            257 Oct 19 16:01 samp.cpp
-rwxrw----   1 myers    gs          74240 Mar 27  2000 sem1a.ppt
drwx------   2 myers    gs            512 Jul 25 22:49 sw
drwxr-xr-x   6 myers    gs           1024 Aug  7 22:57 sysadmin
-rw-r--r--   1 myers    gs             81 Oct 23 13:57 t1
```

So, in the listing above:

The file called "samp.cpp" has the permission set: rw-------
This means that the owner has read and write permission to the file, but nobody else has any permissions.

The file "t1" has permission set: rw-r--r--
This means that the owner has read and write permission, and everybody else (groups and others) has read only access.

The directory "public_html" has permission set: rwxr-xr-x
So, the owner has full access (read, write, execute), and anybody else (group and other) has read and execute permission (but not write).

Seting the Bits:

To set the permissions, we can take each grouping (rwx) and assign a value to each letter. r = 4, w = 2, x = 1. No permission (a dash) gets a 0 value. To obtain the number for the desired grouping, add the values of

the desired permissions together:

```
  For rwx ,  4 + 2 + 1 = 7
  For r-x ,  4 + 0 + 1 = 5
  For r-- ,  4 + 0 + 0 = 4
```

(Essentially, this is done with a binary number with 1's in the set positions and 0's in the off positions. r-x = 101 binary = 5 decimal, for instance).

Now, do this for the three groupings (owner, group, other), and put the numbers together. This code goes into the chmod command.

Examples:

```
  chmod 755 file1
```

This sets the permissions on "file1" to 7 for the owner (rwx), and 5 for group and other (r-x), so the whole permission set on the file would be rwxr-xr-x.

```
  chmod 640 file2
```

This sets the permissions on "file2" to 6 for the owner (rw-), 4 for the group (r--), and NO permission for others (---). Here is the permission set: rw-r-----

---

12) more and less

The "more" command in unix allows you to display a file one screenful at a time. After each screenful, you will see "---More---" at the bottom. You can advance the file one line at a time with the Return key, or you can advance a page at a time with the Spacebar.

Example: To view a code file, one screen at a time:

```
   more program1.cpp
```

"less" is a command that is similar to "more", and is available in some unix systems. It allows the viewing of a file, like "more", but it also allows backward movement through the file while viewing, as well as forward movement.

```
   less program1.cpp
```

To find out about all of the features of more and less, see the manual pages:

```
   man more
   man less
```

---

13) The pipe

The vertical bar | is called a "pipe" in unix, and it has a special power. A pipe allows the output of one program to be re-routed as input into (or "piped into") another program. This can be useful in many situations.

Examples:

- Suppose you run a program called "prog1" that sends output to the screen that scrolls by faster than you can read it. Then, just "pipe" the output of the program to "more" to read one screen at a time:

of the program to "more", to read one screen at a time.

```
prog1 | more
```

- Suppose that you do a directory listing, and there are so many files in your directory that the long listing scrolls past the screen so that you can't see the earlier entries in the listing. Again, you can pipe it to "more":

```
ls -l | more
```